

# perClass Mira Documentation

## Table of contents

---

Introduction .....	6
Getting started .....	7
perClass product structure .....	7
Installation and license activation .....	7
Activation .....	11
Build classifier on existing scans .....	12
Creating a project .....	12
Adding images .....	13
Spectral cube visualization .....	15
Training a classifier .....	16
Switching between labels and decisions .....	19
Improving the classifier .....	20
Improving the labeling .....	20
Adding new classes .....	22
Acquire data and interpret .....	25
Creating a project for acquisition .....	27
Connecting to the stage .....	28
Connecting to the camera .....	30
Recording references .....	31
Defining a scan area .....	32
Recording a scan .....	33
Building a classifier and applying to live data .....	35
User guide .....	38
New project .....	38
Objects .....	39
Object segmentation .....	40
Object separation .....	41
Object classification .....	43
Regions .....	44
Region annotation .....	44
Confusion matrix .....	46
Test set confusion matrix .....	48
Current image confusion matrix .....	51
Optimizing classifier performance .....	52
Cost sensitive optimization .....	52
Performance constraints .....	53
Object confusion matrix .....	57
Detailed information on object matching .....	59
Copying confusion matrix .....	60
Visualization (spectral indices) .....	62
Adjusting spectral features .....	64
Scaling spectral features .....	66
Applying feature extraction to foreground .....	67
Colormaps .....	68
Feature extraction (exporting) .....	71
Extracting multiple features .....	74
Extracting from region grid .....	76



Defining region extraction template .....	77
Exporting into XML .....	78
Regression .....	78
Step 1: Pixel classification .....	79
Step 2: Object segmentation .....	80
Step 3: Object annotation .....	80
Step 4: Regression modeling .....	83
Step 5: Defining test data set .....	85
Step 6: Improving regression model .....	87
Regression plot .....	87
Performance statistics .....	88
Outlier plot .....	90
Error plot .....	90
Regression using subset of bands .....	92
Regressor and classifier band subsets .....	94
Preprocessing .....	94
Additional regression tools .....	96
Model search versus retraining .....	96
Applying to new images .....	96
Pixel visualization of regression output .....	98
Spectral plot .....	100
Class-specific display .....	100
Display range and scaling .....	101
Band selection .....	103
Band subsets used by models .....	105
Frame widget .....	106
Stage .....	108
Stage commands .....	109
Camera .....	110
Camera controls .....	110
Recording panel .....	111
Recommended screen setup .....	113
Setting scan name .....	113
Exporting .....	114
Exporting per-image results .....	115
Exporting per-object results .....	116
Exporting visualizations .....	117
Exporting visualizations as float images .....	118
Exporting cubes .....	120
Exporting regions .....	121
Importing regions .....	122
Exporting label images .....	124
Model testing .....	125
Flagging images for testing .....	126
Cross-validation .....	127
Cross-validation over images .....	129
Cross-validation over replicas .....	131
Default action .....	134
Reference .....	135
Integration .....	135

Example of acquisition from Camera API .....	136
Release notes .....	139
Application Server .....	149
Enabling application server .....	150
Communicating with the server .....	151
Command list .....	153
Example communication using Tcl .....	155
perClass Stage .....	156
Assembling instructions .....	157
Disassembling instructions .....	158
Supported cameras .....	160
Supported spectral cameras .....	160
Cubert .....	161
Pleora eBUS .....	162
Headwall .....	162
Headwall MV.X .....	162
Headwall MV.C NIR .....	163
Headwall MV.C VNIR .....	163
Resonon .....	163
perClass Camera API .....	164
miraacq_Init .....	164
miraacq_GetVersion .....	165
miraacq_GetAPIVersion .....	165
miraacq_GetRecorderType .....	166
miraacq_GetErrorCode .....	166
miraacq_GetErrorMsg .....	166
miraacq_ScanDevices .....	167
miraacq_GetDeviceCount .....	167
miraacq_GetDeviceName .....	167
miraacq_OpenDevice .....	168
miraacq_CloseDevice .....	168
miraacq_DeviceIsSnapshot .....	168
miraacq_InitializeAcquisition .....	169
miraacq_GetFrameSize .....	169
miraacq_GetFrameWidth .....	169
miraacq_GetFrameHeight .....	169
miraacq_GetFrameBands .....	170
miraacq_GetFrameDataType .....	170
miraacq_GetFrameDataLayout .....	170
miraacq_CanReturnWavelengths .....	171
miraacq_GetFrameWavelength .....	171
miraacq_SetResamplingWavelengthCount .....	171
miraacq_SetResamplingWavelength .....	172
miraacq_SetResampling .....	172
miraacq_StartAcquisition .....	172
miraacq_GetFrame .....	172
miraacq_StopAcquisition .....	173
miraacq_SetExposure .....	173
miraacq_GetExposure .....	173
miraacq_SetFrameRate .....	174

miraacq_GetFrameRate .....	174
miraacq_Release .....	174
perClass Mira Runtime API .....	175
mira_Init .....	176
Error codes .....	176
mira_GetVersion .....	177
mira_GetErrorCode .....	177
mira_GetErrorMsg .....	177
mira_RefreshDeviceList .....	178
mira_GetDeviceCount .....	178
mira_GetDeviceName .....	178
mira_SetDevice .....	178
mira_LoadModel .....	179
mira_LoadCorrection .....	179
mira_SetMinObjSize .....	180
mira_SetSegmentation .....	180
mira_GetInputWidth .....	181
mira_SetInputWidth .....	181
mira_GetInputHeight .....	181
mira_GetInputBands .....	181
mira_GetInputDataType .....	182
mira_GetInputDataLayout .....	182
mira_GetMaskType .....	182
mira_StartAcquisition .....	183
mira_ProcessFrame .....	183
mira_ProcessCube .....	183
mira_StopAcquisition .....	184
mira_GetFrameDecisions .....	184
mira_GetDecCount .....	184
mira_GetRegVarCount .....	184
mira_GetRegVarName .....	185
mira_GetObjDataRegOutput .....	185
mira_GetFrameRegOutputVar .....	186
mira_GetDecName .....	187
mira_GetDecColor .....	187
mira_GetObjCount .....	187
mira_GetObjDataInt .....	188
mira_GetObjDataClassSize .....	189
mira_GetObjDataClassFrac .....	189
mira_SaveImage .....	189
mira_Release .....	190
Troubleshooting .....	190
How to enable logging .....	191

## Introduction

perClass Mira® is the easiest user interface for interpretation of spectral images with real-time deployment.

It is a comprehensive collection of tools empowering users to

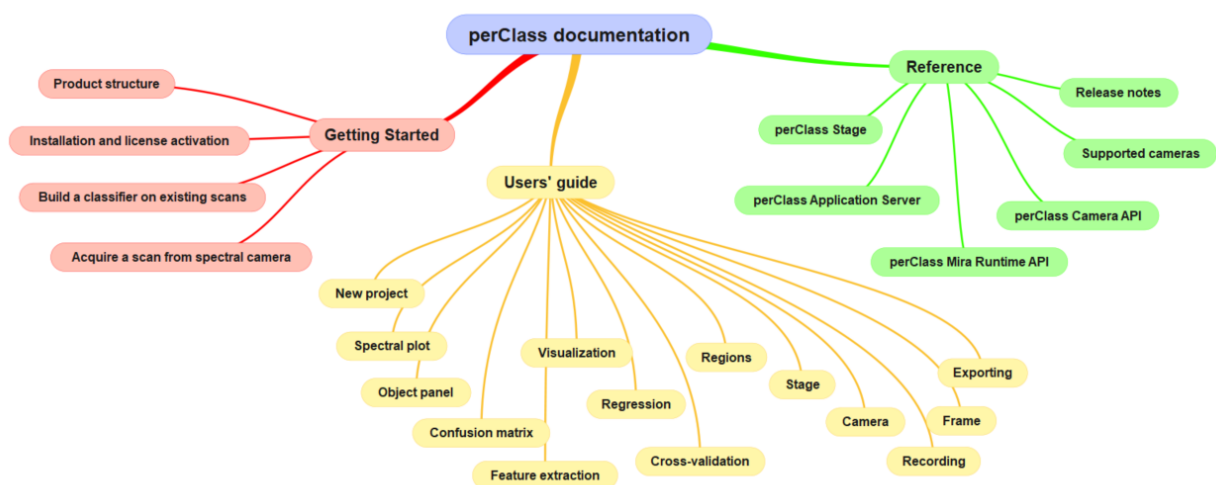
- Acquire high quality spectral images using a range of multi- and hyper-spectral cameras
- Build classification and regression solutions
- Export specific information from detected objects or user-defined regions
- Batch process new scans using user-defined solutions
- Deploy solutions to a live data stream and integrate in production

It is an application development tool for industrial and applied researchers who leverage spectral imaging as a tool. Specifically, two use-cases are optimized:

1) *Development and deployment of smart sorting machines by system integrators* . Example application is nut sorting, fruit quality grading or foreign object detection machines in food industry.

2) *Processing of large number of spectral scans extracting relevant information of further research*. Example application domains are plant phenotyping and food quality

perClass Mira is tightly integrated with [perClass Stage](#), a linear lab-scanning kit supporting a range of spectral cameras.



perClass documentation is structured in the following way:

- The [Getting Started chapter](#) guides you step-by-step how to
  - Understand [perClass product structure](#)
  - [Install the software and active its license](#)
  - [Build a classifier for spectral images you already have stored in files](#)
  - [Acquire new scans from spectral camera and interpret them](#)
- The [Users' guide](#) describes each software component in detail.
- The [Reference](#) provides detailed information on
  - [Software integration](#)
  - [Software release notes](#)
  - [Supported sensors](#)
  - [perClass Stage](#)

- [perClass Mira Runtime API](#)
- [perClass Camera API](#)
- perClass Application Server API

## Getting started

---

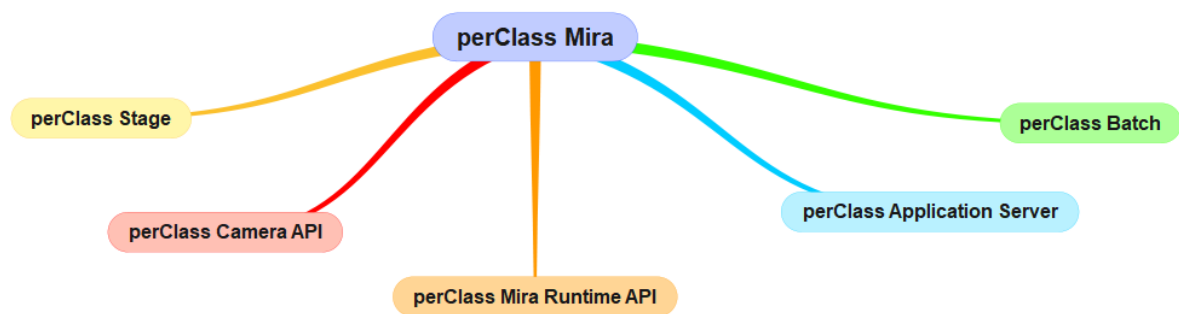
This document [describes perClass product structure](#) and provides quick guides to:

- [Build a classifier on existing scans](#)
- [Acquire new data from a spectral camera and interpret it using a classifier](#)

### perClass product structure

perClass is a collection of multiple tools including

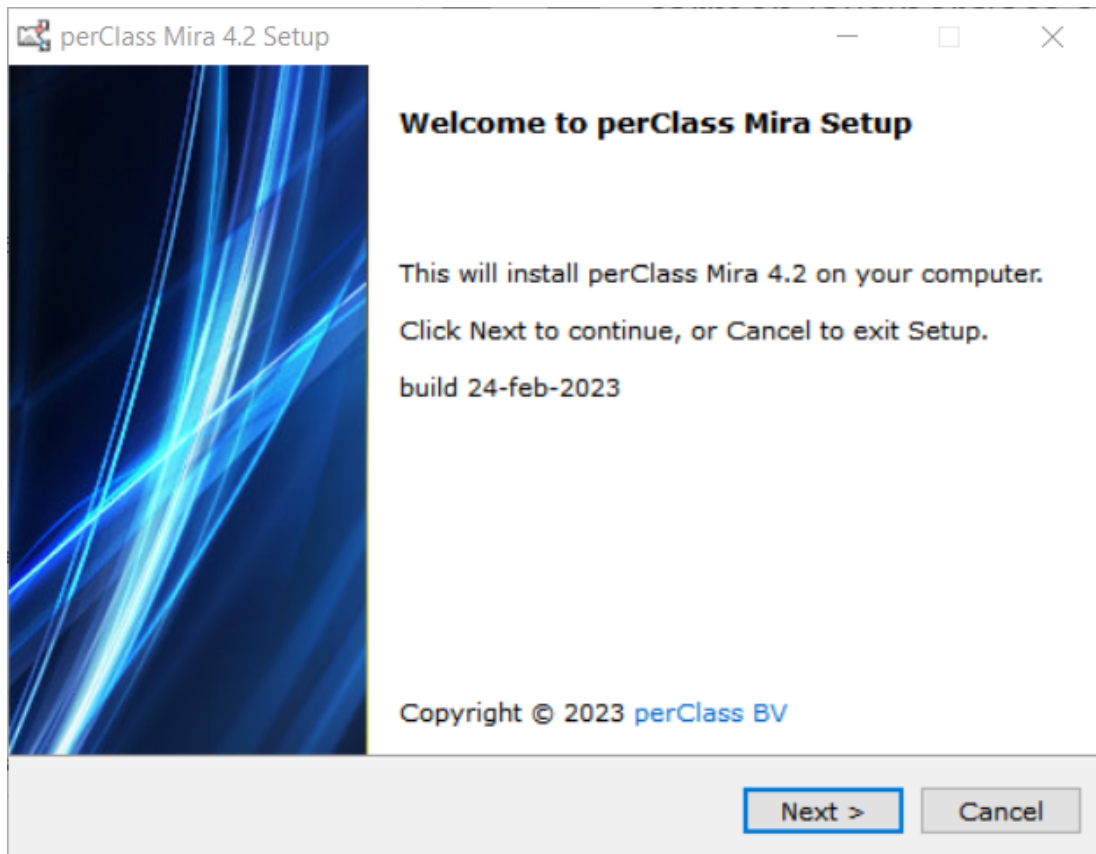
- perClass Mira User interface emporing users interpret spectral images
- [perClass Stage](#), a hardware lab-scanning kit supporting different spectral cameras.
- [perClass Camera API](#) connecting to different spectral cameras
- [perClass Mira Runtime API](#) to embed solutions developed in the perClass Mira to custom applications
- [perClass Application Server](#) allowing one to build live demonstrators by remotely controlling perClass Mira GUI and acquisition via text commands
- perClass Batch processor allowing one to apply solutions to new scans without launching the perClass Mira GUI



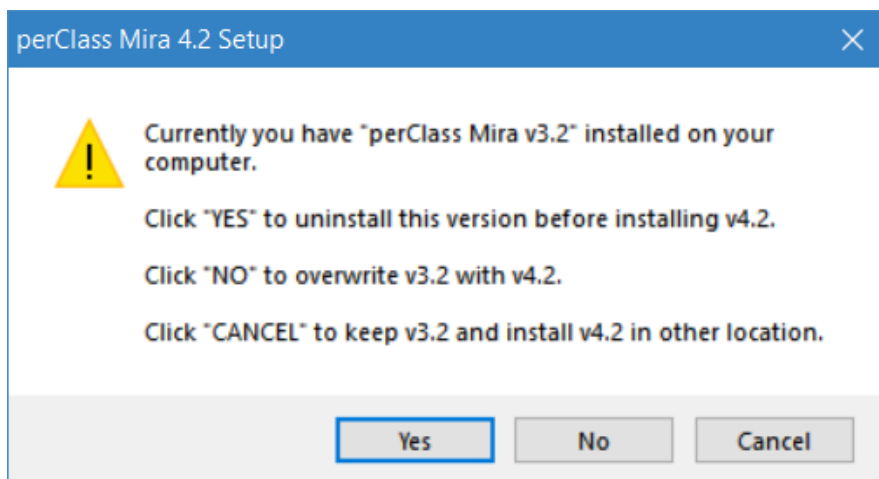
### Installation and license activation

This section describes how to install perClass Mira software and activate its license.

Run the installation file, the installer dialog will appear:

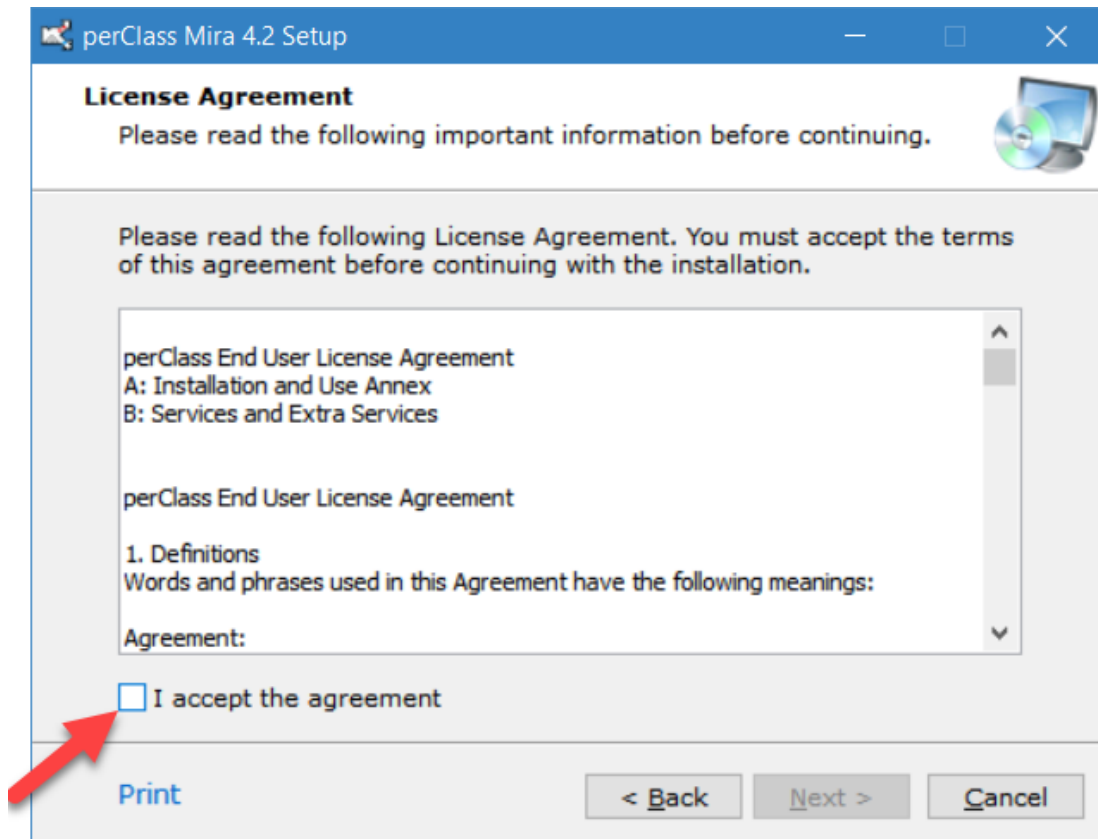


In case you already have an existing perClass Mira installation, the following dialog will also appear:

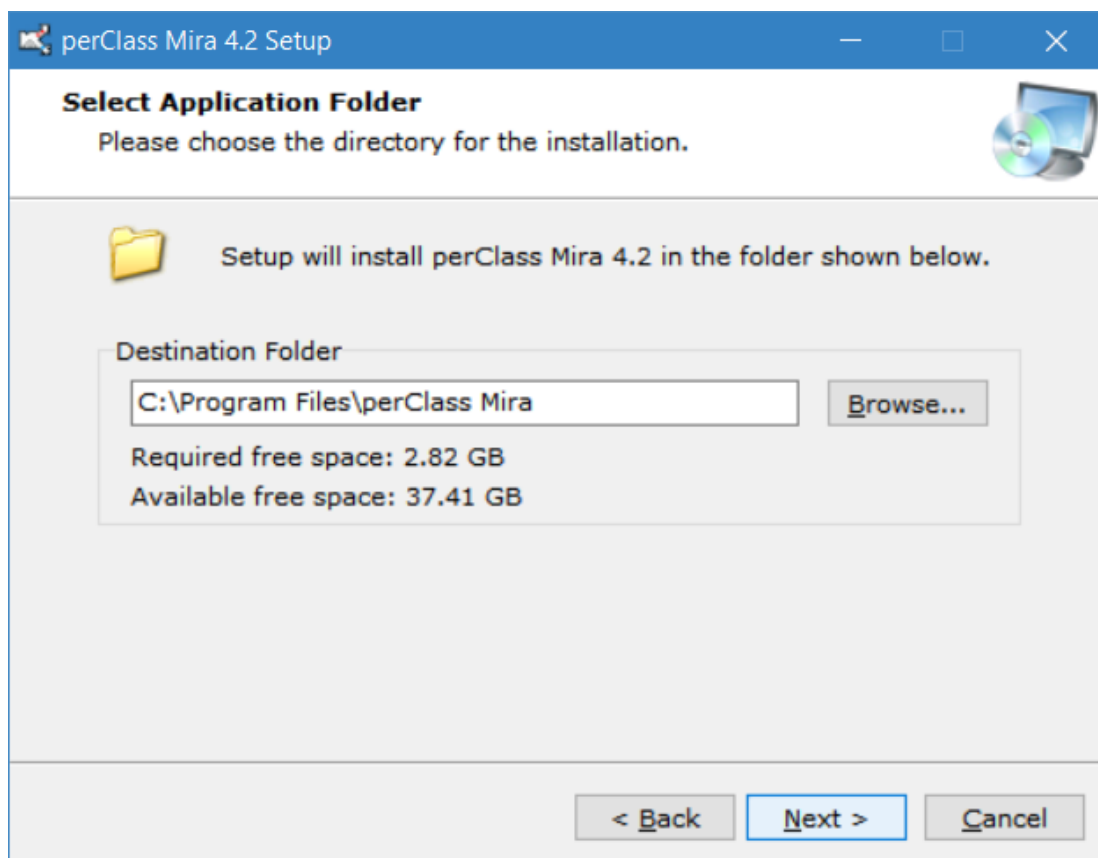


If you wish to install the new release into a directory you specify, click on *Cancel* button.

You need to accept end-user license agreement before proceeding:

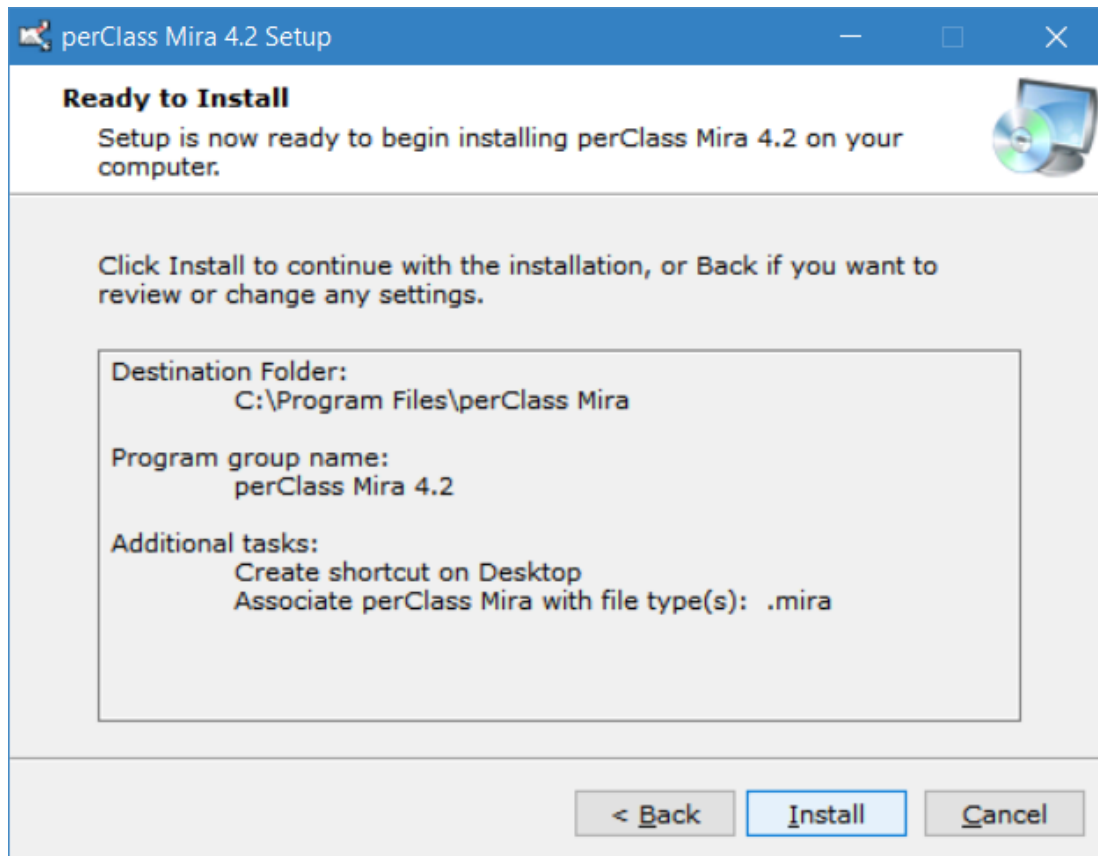


If you selected to customize location, you may now specify the installation directory:

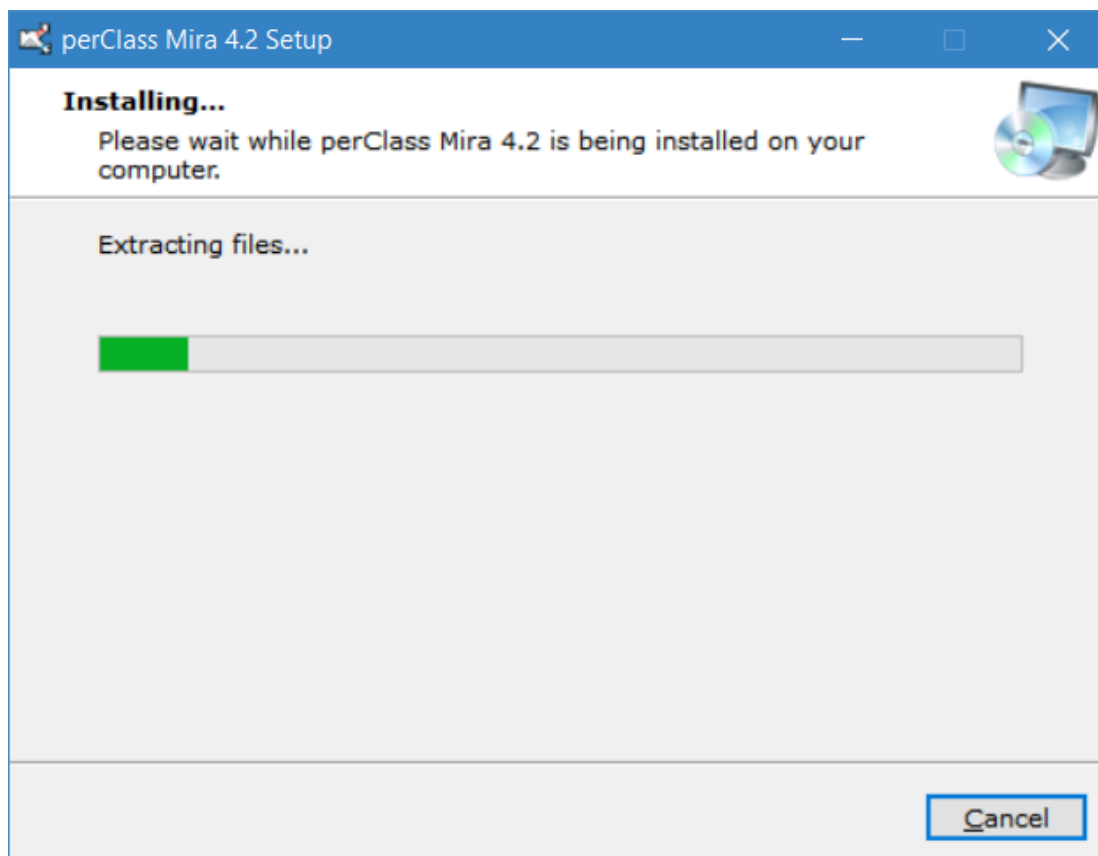


**TIP:** You may freely install multiple releases of perClass Mira on the same computer. If you, it is recommended to use full version including date in the file name, for example: "perClass\_Mira\_4.2\_24feb23"

In the last step, you may confirm or cancel the installation:

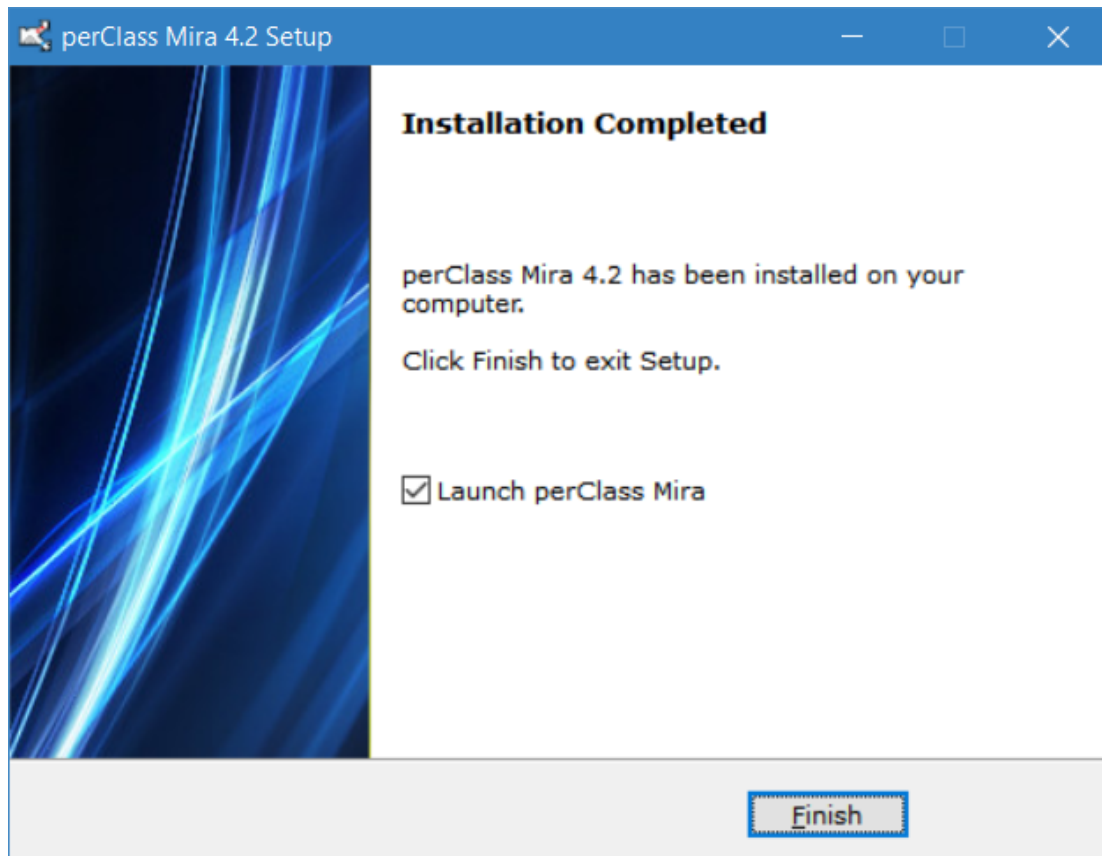


After extracting the files:



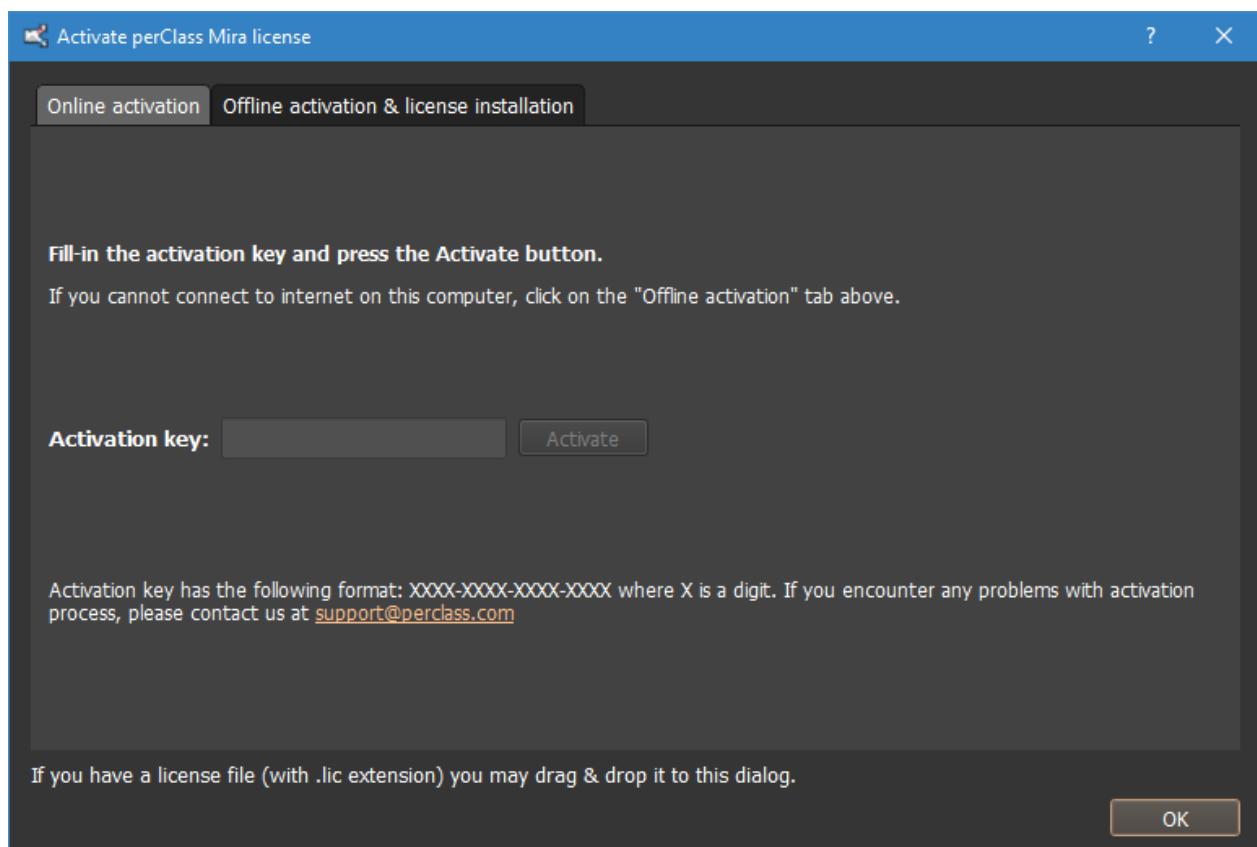


Finally, you may launch the installed software directory from the installation dialog:

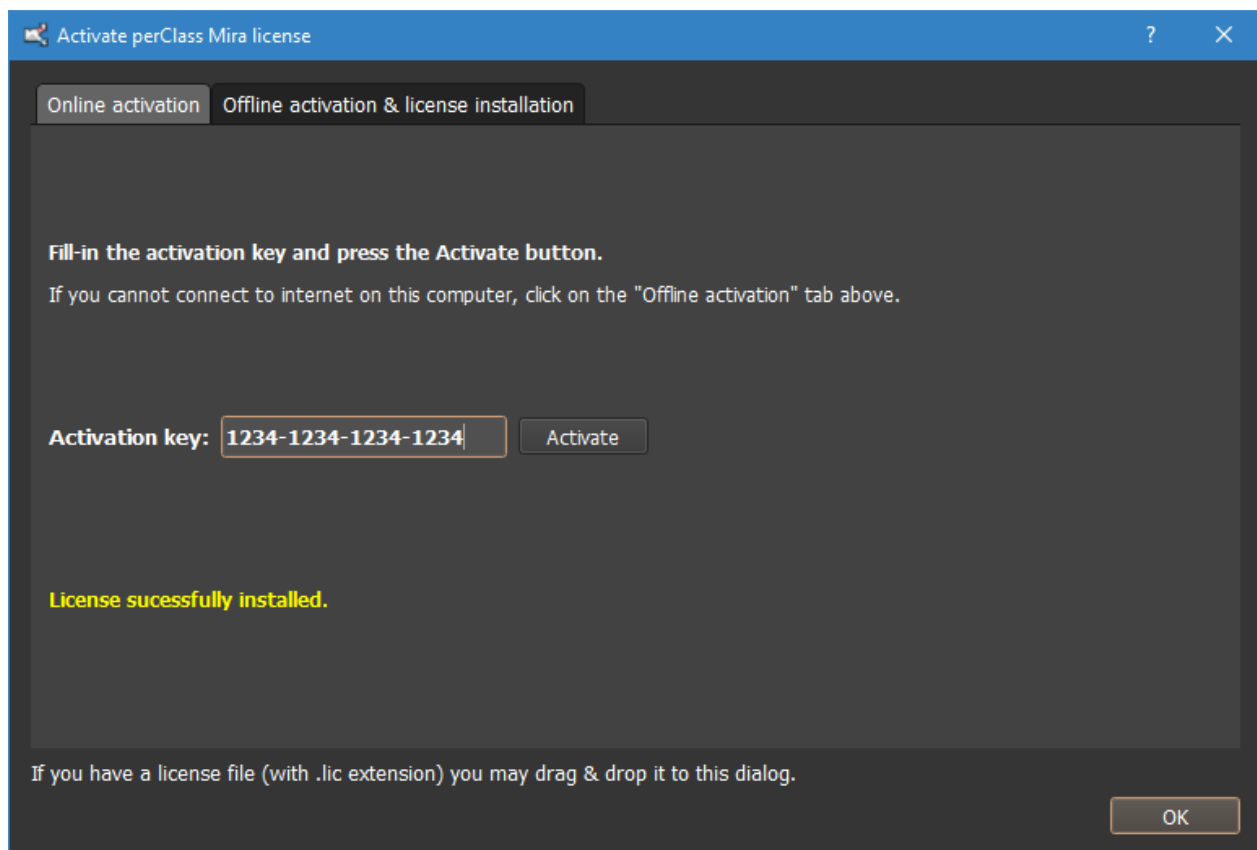


## Activation

On the first run, the *Activation* dialog appears:



Fill in the activation key and press *Activate*



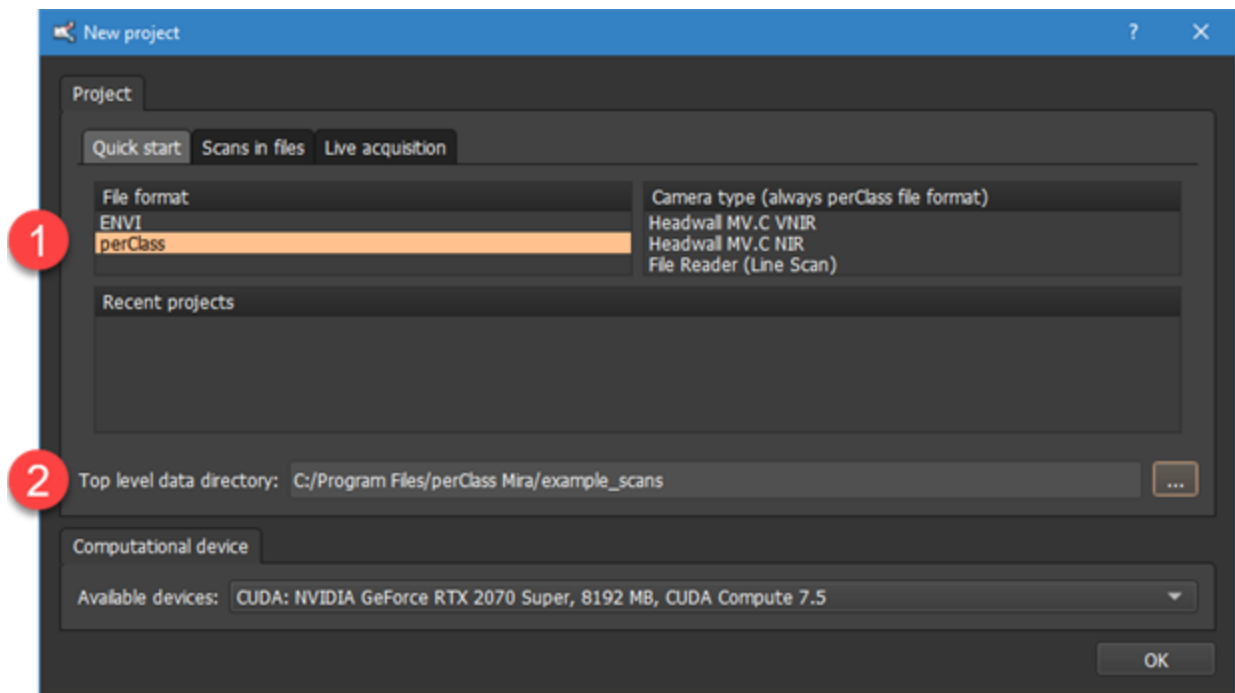
## Build classifier on existing scans

This tutorial demonstrates how to quickly build a classifier interpreting data in existing scans.

If your system contains a GPU, please start the software via `perClass_Mira_gpu` shortcut or `.exe` file. Otherwise, use the `perClass_Mira` shortcut that provides only CPU backend.

## Creating a project

Create a new project using *File / New Project* menu command.



The user has an option to either start from scans already stored in files or acquire data from a connected spectral camera.

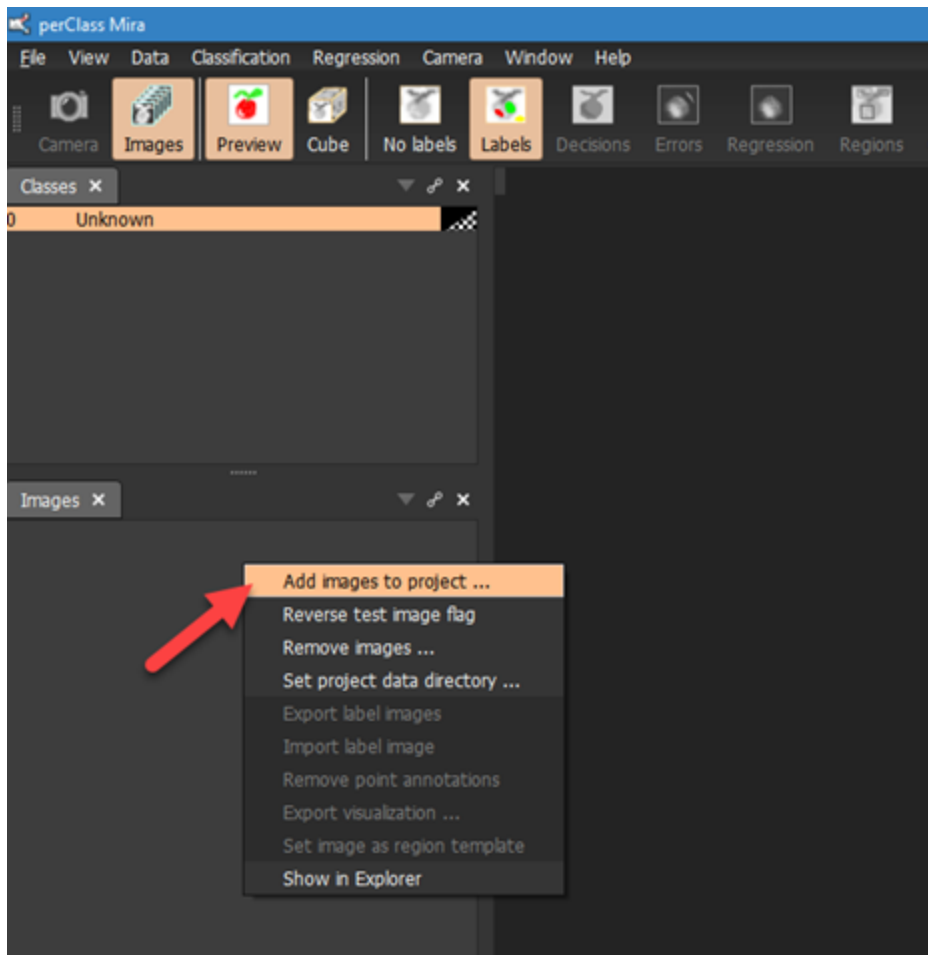
In this tutorial, we load an existing scan. Therefore, select the *perClass* project type.

The *top level data directory* field specifies where perClass Mira expects the scans. We will select the `example_scans` sub-directory in the installation directory, typically `C:/Program Files/perClass Mira/example_scans`.

The *Computational device* combo box allows you to specify the GPU, if available.

### Adding images

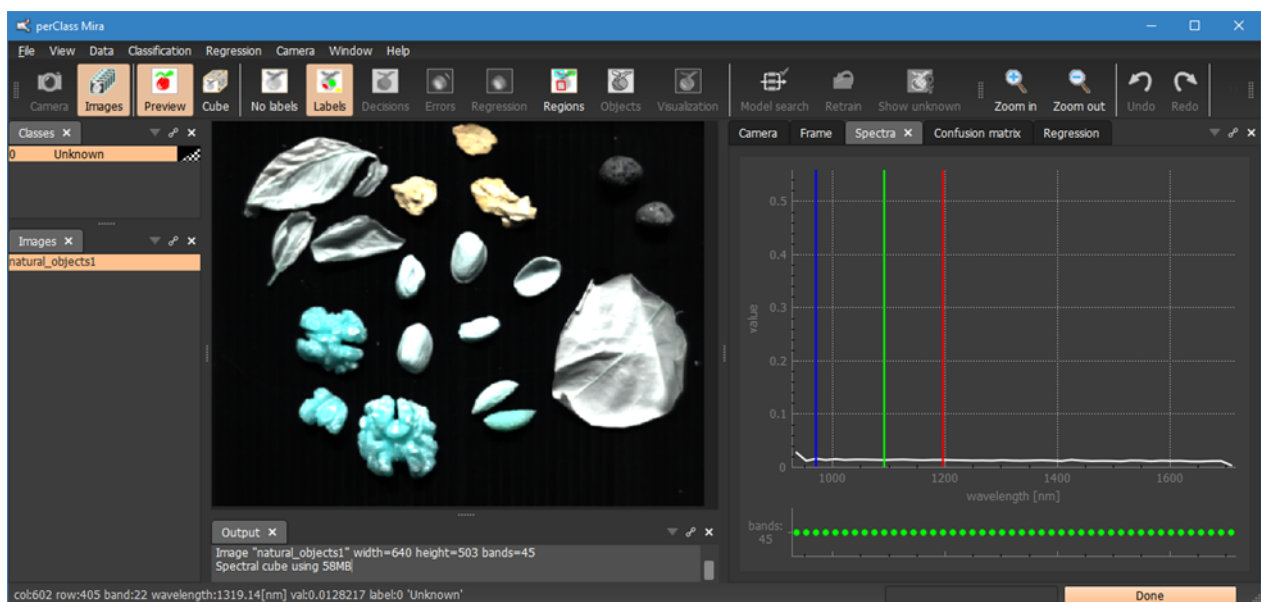
perClass Mira main window will open. In order to add images, right-click with your mouse in the *Images* list and select *Add images to project...* from the context menu.



A dialog box will appear where you can select one or more images. For the perClass project type, the ENVI .hdr files are listed corresponding to the ENVI cubes.

**TIP:** In order to select more than one image, hold Ctrl and click on the desired file names. If you wish to select a set of images, you may click on the first one, then hold Shift key and click on the last one of the desired group

In our example, we open the `natural_images1.hdr` file.



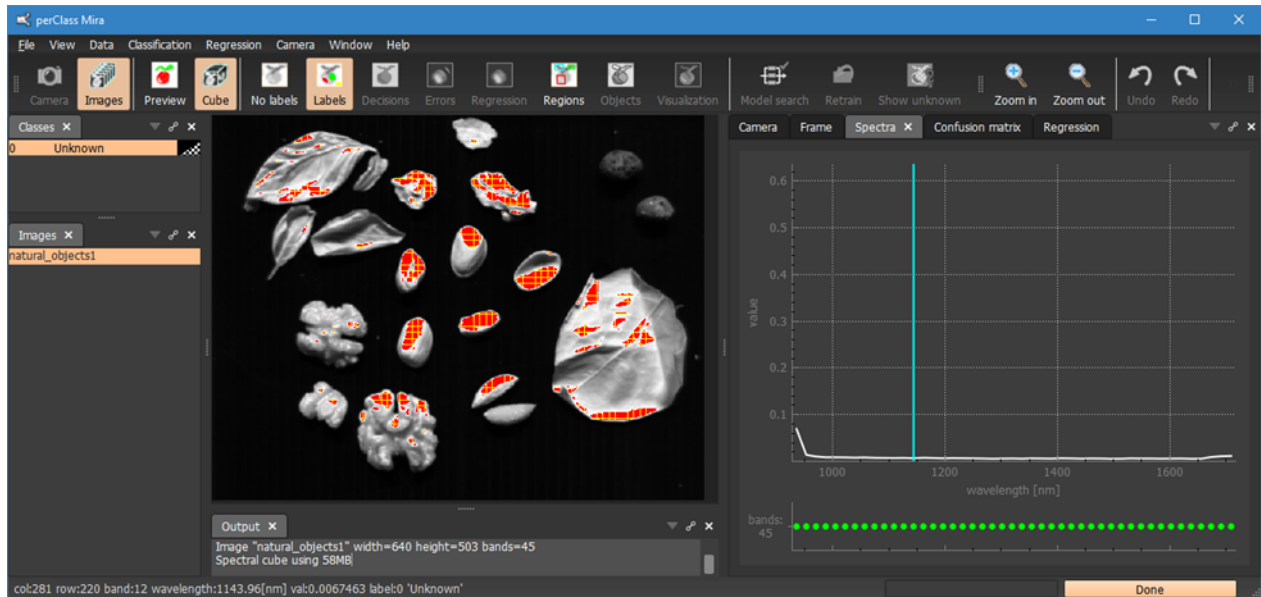
## Spectral cube visualization

If the spectral cube specifies default R,G,B bands, like in our example, the image will open in false-color preview mode. Otherwise, the single band mode is used.

You may drag the R,G,B lines in the spectral plot to adjust the false-color view. As you can see, the scan was acquired by a sensor operating in NIR spectral range (900-1700nm). Therefore, the colors are only visual aid and do not correspond to the color of the physical objects.

Moving the mouse over the image, you will see the spectrum at each point as a white line in the spectral plot. Details on pixel coordinates, wavelength and a value can be found in the status bar area.

You may switch to the single band mode using the *Cube* button on the toolbar.

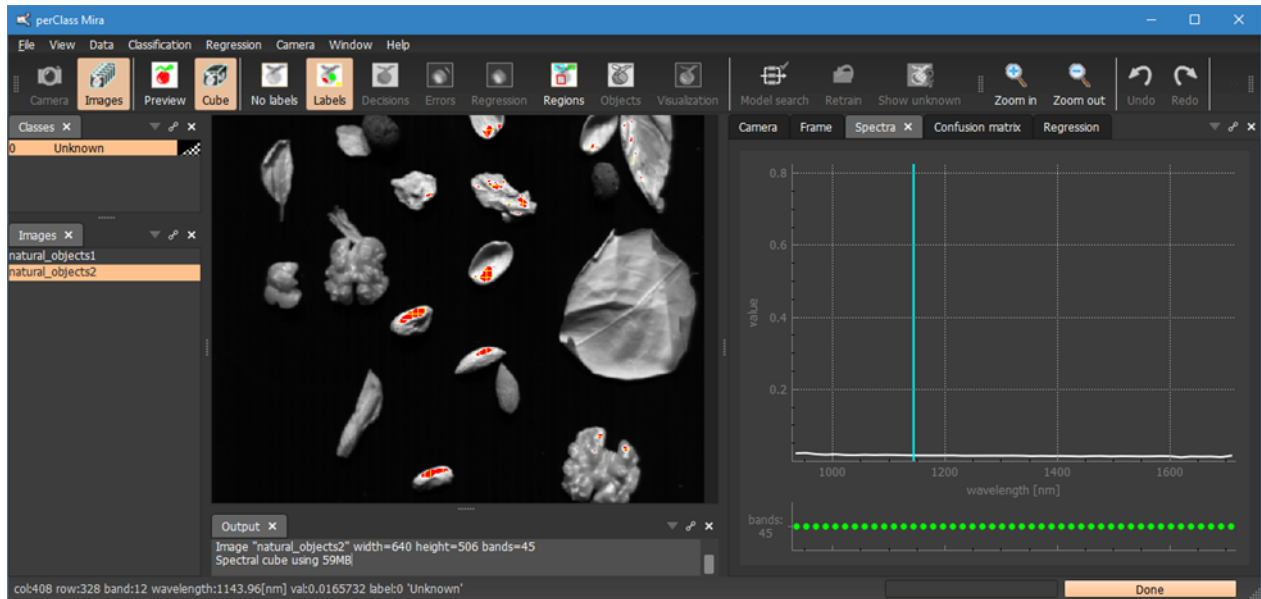


Dragging the blue line in the spectral plot, you may change the band. Note the band index and wavelength in the status bar.

Note the red pattern on some of the pixels. This is a visualization feature that highlights pixels with values higher than the current maximum bound of the spectral plot.

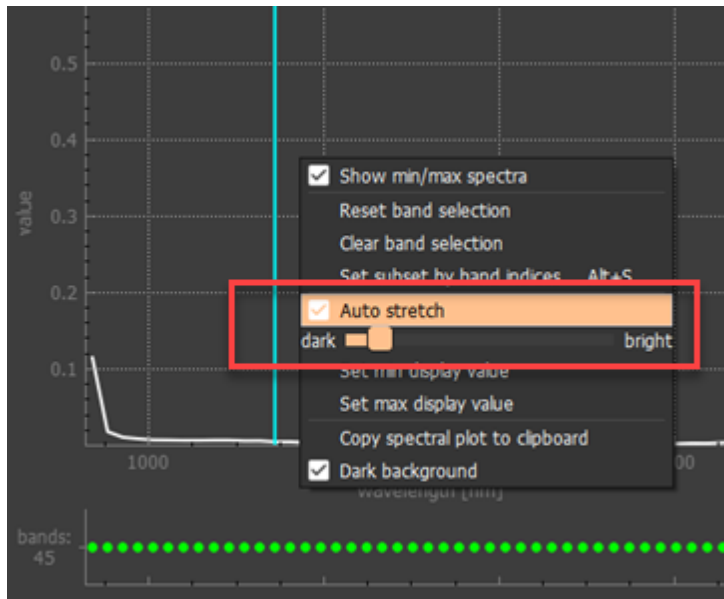
You may adjust the spectral plot range using the mouse wheel or by clicking and dragging in the spectral plot.

We will load the second image. Select again *Add images to project...* in the *Images* context menu and select *natural\_objects2.hdr*.



Note, that the spectral plot bounds do not change when moving between images. With the manual spectral bounds, you always see comparable view of the data.

We may also enable **auto stretch** of image brightness in the spectral plot context menu (via right mouse click).



The auto-stretch extends each image so that a fixed percentile of the image histogram is visualized. The percentile can be adjusted by the slider.

When moving between the images, note that the spectral plot bounds change depending on the image histogram. The auto-stretch mode assures you always see "something" irrespective of how bright or dark an image is. You leave the auto-stretch mode by either disabling it in the menu or manually adjusting the spectral plot bounds.

**TIP:** The red pattern for pixels above the top spectral plot bound can be changed into white color in *View / Show saturated data as* menu command

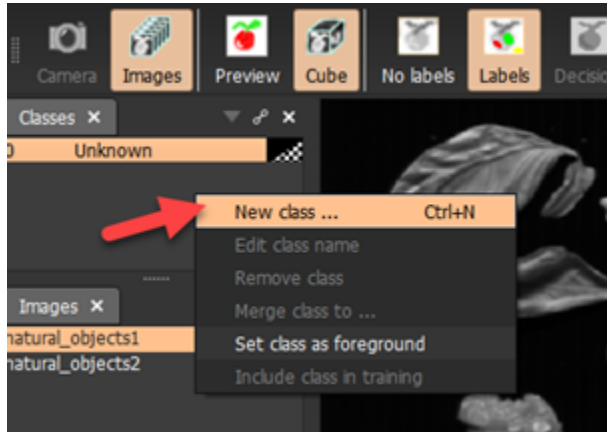
## Training a classifier

perClass Mira allows you easily define custom classification solutions. Classifier is an algorithm able to assign any pixel of your image to one of pre-defined classes.

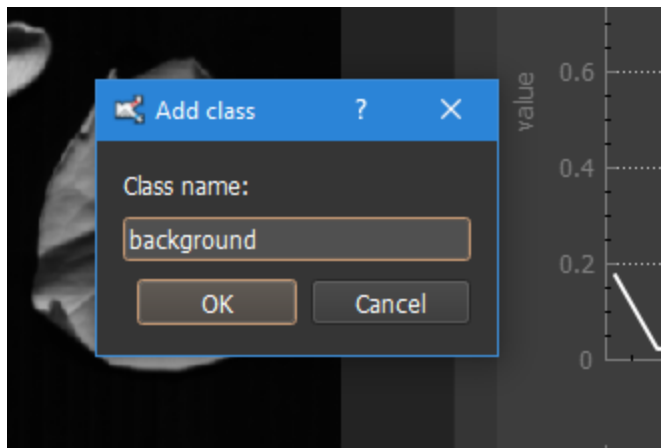
The process of building a classifier comprises three steps:

- Defining the classes of interest
- Labeling examples used for training
- Testing / validating that the classifier behaves as expected on unseen examples or images

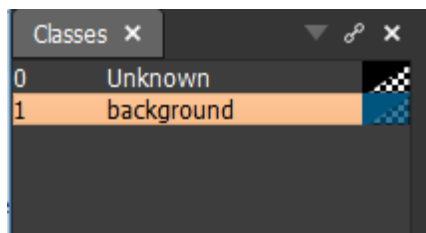
In order to define a class, right click in the Classes list and select *New class...* command:



A dialog will appear, where we can specify class name. We're interested to define the class called background:

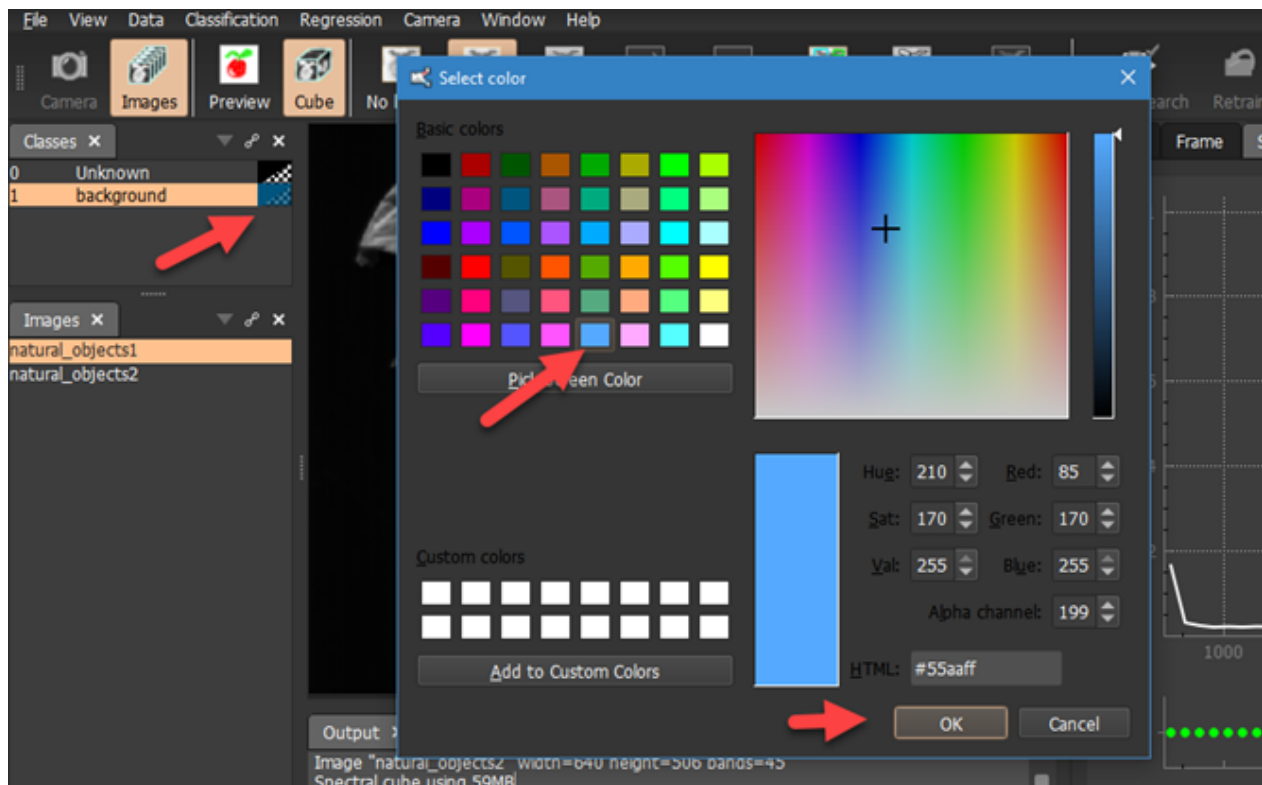


The new class will be added to the class list:

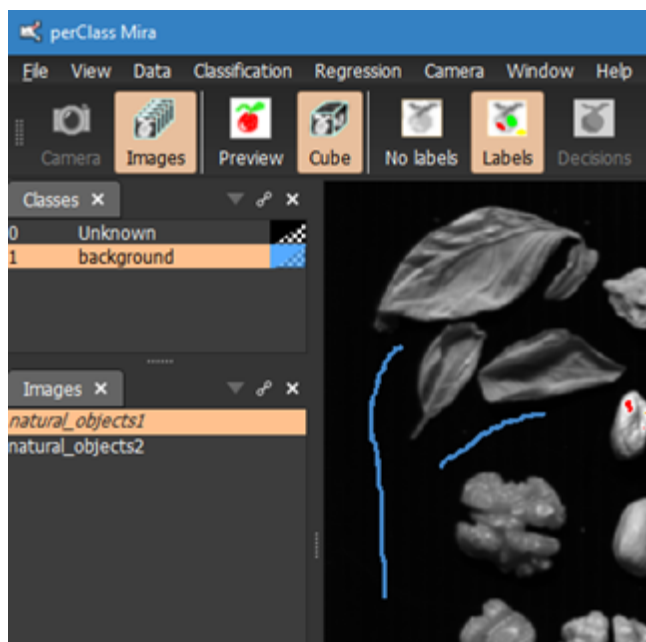


We may now label (paint) some pixels we consider background:

Let us change the color to lighter blue that is easier to see in our image by clicking on the color swatch next to the class name.



In order to label pixels using the selected class, hold left mouse button and move over the image.



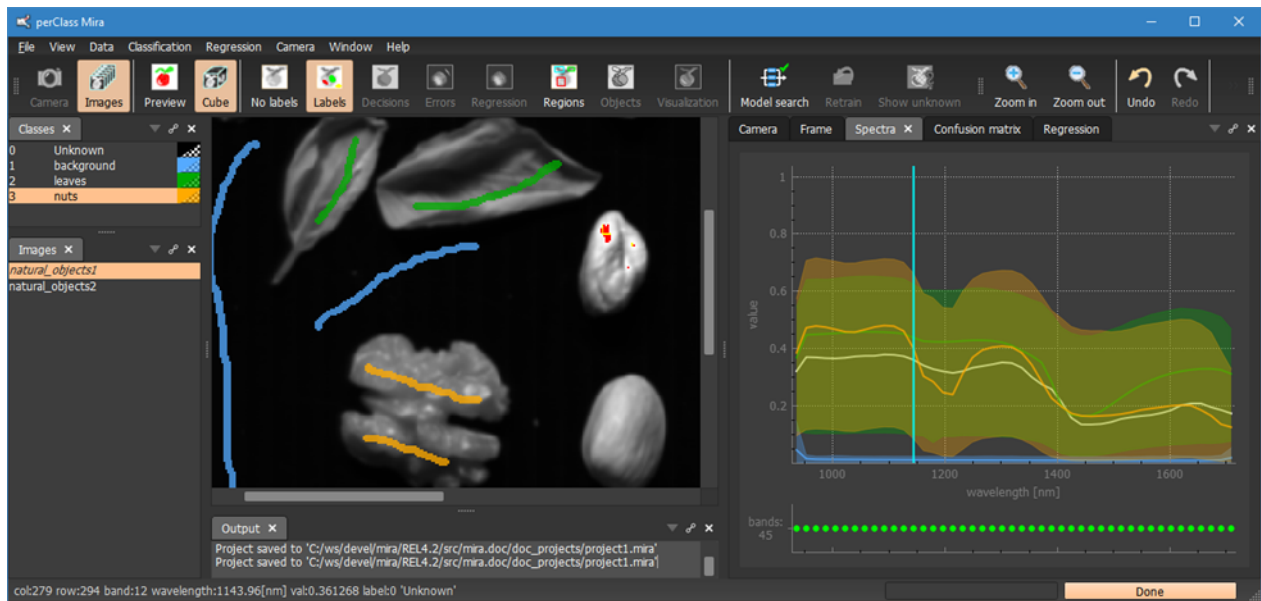
**To zoom** the view in or out, you may hold Ctrl key and use mouse wheel. This zooms aiming at the mouse pointer. Alternatively, use the Zoom buttons on the toolbar

**To delete labels**, hold the *Shift* key and paint with a mouse.

You may also adjust the brush size using the *Brush* button on the toolbar.

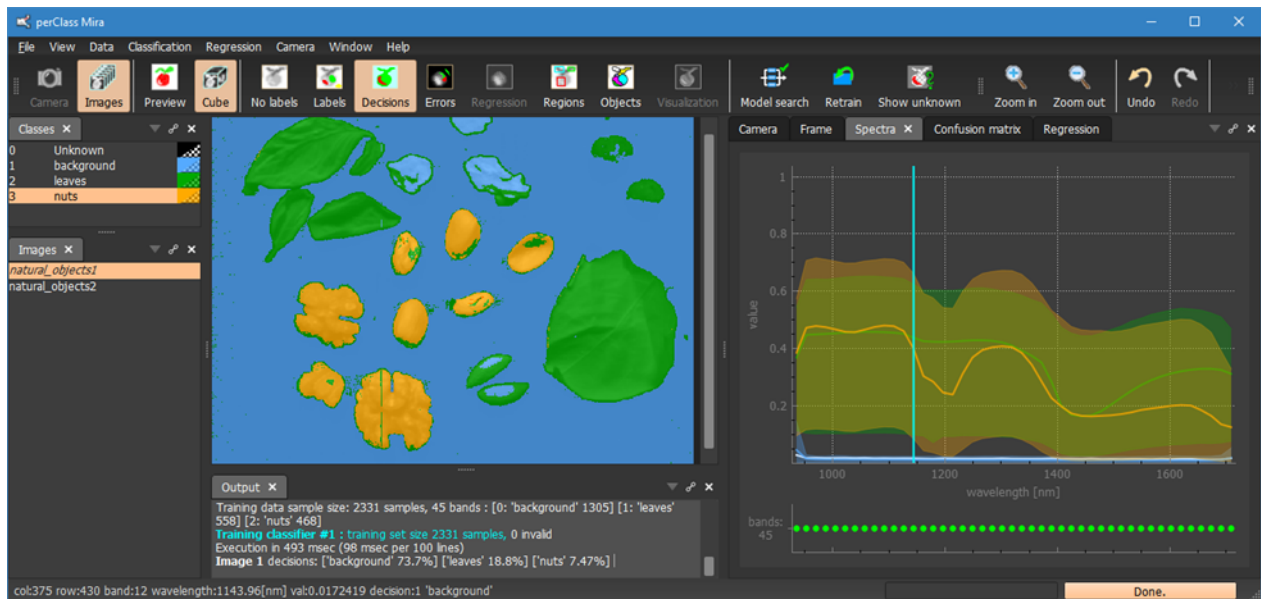
We define also the second class called leaves and the third one called nuts:





Note, that mean spectrum of each class and its min/max range are indicated in the spectral plot.

We can now click the *Model search* toolbar button to build a classifier. The software will use the labeled pixels to optimize classification model. It will then apply the trained model to the entire image switching to the Decisions view. To see the entire image, we zoomed out.

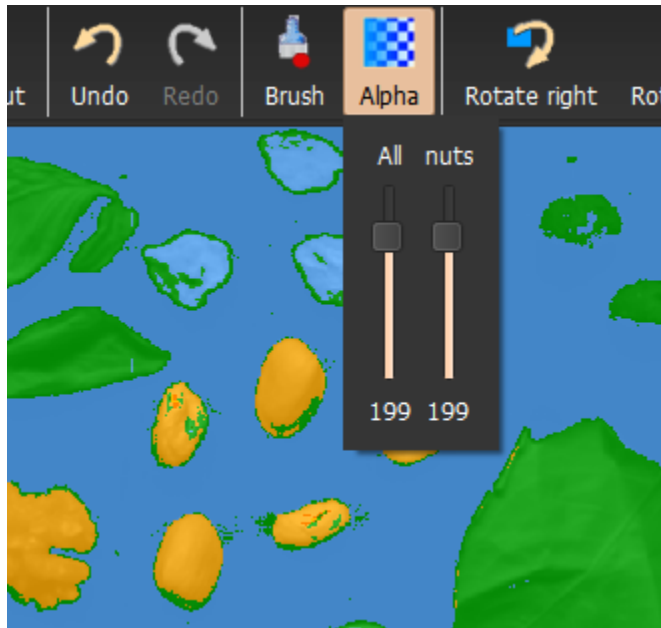


## Switching between labels and decisions

In the decision view, each image pixel is assigned to one of the user-defined classes.

Note, that we view a visualization comprised of the decision layer as a transparent overlay over the data layer (RGB or single band).

We may change the transparency using the *Alpha* toolbar button. When you click the Alpha button, two sliders will appear.



The left "All" slider controls all classes. Adjusting it, we make all classifier decisions more transparent revealing the data layer (single band or the RGB Preview depending on the selection).

The right slider is class-specific. It controls the class, selected in the class list. It allows us to fine-tune visualization based on our use case. For example, we may wish to keep only specific defect strongly visible where all other classifier decisions are fully transparent.

In order to label more examples, click on the *Labels* button.

**TIP:** You may use Spacebar key to switch to the previous layer (here we would switch from Decisions to Labels)

## Improving the classifier

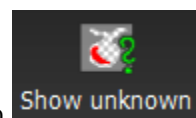
There are several ways we may improve the classification results:

- Improve the labeling and retrain the model
- Add / remove classes
- Remove noisy or uninformative bands

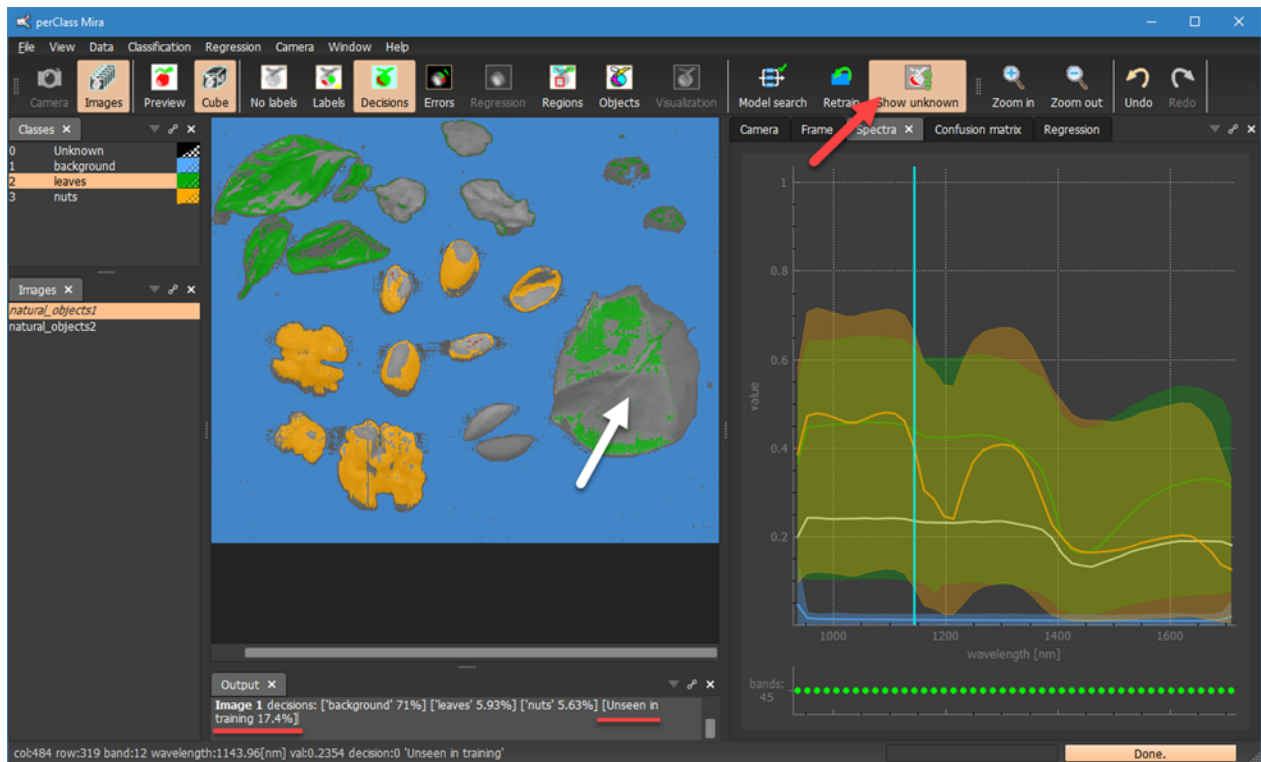
Subjectively, we may judge classifier performance visually by applying it to new images unseen in training. In order to assess objective classification performance, we may estimate error using confusion matrix tool on test images, unseen in training.

## Improving the labeling

perClass Mira provides an active learning tool that helps us to understand what examples the model did not see in training.

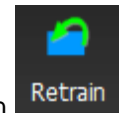


You may enable it using the *Show unknown* toolbar button or by pressing **u** (unknown)

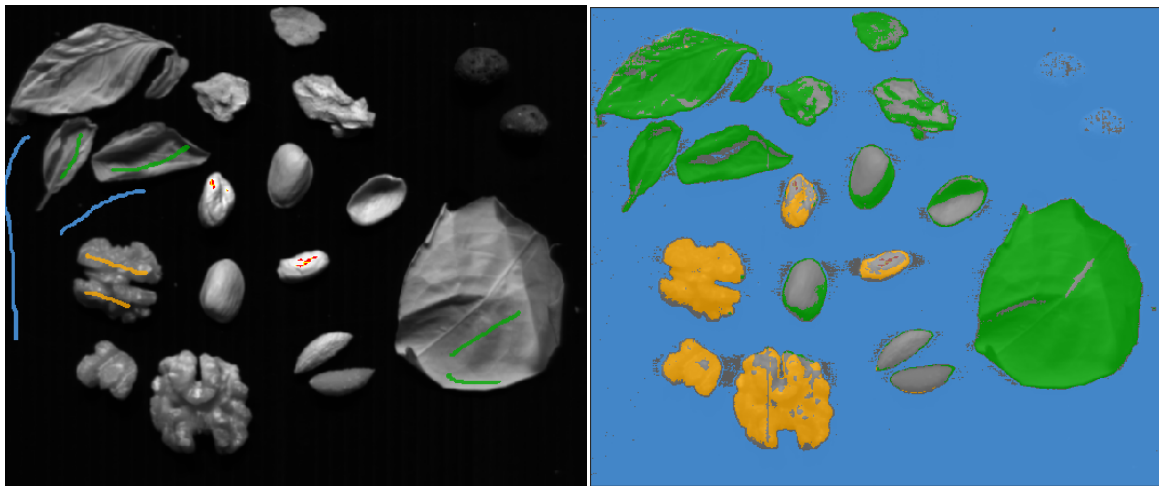


There is an extra decision "Unseen in training" with high transparency. The extra decision highlights the pixels that the classification model considers very different from anything labeled.

For example, the leaf on the right side is largely rejected being slightly different than the two leaves we labeled.



We may add extra labeling and retrain the model using the *Retrain* toolbar button



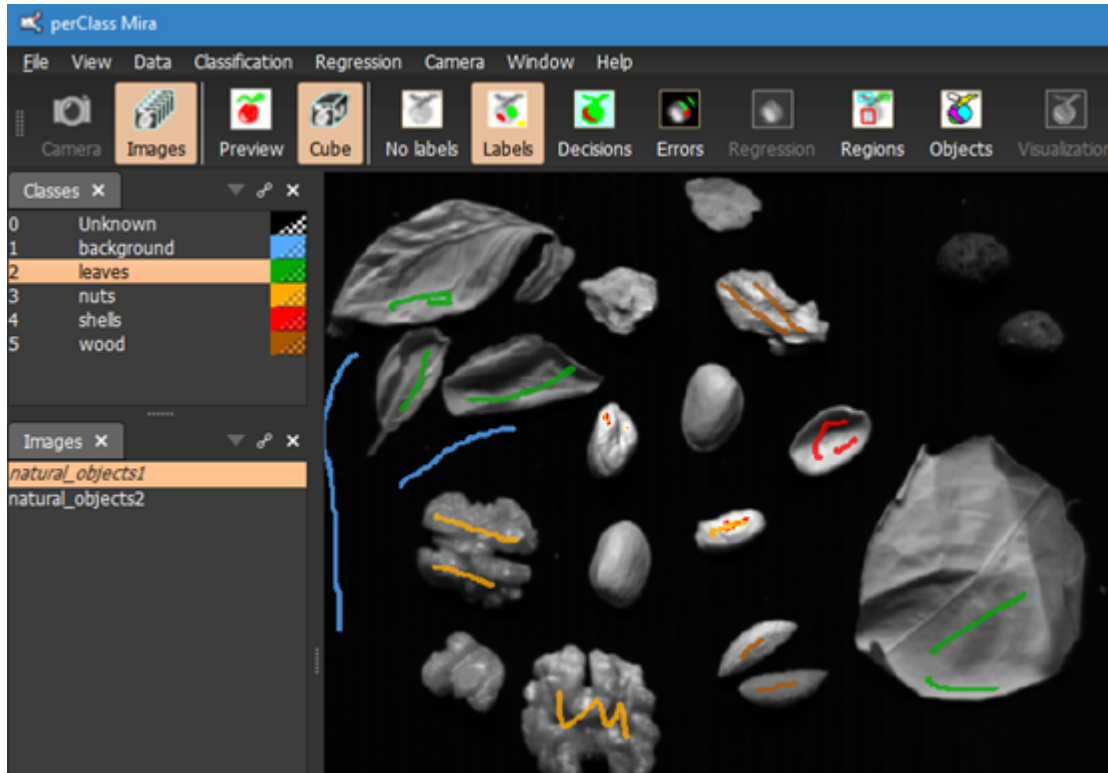
The *Show unknown* tool allows us to label in the areas needed and, thereby, building representative training sets.

**TIP:** It is generally better in perClass Mira to define less but accurate and representative labels. Use small brush with at least 2 pixels (to allow assign-stroke-to-class)

## Adding new classes

We may add or remove classes anytime. In our example, we labeled only background, leaves and nuts so far. However, there are other objects of other materials present, such as nut shells, olive kernels, wood or stones.

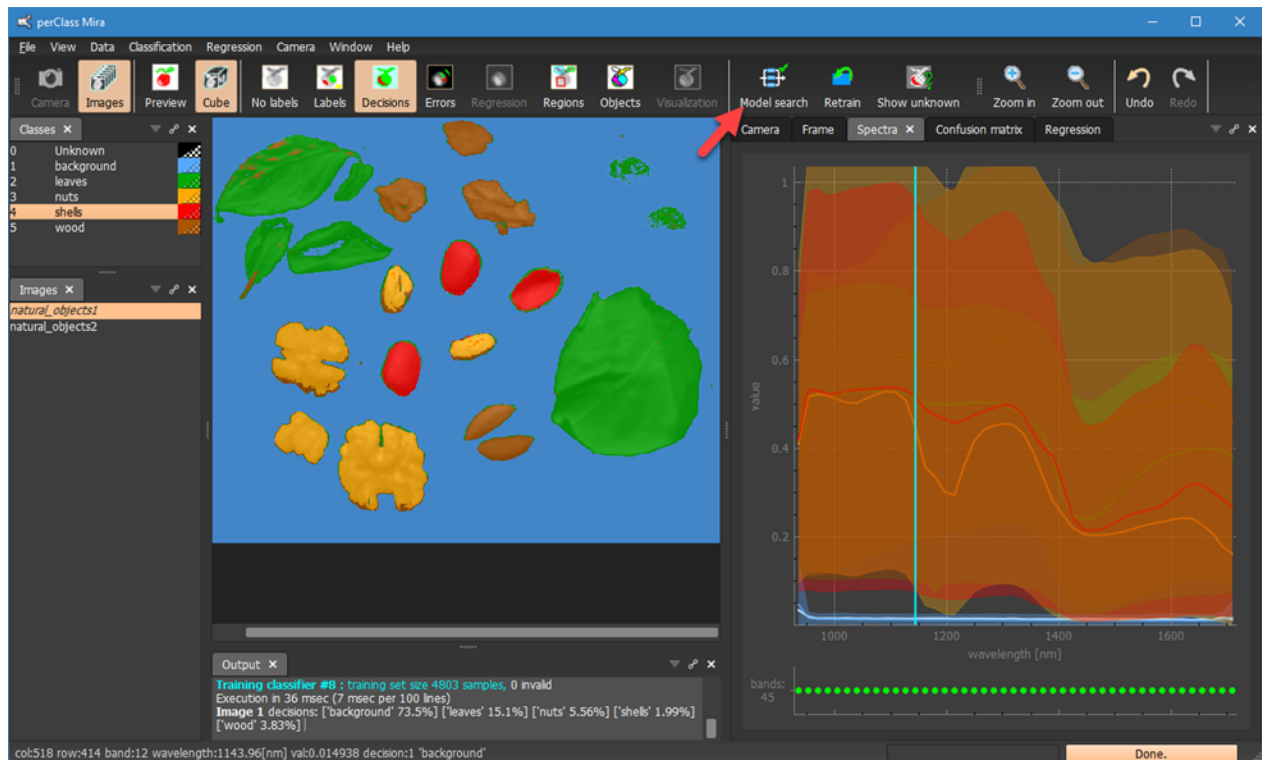
We add several classes to the project:



We will then use *Model search* to find a new model. The difference between *Model search* and *Retrain* tools is as follows:

- The *Model search* performs full search for the best model
- The *Retrain* tool uses the existing model definition and retrains it using the current labels

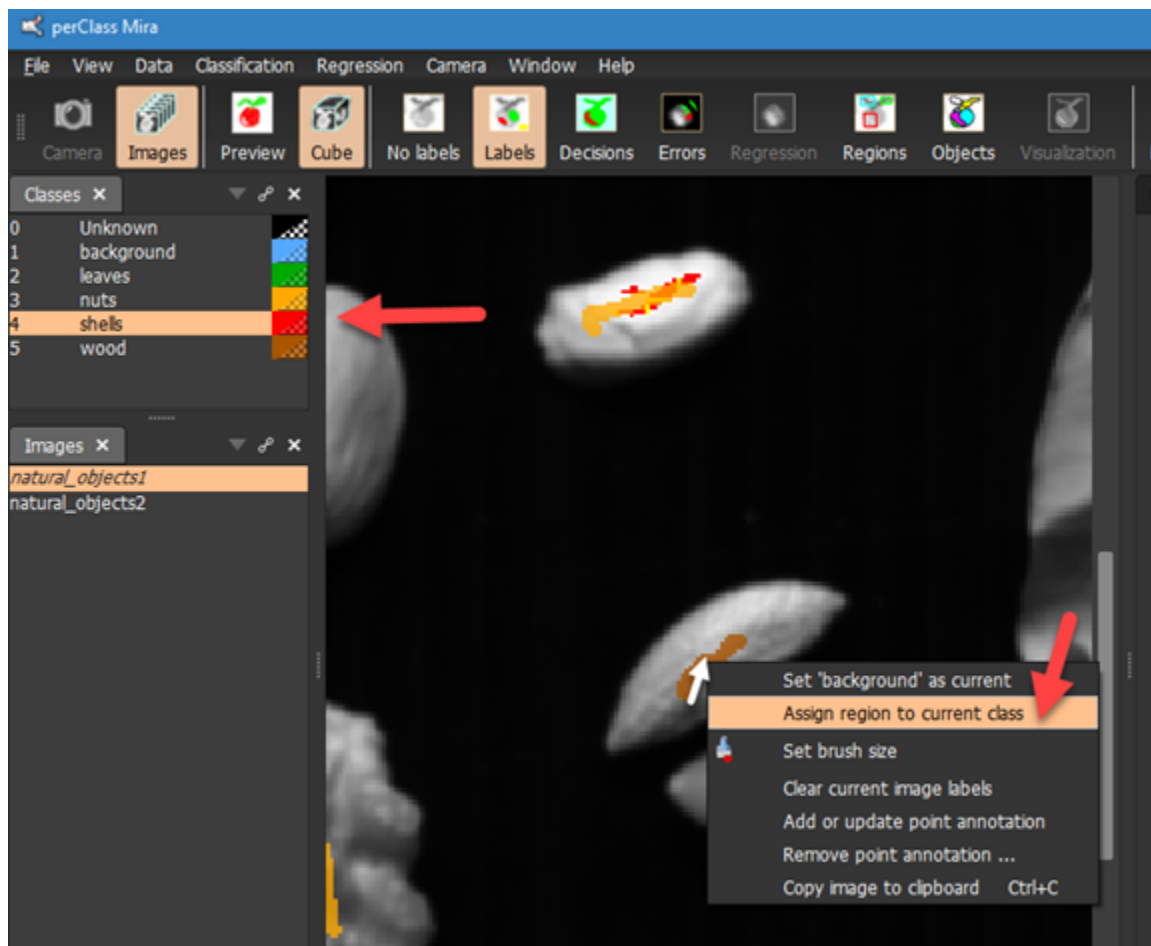
If we only perform slight update of the labeling, for example, around the edges or adding more representative examples of existing materials, Retrain is sufficient. The model search is useful when we add entirely new classes or label quite different examples of the existing classes.



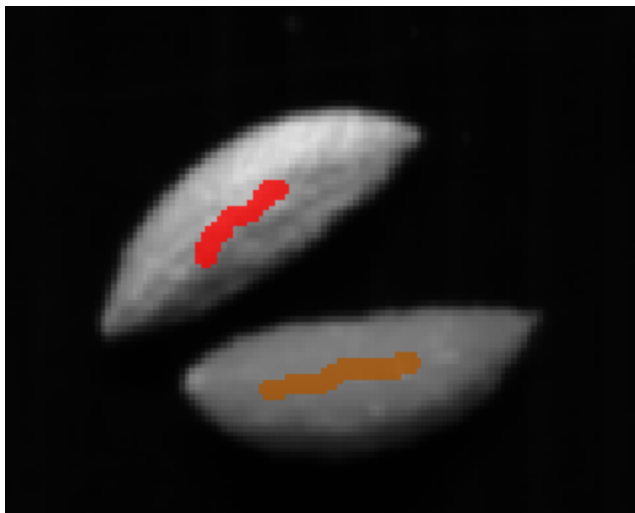
We can see that we have labeled olive kernels as "wood". We may wish to have them labeled as "shell" instead. In perClass Mira, we have a full control on the labeling. We can easily assign individual labeling strokes to different classes.

Switch to the *Labels* mode (via the toolbar button or by pressing small l key (l as in Labels)).

In the *Classes* list, select the class you wish to assign the stroke to (Shells). Then, right-click with the mouse over the brown label stroke on top of the olive kernel.  
In the context menu, select *Assign region to current class*.



The label stroke will be assigned to the *shells* class and will become red:



We also assign the second label stroke on top of the other olive kernel to *shells* and re-run the *Model search*.

The olive kernels are now part of the red *shells* class. Note, that in perClass Mira, the user may define classes that span multiple materials or represent generic high-level concepts (defect, background, product etc.)



## Acquire data and interpret

This tutorial illustrates how to

- acquire data using perClass Stage
- build classification model
- apply the model on the live data stream

This chapter uses [perClass Stage](#) scanning kit and Headwall MV.C VNIR camera connecting over USB3. [Follow these steps](#) for specific camera installation instructions.

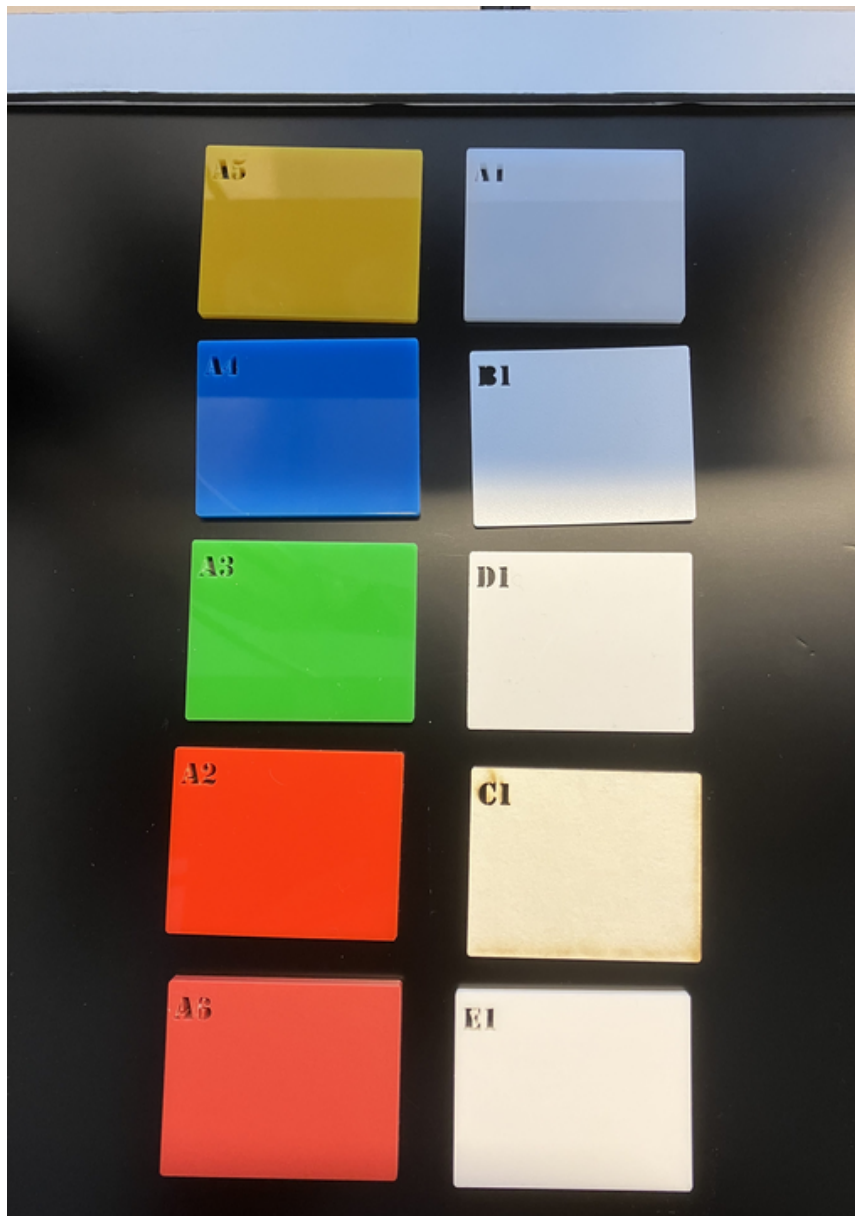




We will leverage the default plastic samples included with each perClass Stage. These are tiles of different technical plastics with etched labels such as **A1-A6** and **B1-E1**. The letter corresponds to a material and the

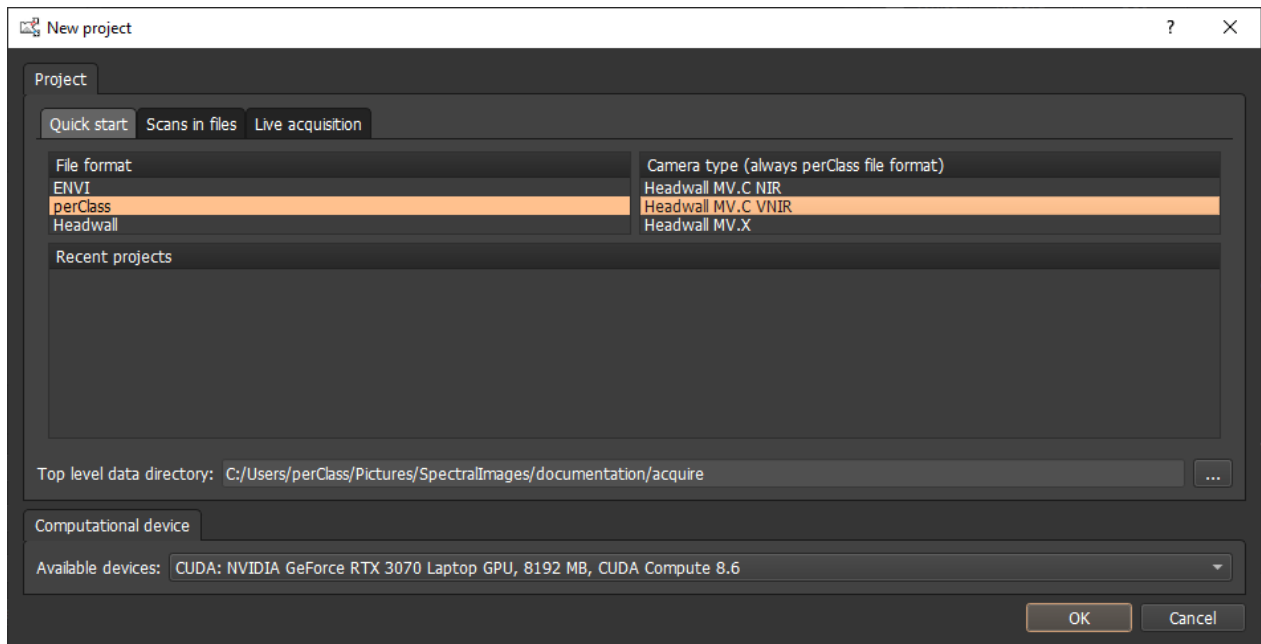


number to its variant. For example, **A1** and **A3** represent the same material in different color combinations while **A1** and **B1** two different white plastics.



### Creating a project for acquisition

When starting an acquisition project, we need to select the desired camera type on the right side of the *New project* dialog. Note, that any image acquisition in perClass Mira leverages "perClass" project type which gets automatically selected when clicking on the camera type. In this tutorial, we select the Headwall MV.C VNIR camera.

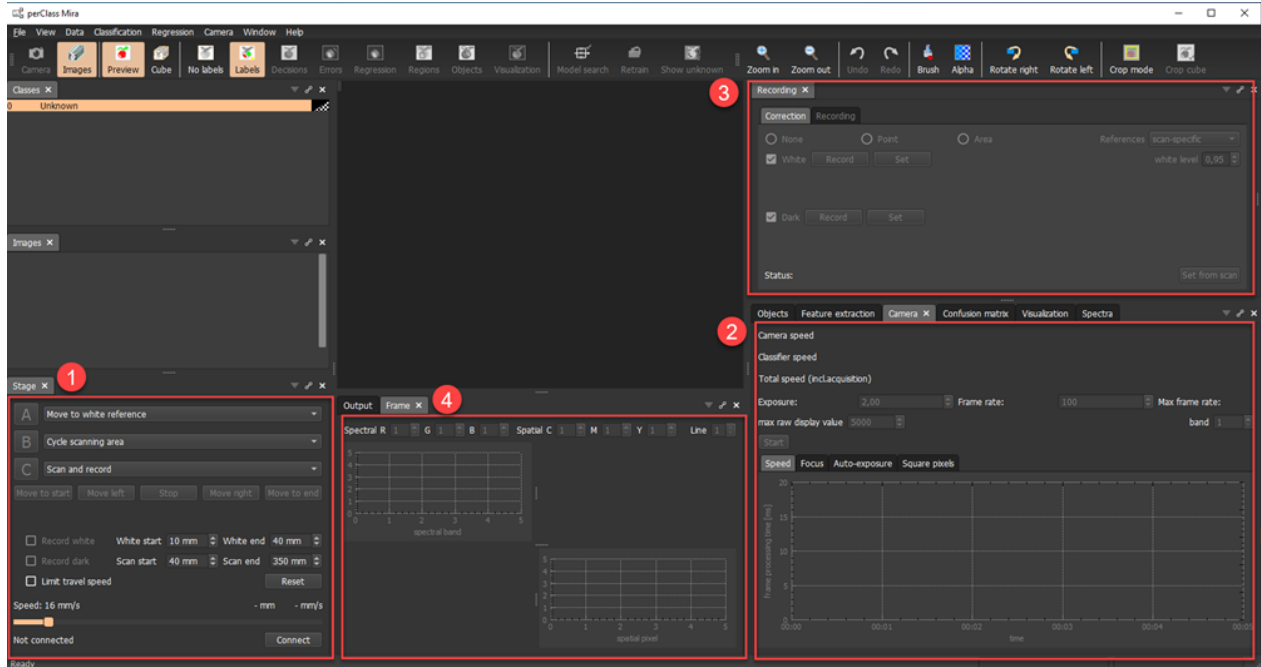


**TIP:** The Quick start tab only lists camera and project types we have flagged as favorite. You may select from all supported acquisition devices in *Live acquisition* tab and project types in *Scans in files* tab.

## Connecting to the stage

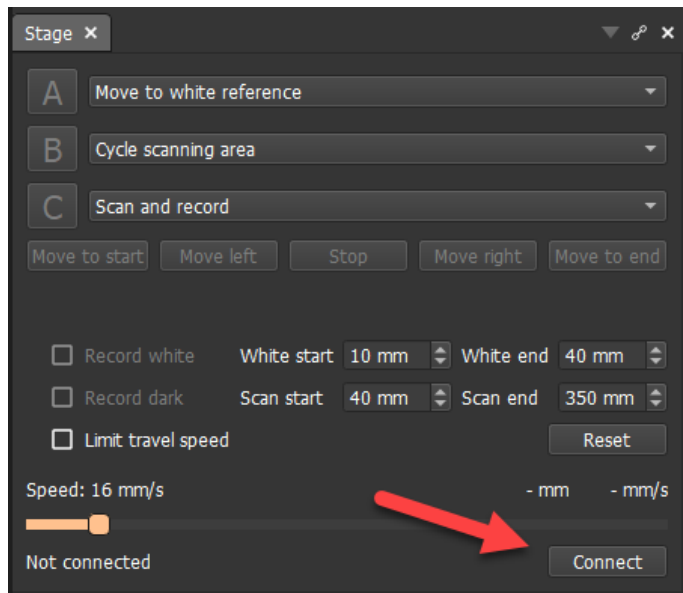
In order to scan, we need to connect to perClass Stage and to the camera.

Recommended screen organization for scanning is as follows:

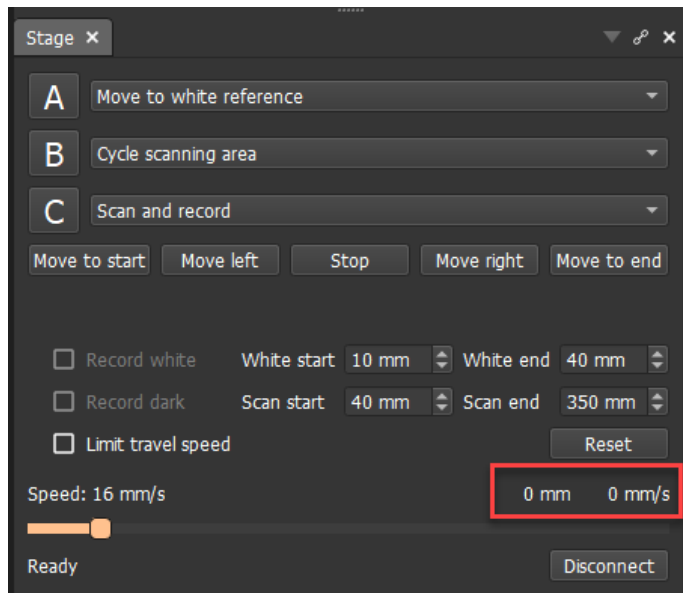


- the **Stage** panel in the bottom-left provides all stage controls.
- the **Camera** panel shows controls of the acquisition
- the **Recording** panel in top-right allows us to define references and scan recording settings
- The **Frame** panel in the center shows the raw signal from the camera

We need to connect to the perClass Stage in order to control it by pressing the *Connect* button in *Stage* panel:

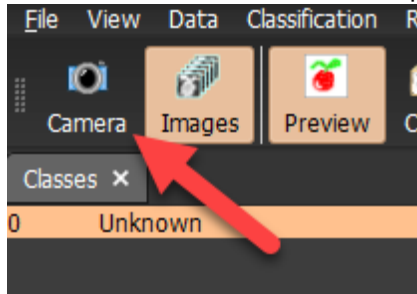


The stage performs a homing run. The white reference block will then be under the camera. This represents the "position 0" of the stage.

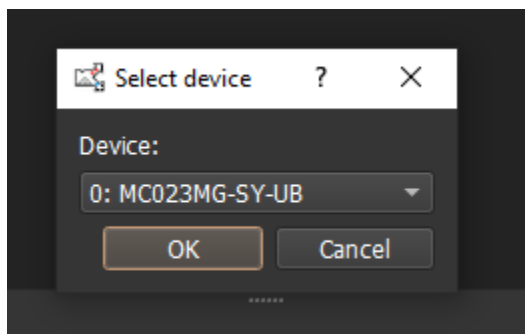


## Connecting to the camera

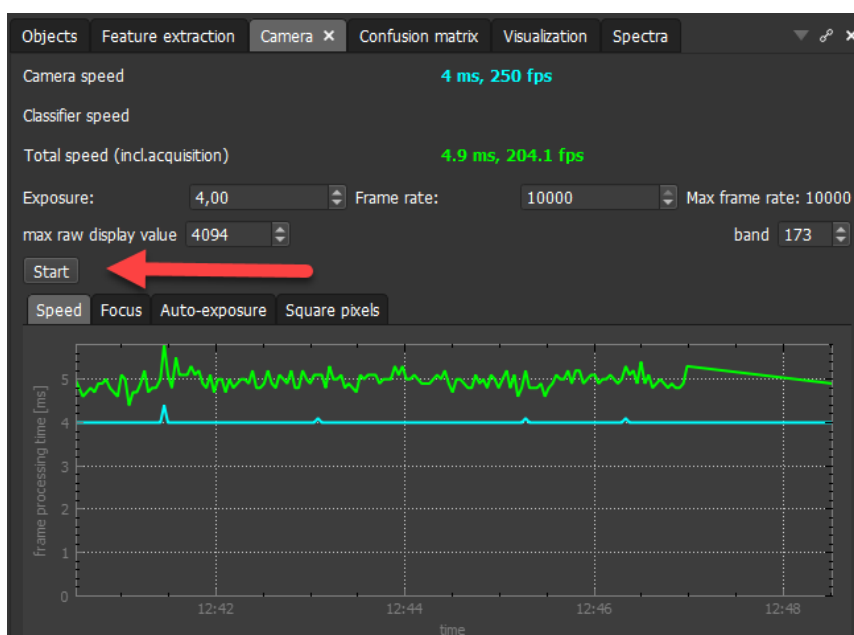
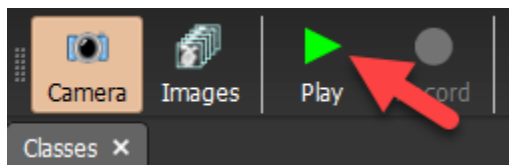
In order to connect to the camera, press the *Camera* button on the left side of the toolbar:



When the camera is initialized for the first time, we are offered a dialog to select the device:



Once connected, we may start the acquisition either using the *Play* toolbar button or via *Start* button in the *Camera* panel:



The *Camera* panel then shows the speed of acquisition, namely the camera speed and the total speed

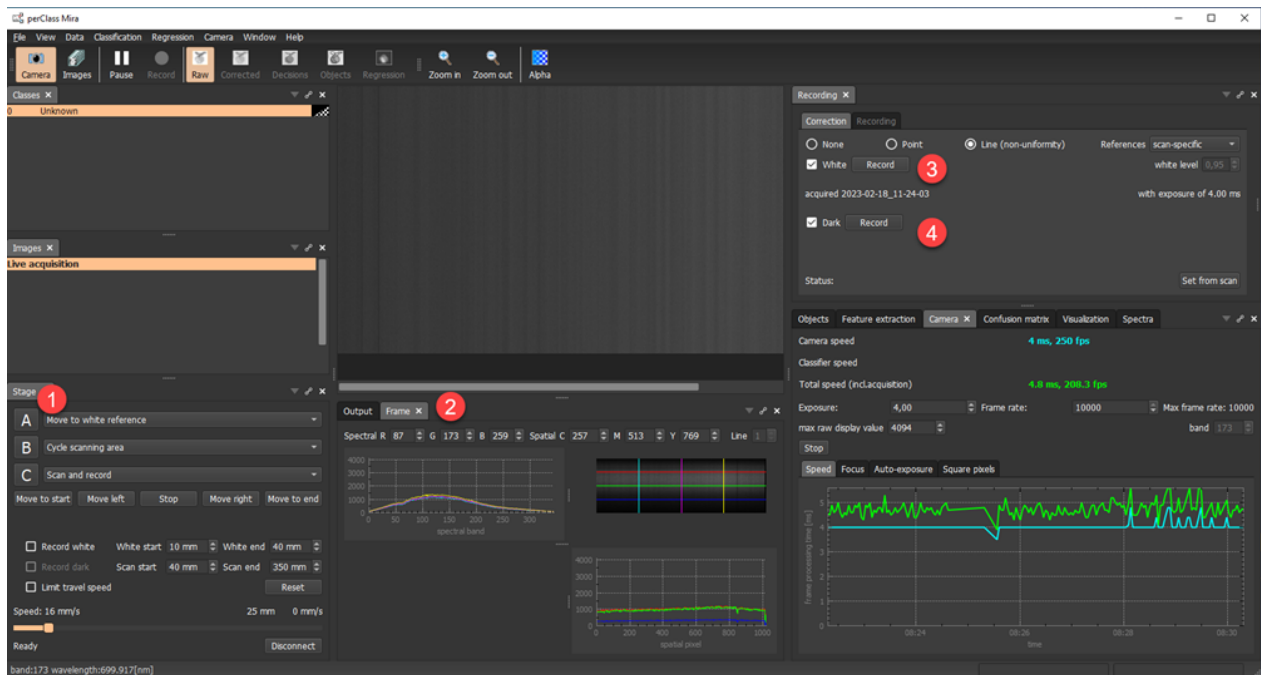
including also user interface updates and all other activities.

The *Camera* panel can be used to adjust acquisition parameters such as exposure (integration time) and frame rate. In this example, we keep the default settings.

## Recording references

In order to interpret spectral images, it is highly recommended to correct the raw data from the sensor into reflectance. This operation makes the data robust to illumination changes and thereby the interpretation models more generally applicable.

Reflectance correction is based on two reference scans, namely the "white" reference corresponding to the highest reflection we're considering and the "dark" reference defined by the noise of the imaging sensor.



1

We will first move the scanning table to the white reference bar.

In the *Stage* panel, we have selected the *Move to white reference* command in the combo box next to the **A** button. Therefore, we may execute this command by pressing the **A** button in the *Stage* panel or the physical **A** button of the stage



**TIP:** You may define the most useful commands for your scanning work flow in the A,B,C command combo

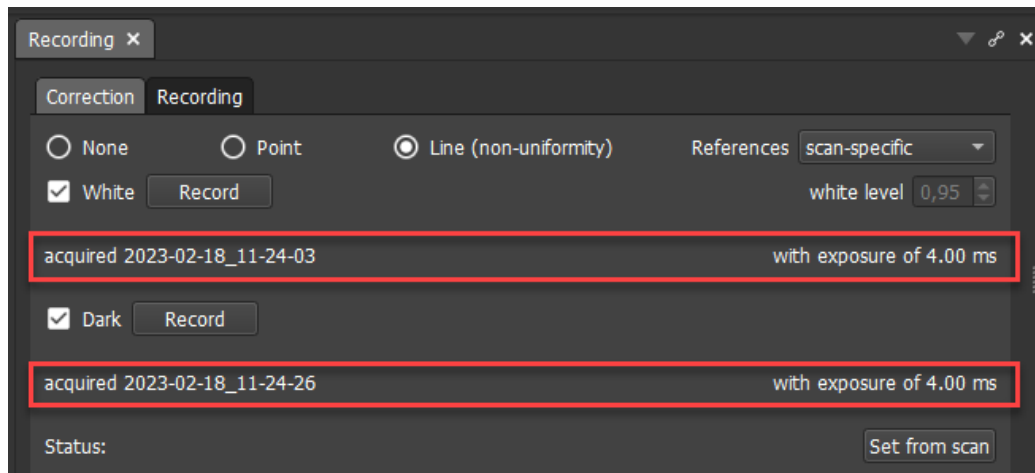
boxes. The choices are stored in mira.ini file and are available for future software sessions.

We can see actual raw data from the image sensor in the *Frame* panel <sup>2</sup> We need to make sure that the entire field of view is occupied with the white reference.

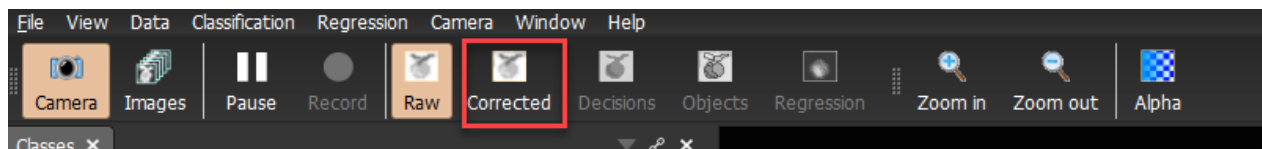
We can define the white reference using the respective button <sup>3</sup> in the *Correction* tab of the *Recording* panel.

In order to record the dark reference, we need to close the optical path of MV.C VNIR camera manually and press the *Record* button <sup>4</sup>

The time stamp and exposure used when recording the references is displayed in the *Recording* panel.



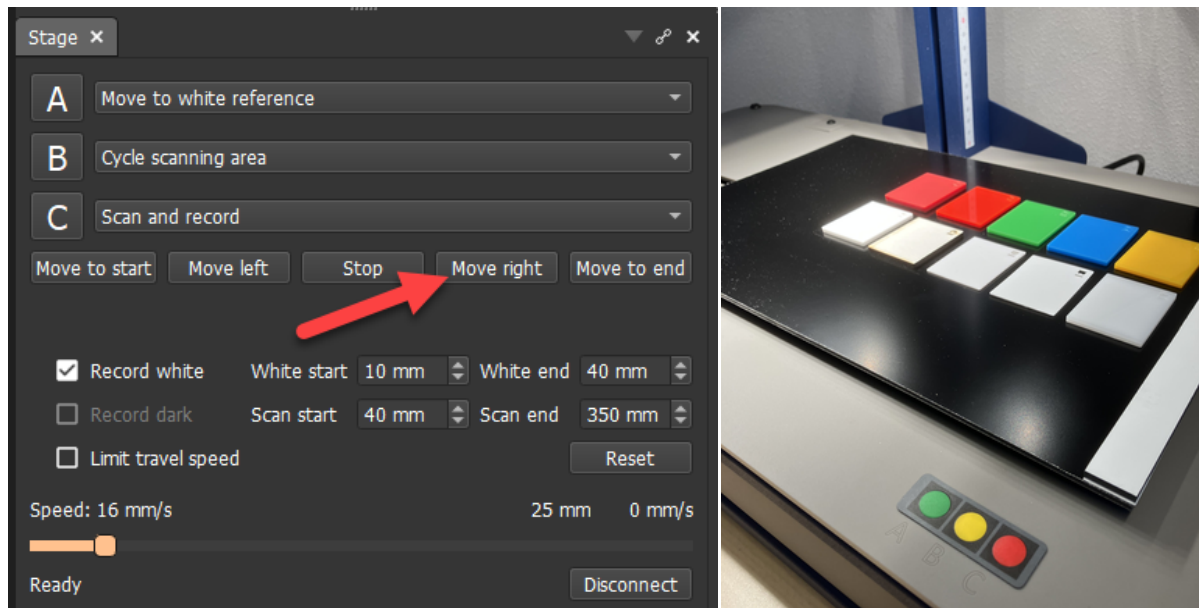
Once references are acquired, we may switch to the *Corrected* stream visualization in *Camera* mode.



**TIP:** If you change camera exposure, you need to retake the references. Difference in acquisition and reference exposure is indicated by red color of exposure value.

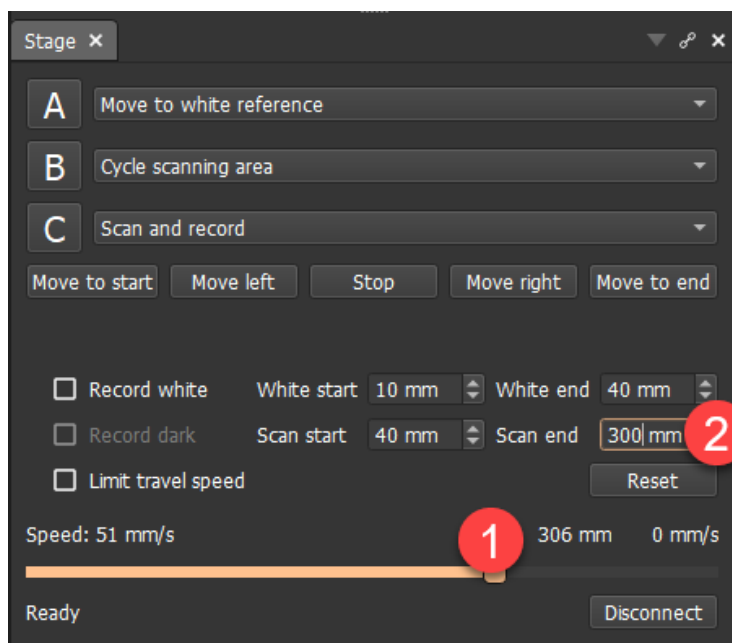
## Defining a scan area

In order to record a scan, we define what area of the scanning table we wish to acquire. We start the camera (*Play* toolbar button in *Camera* mode) and move the stage by **pressing and holding** the *Move right* button



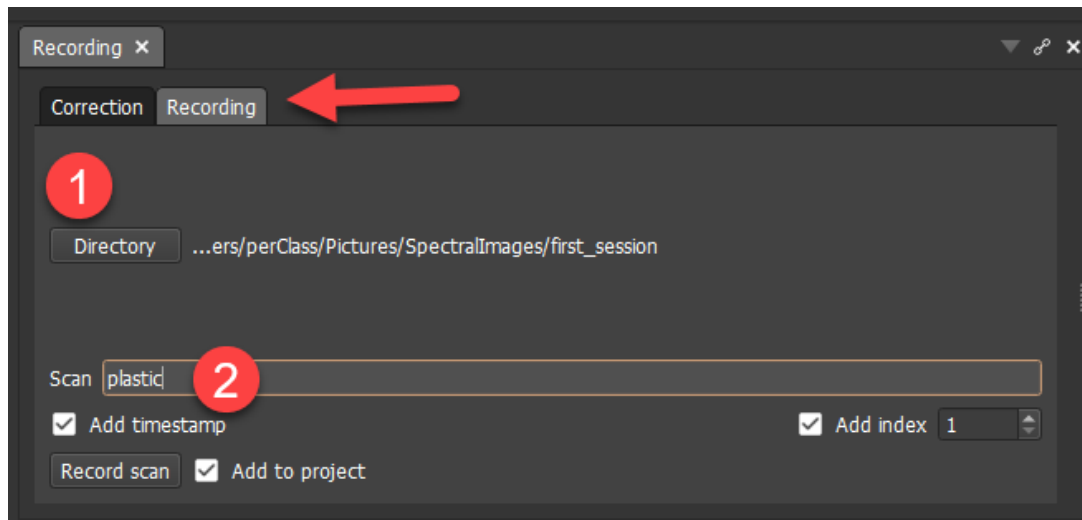
In our setup, we move the stage until we start receiving the data from the table after all our plastic samples.

We can then read the stage position <sup>1</sup> and fill the desired value into the *Scan end* edit field <sup>2</sup>



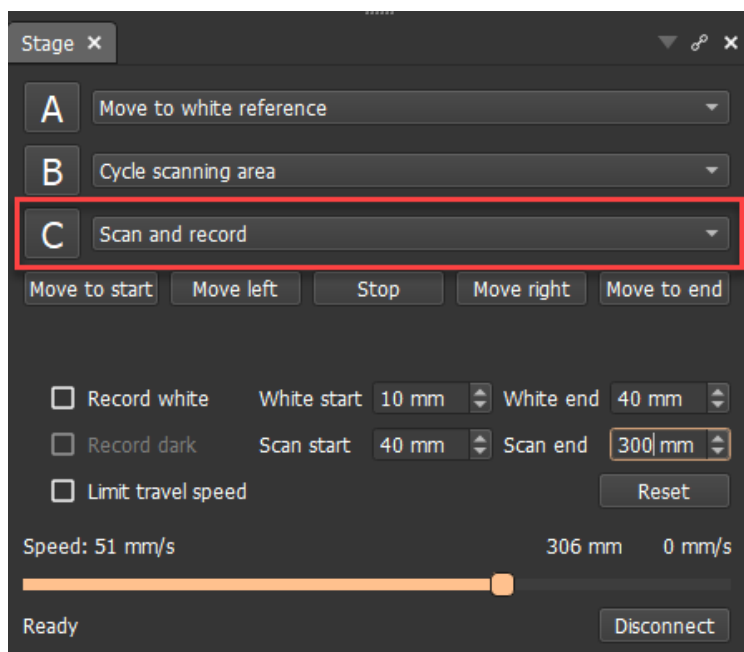
## Recording a scan

In order to record a scan, we can switch to the *Recording* tab next to *Correction*:

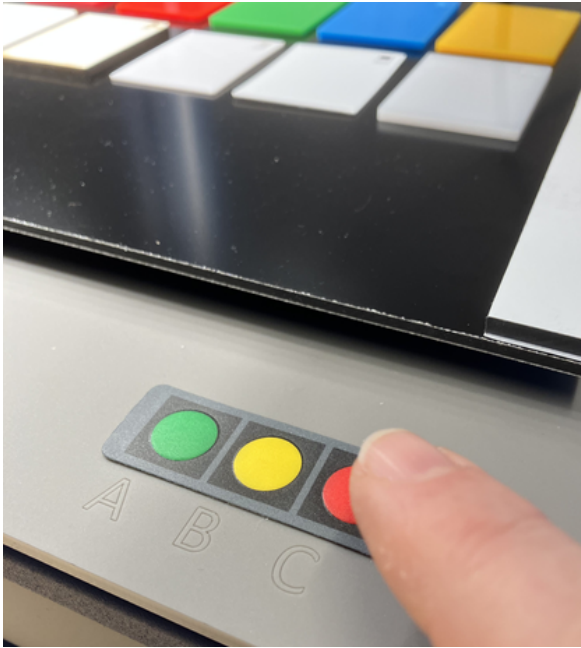


The *Recording* tab allows us to specify the directory where our scans will be saved <sup>1</sup> and the scan name <sup>2</sup>.

**Once the scan name is defined**, we can record our first scan. In our example, we have the button **C** set to *Scan and record* command. Therefore, we may press this button in *Stage* panel or press in the physical **C** button on the stage:

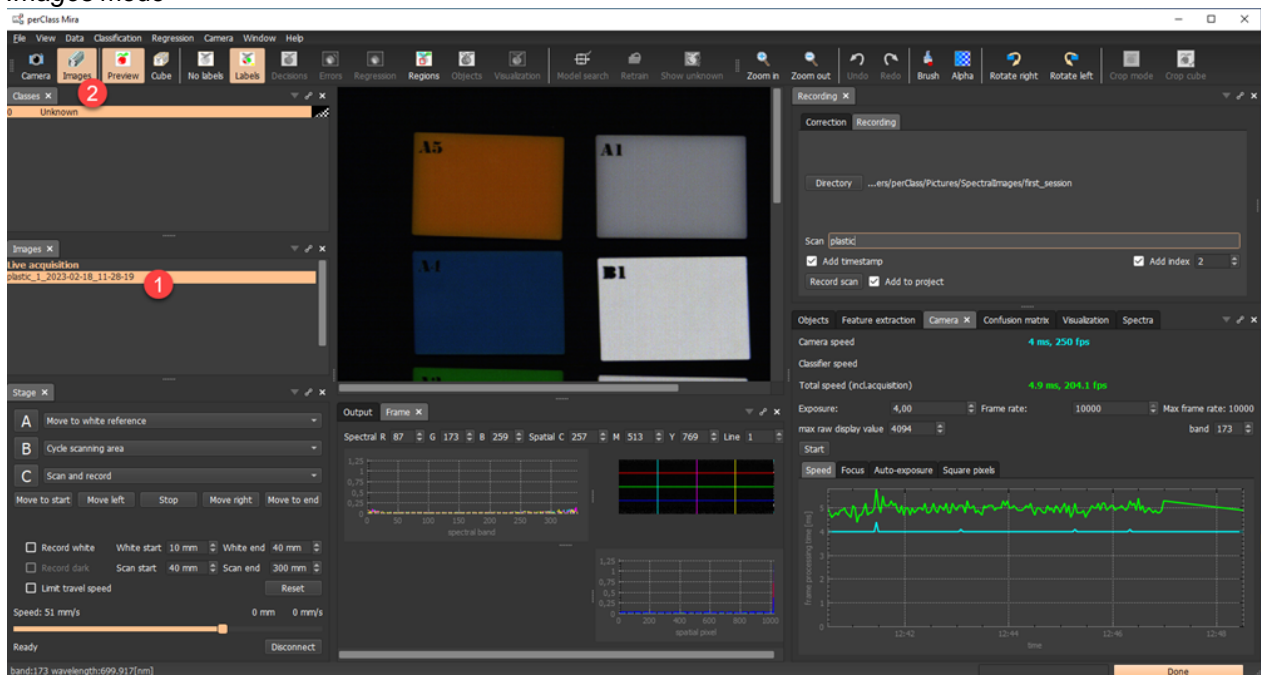






**TIP:** If you use *Scan and Record* command and "nothing happens", double check that you have filled in a scan name in *Recording* panel. If the name field is empty, scan cannot be recorded.

After the scan is recorded, it is loaded in perClass Mira image list <sup>1</sup> and the software switches to the *Images* mode <sup>2</sup>



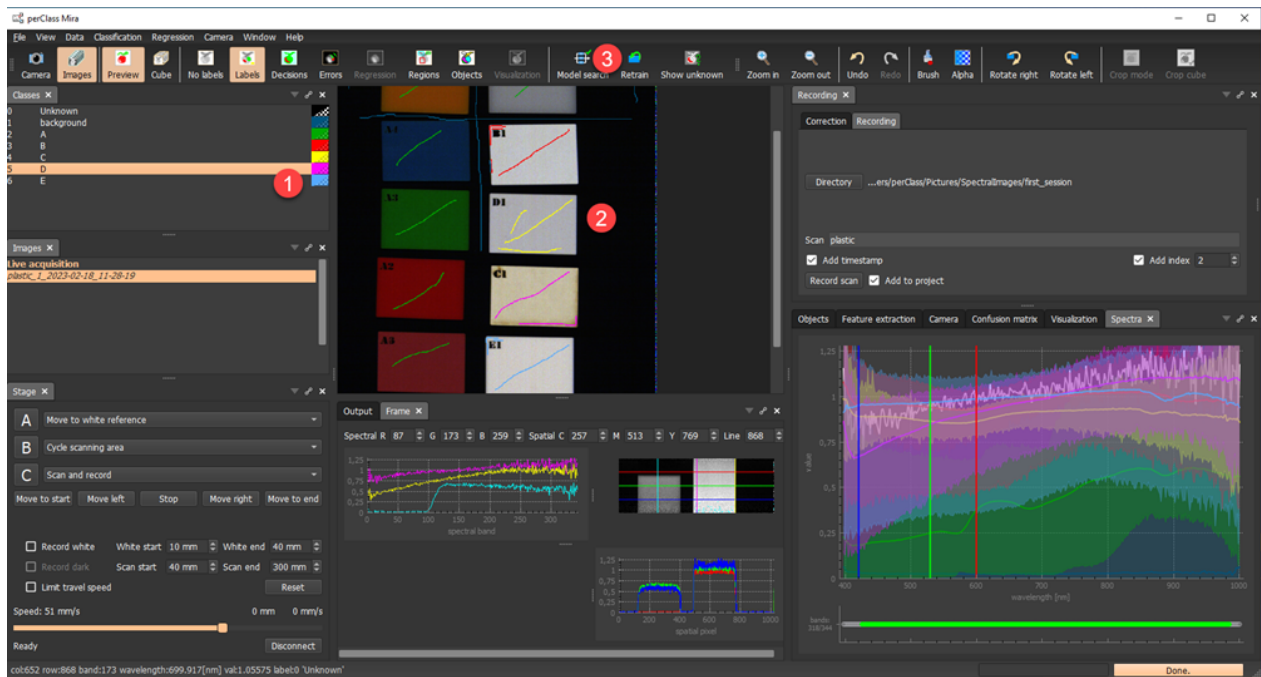
**TIP:** perClass Mira interface is optimized for efficient acquisition of a large number of scans. All you need to do is to place new physiological samples on the scanning table and press the **C** button.

We're now ready to interpret our first scan.

### Building a classifier and applying to live data

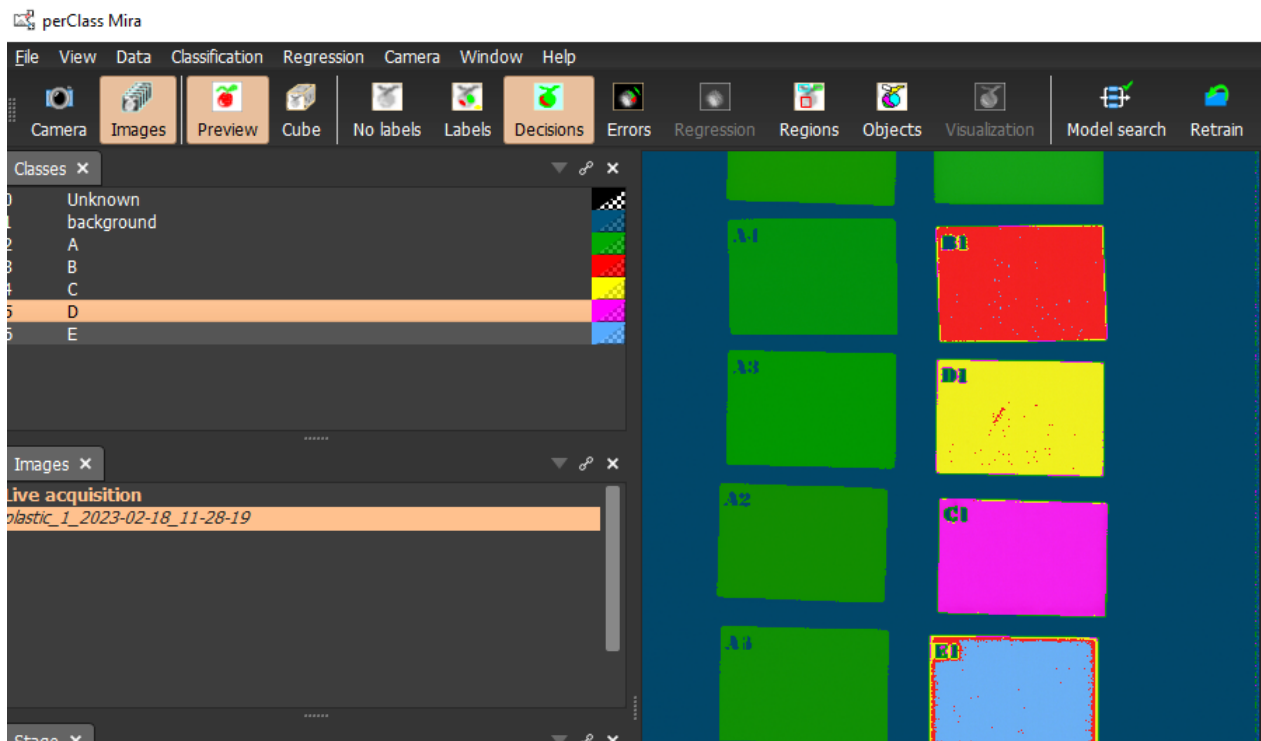
We can design our classifier in the familiar way. We define classes, paint labels and build a model.

In our example, we define background and material classes A - E 1

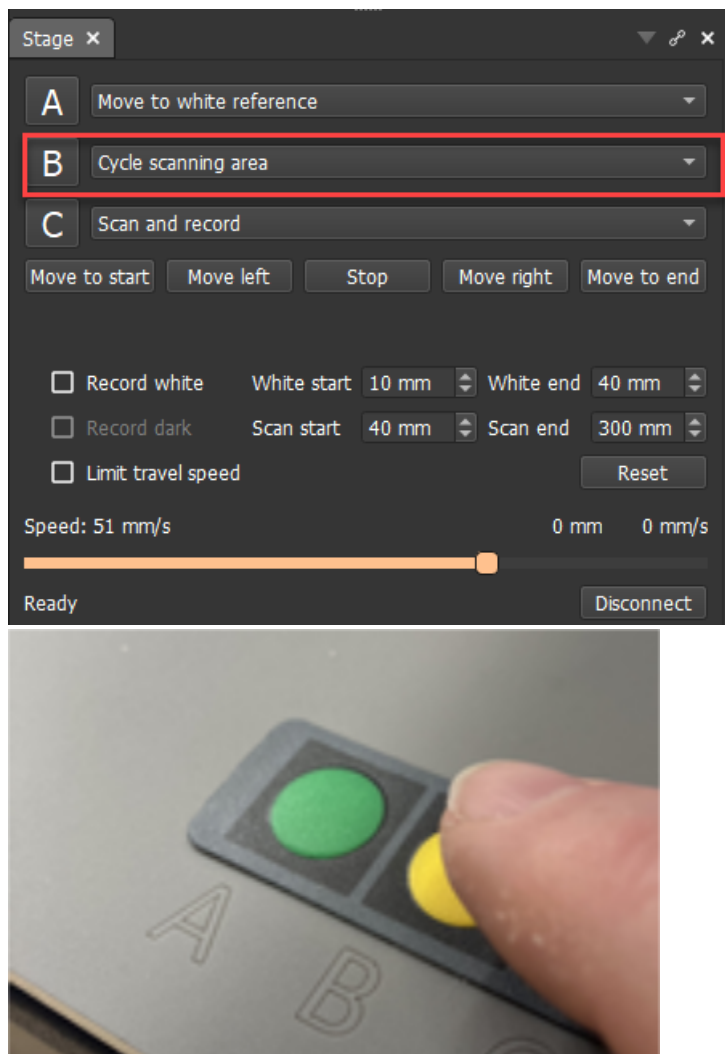


Then, we paint the labels 2 and create a model by pressing *Model search* 3

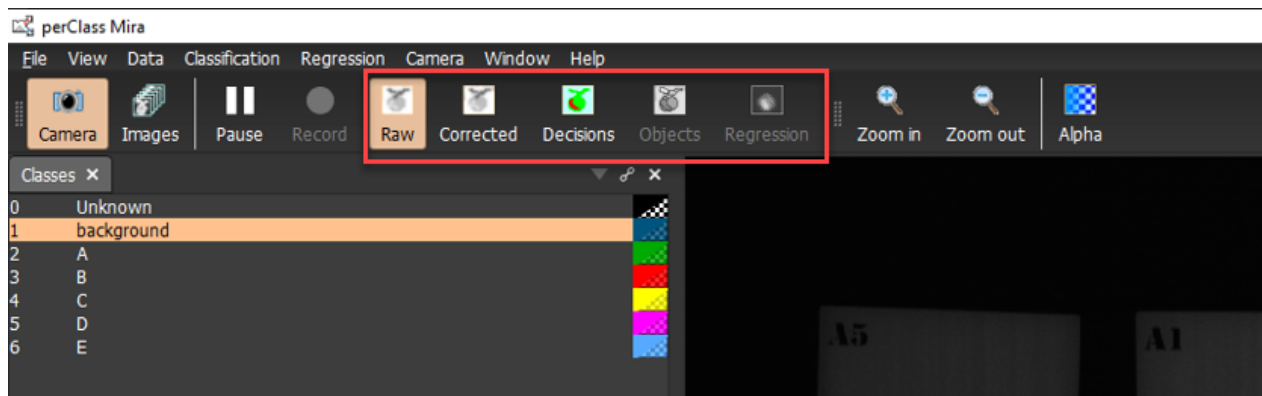
We can see that all different variants of material A can be separated from different white plastics B-E.



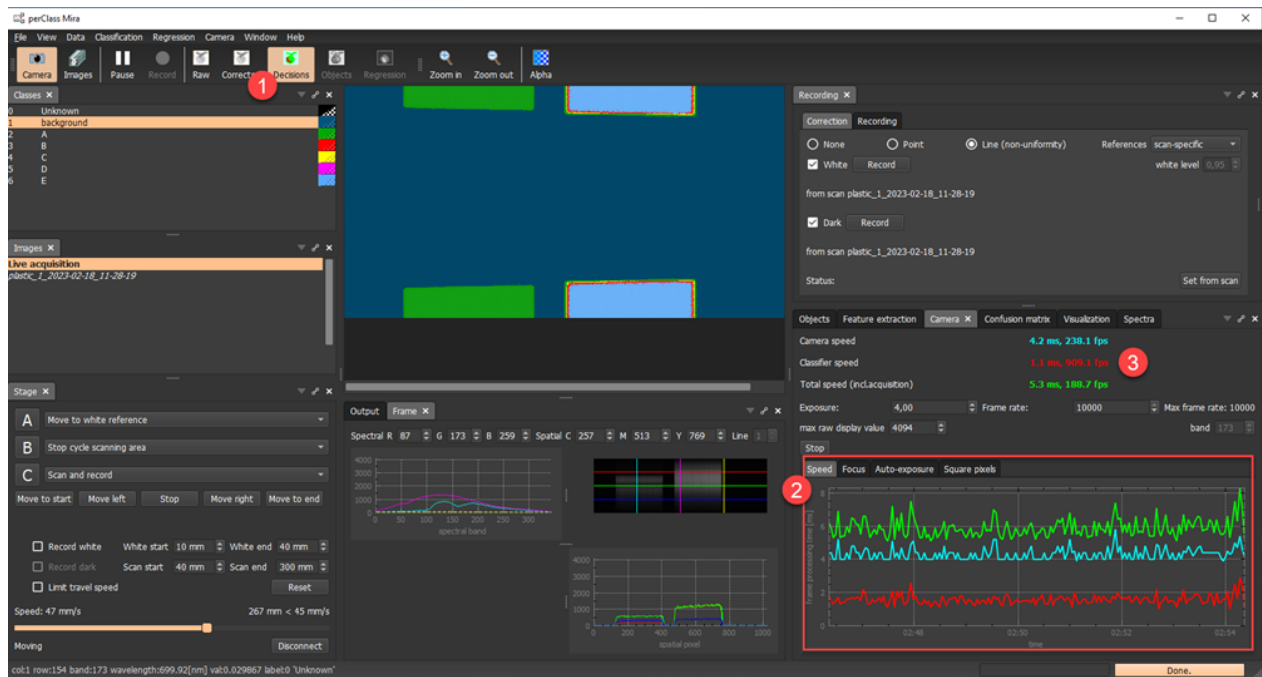
By pressing the **B** button in the *Stage* panel or the physical **B** button, we can start moving the stage (because we selected this command previously):



Note that we now have more options in the *Camera* mode: Apart of the *Raw* and *Corrected* data stream, we can also visualize live *Decisions* of our classification model:



The *Camera* panel will then also show a classifier speed (the red line) <sup>2</sup>.



This concludes our acquisition tutorial. We have learned how to acquire references, record scans, define models and run the full correction and modeling pipeline on a live data stream.

## User guide

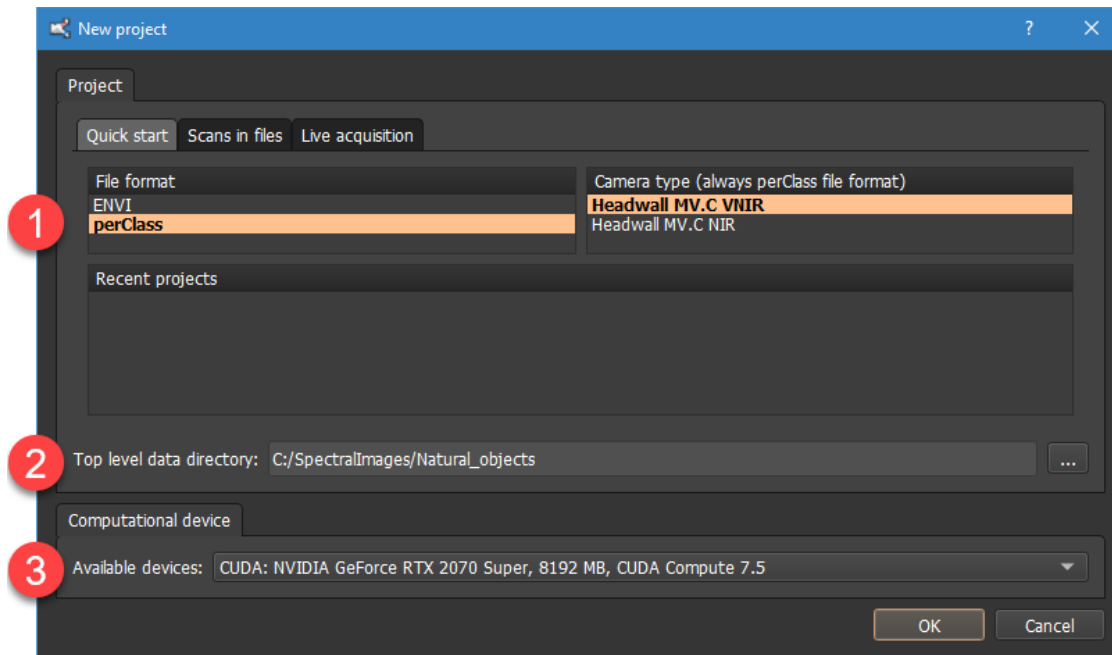
This chapter provides detailed description of individual perClass Mira components.

### New project

Once the software starts, the *New Project* dialog appears.

The default *Quick start* tab allows the user to select three important settings:

1. The project type and camera used for acquisition if any camera is available
2. The top-level data directory
3. The computational device



perClass Mira allows the users to either start from existing scans already stored in files or to acquire new scans from an attached spectral camera.

Note, that perClass Mira supports a broad range of [project types](#) and many common [spectral camera types](#). You may view all available options in the **Scans in files** and **Live acquisition** tabs.

Direct acquisition into perClass Mira *always* uses the "perClass" project type. This assures that users of all camera types can take advantage of identical and complete work-flows for data correction.

The **top-level data directory** 2 defines where all scans are located. It is stored in the project file but can be changed anytime from the *Image list* context menu.

perClass Mira does not write into the data directory unless the user explicitly asks to (for example when exporting the results). The reason is that data directories are assumed to be read-only so that original data set is not altered in processing.

The **Computational device** 3 combo box allows the user to define what computational resource will be used for data processing. Note, that perClass Mira installation comes with two executables: The "perClass\_Mira.exe" that is CPU-only and works on any PC and "perClass\_Mira\_gpu.exe" that offers multiple backends including CPU, NVIDIA GPU and OpenCL CPU/GPU. If no correct GPU drivers are found, the later executable may not be able to start.

**TIP:** In case you experience crashes or very slow operation when using a GPU, please update your GPU drivers to the latest available. For most users this resolves the issues.

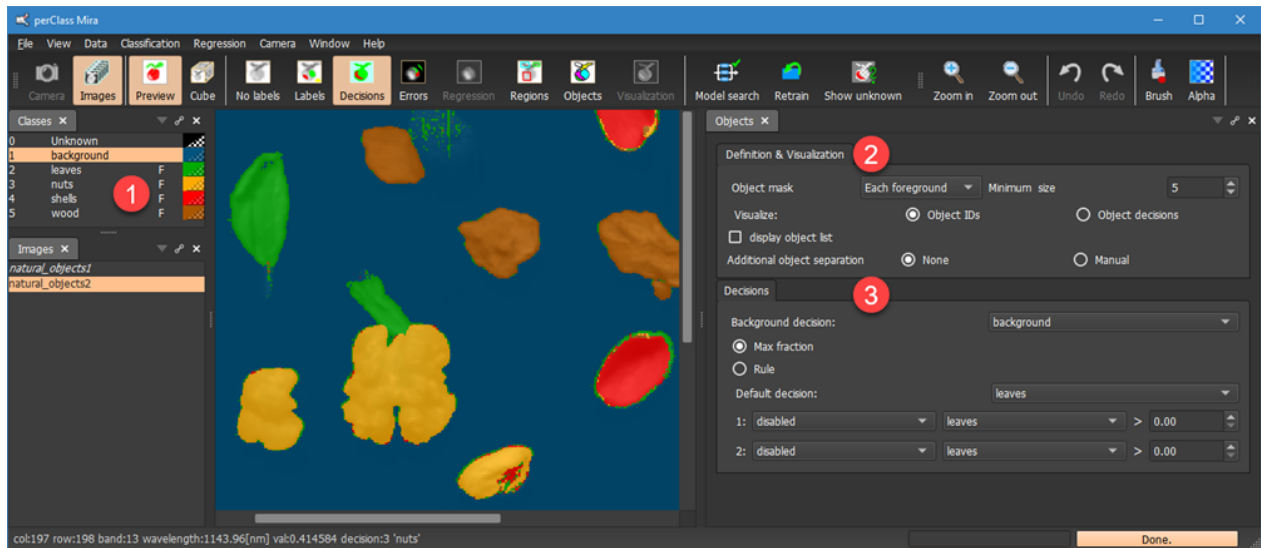
## Objects

Object segmentation defines spatially coherent objects based on pixel classification results. In perClass Mira, many operations and analysis types extend naturally to objects. For example, we may wish to compute spectral index only on objects of interest, model quality of some objects using regression analysis or export mean spectrum per object for further research.

The first step to apply object segmentation is to define one or more *foreground* classes. This can be done using *Image list* context menu and selecting *Flag class as foreground*. Alternatively, the 'F' shortcut toggles

the foreground flag for the currently selected class in the *Class list* 1

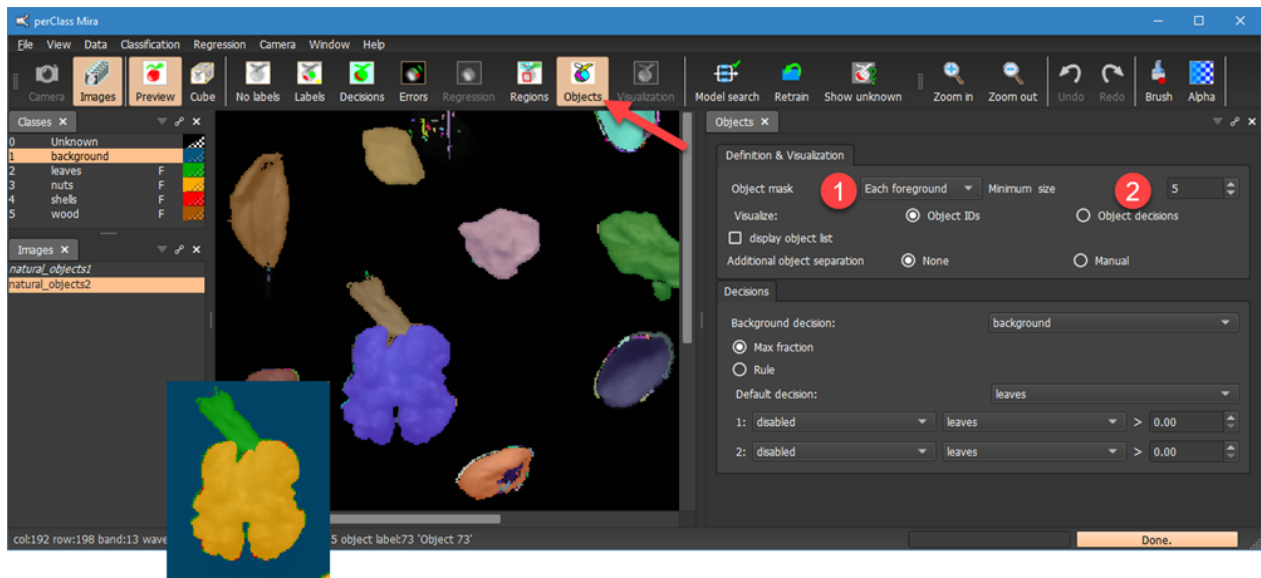
The *Objects* panel presents a separate section <sup>2</sup> where objects are defined (object segmentation) and where objects are classified <sup>3</sup>



## Object segmentation

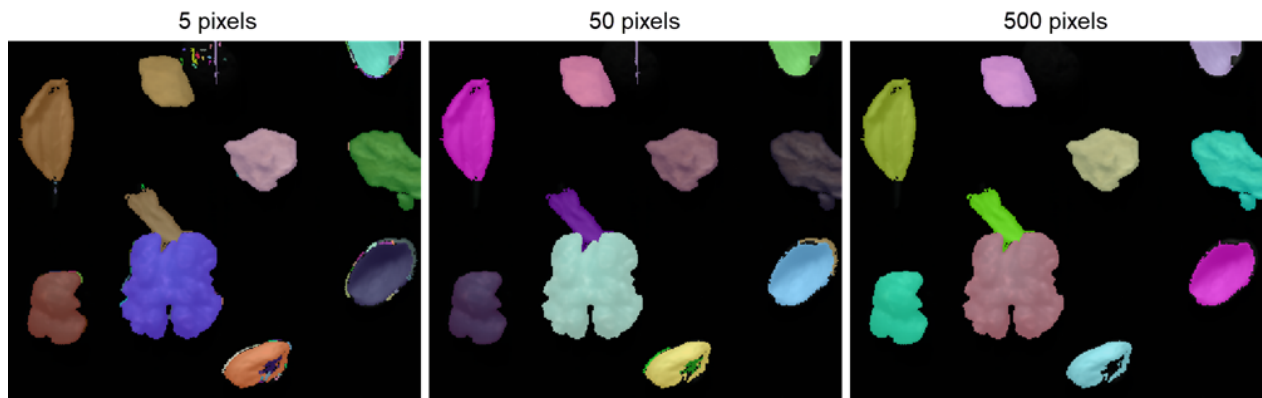
Object segmentation is performed by selecting *Objects* toolbar button. In perClass Mira, object segmentation is controlled by segmentation mask. By default, it is constructed handling each of the

foreground classes separately. This is the *Each foreground* mode <sup>1</sup>. Note, how the leave touching the walnut in pixel decisions are segmented separately.

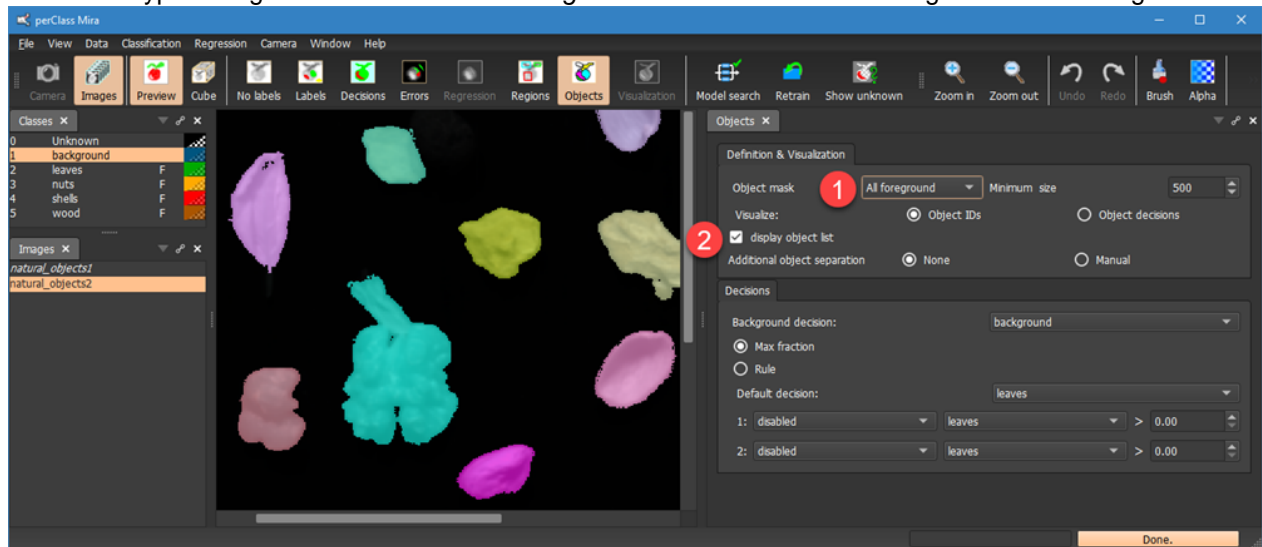


Object segmentation directly removes all objects smaller than specified *minimal size* <sup>2</sup>. Changing the minimum size, we can quickly focus on large structures:

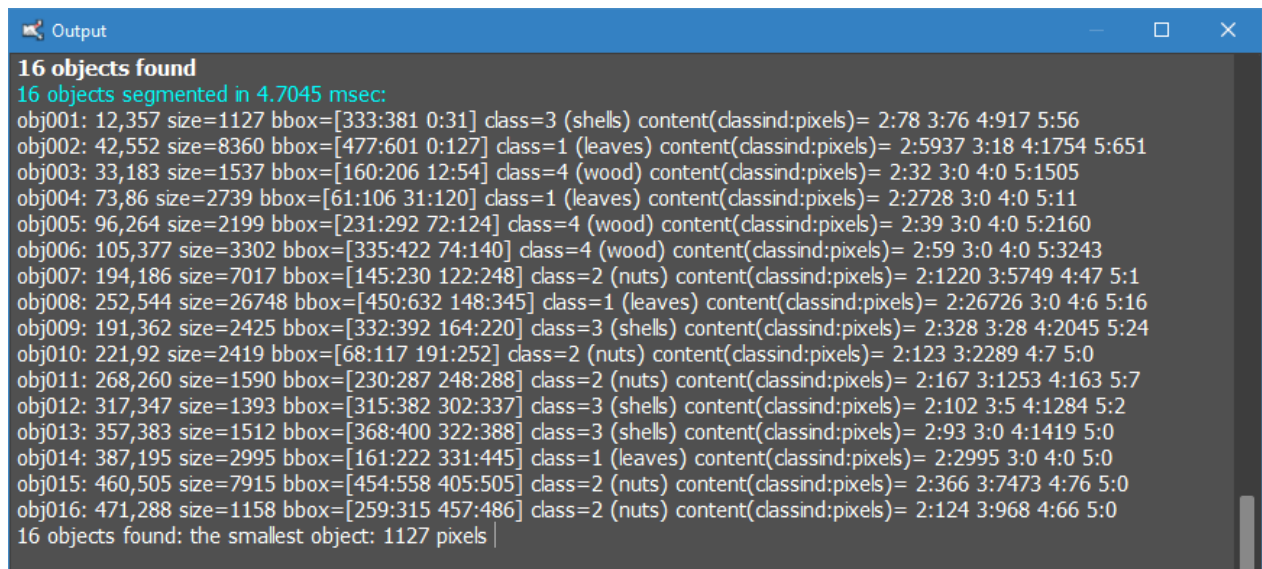




Alternative type of segmentation mask is *All foreground* which combines all foreground classes together:



The *display object list* checkbox enables detailed information on segmented objects in the *Output* window:

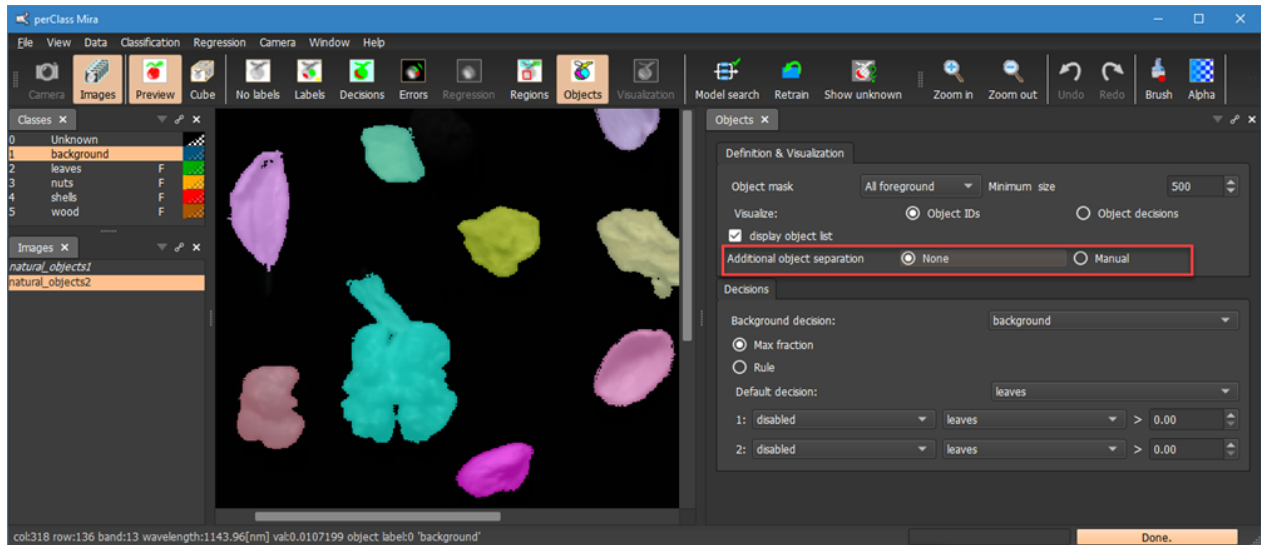


This includes object id, centroid, size in pixels, bounding box and object decision. At the last line, the smallest object size is also provided. This is helpful when adjusting minimum object size.

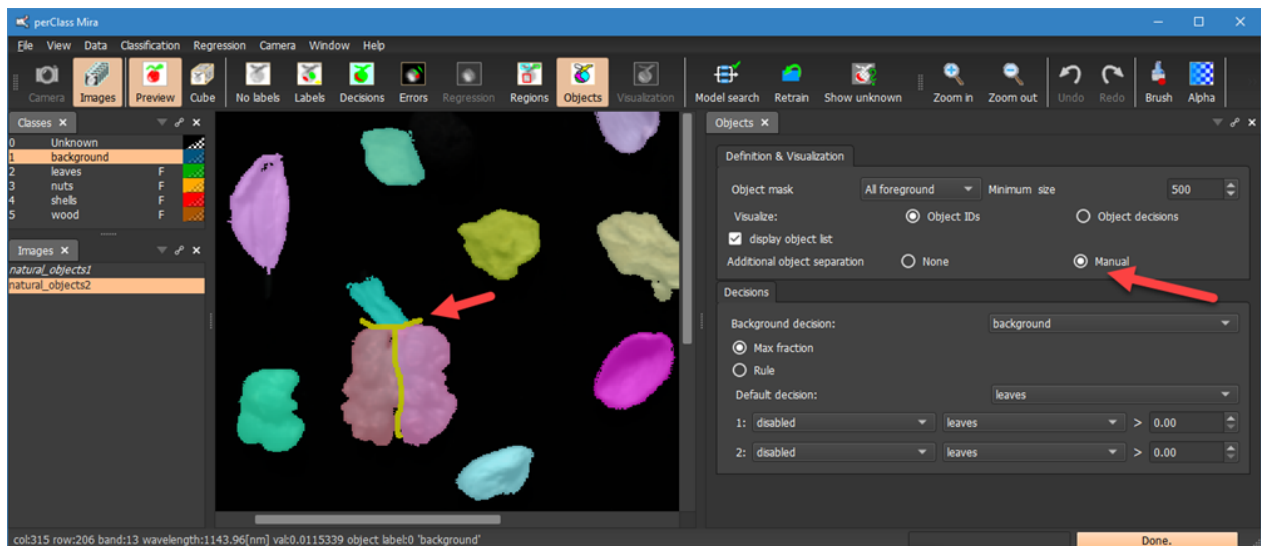
## Object separation

In some situations, we may wish to define objects manually. For example, we may wish to provide very specific local areas for regression modeling or we wish to export mean spectra of specific regions that do

not separate using our pixel classifier. This is possible using the *Additional object separation* tool:



By setting the *Manual* option, we may define object separation by **drawing background labels** in the object segmentation result. In our example, we may force separation of the leaf from the nut even when we use the *All foreground* mode or separate the nut object into two parts (for example, to extract mean spectra from both).



Note, that the manual object separation does not extend to runtime. It is intended for specific highly controlled extraction of data, not for automatic application to new images.

Holding shift, we may remove the separation labels. Clearing separation labels entirely is possible via the context menu:





## Object classification

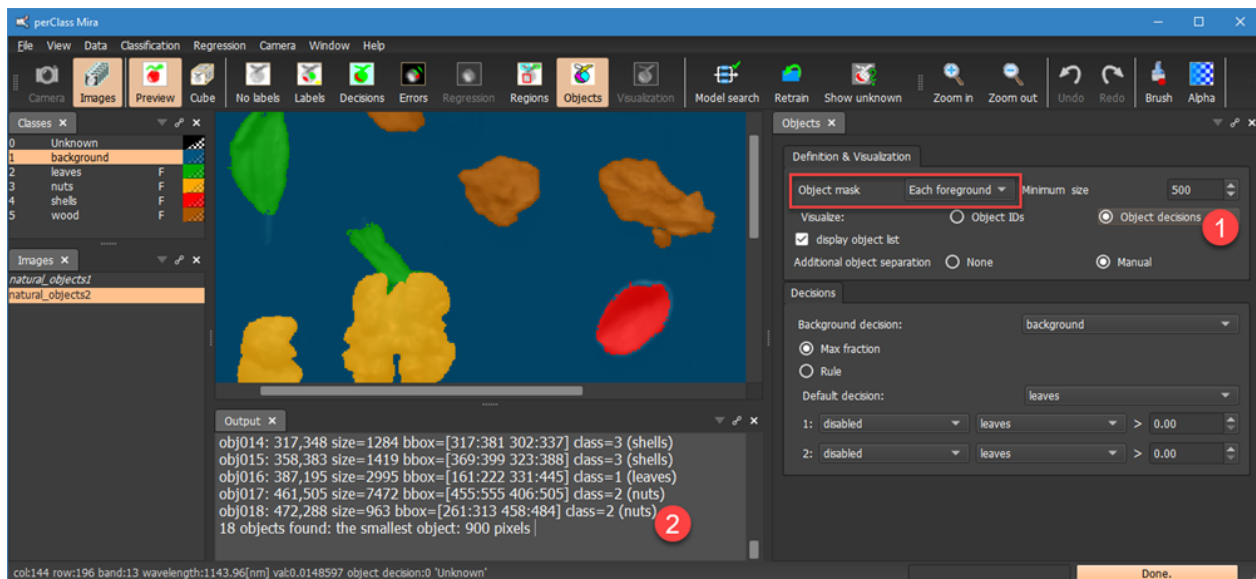
perClass Mira directly applied object classifier after object segmentation. We may view per-object

decisions instead of the object IDs by selecting the *Object decisions* option <sup>1</sup>

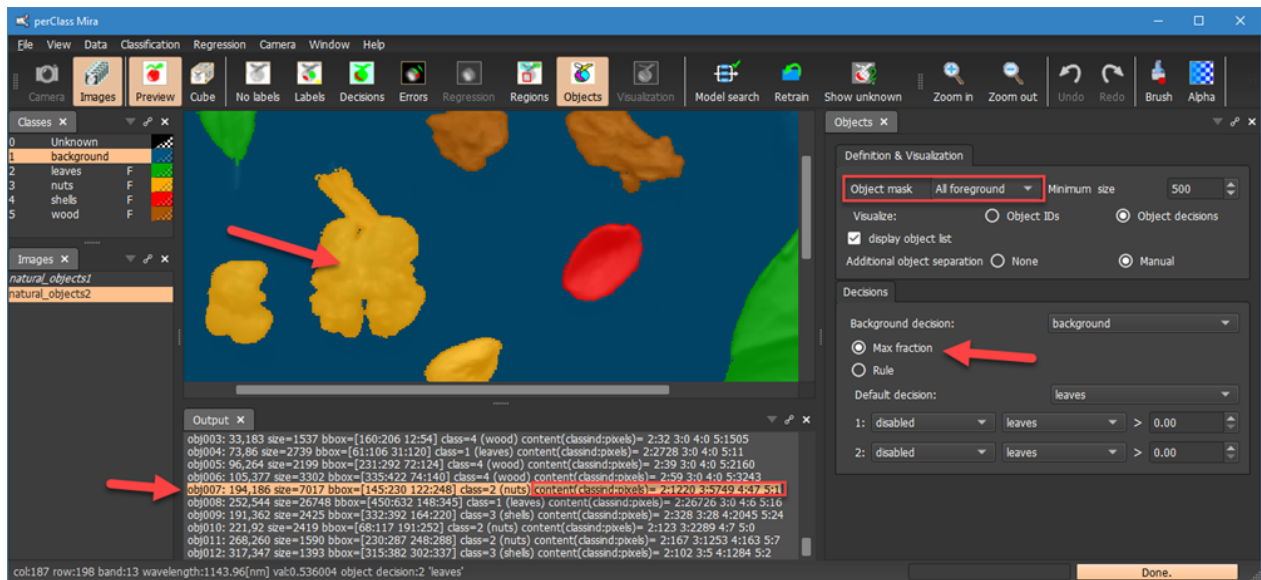
In *Each foreground* mode, the classification is implicit i.e. each segmented class is the object decision.

This is indicated by the class color and displayed in the object list <sup>2</sup>,

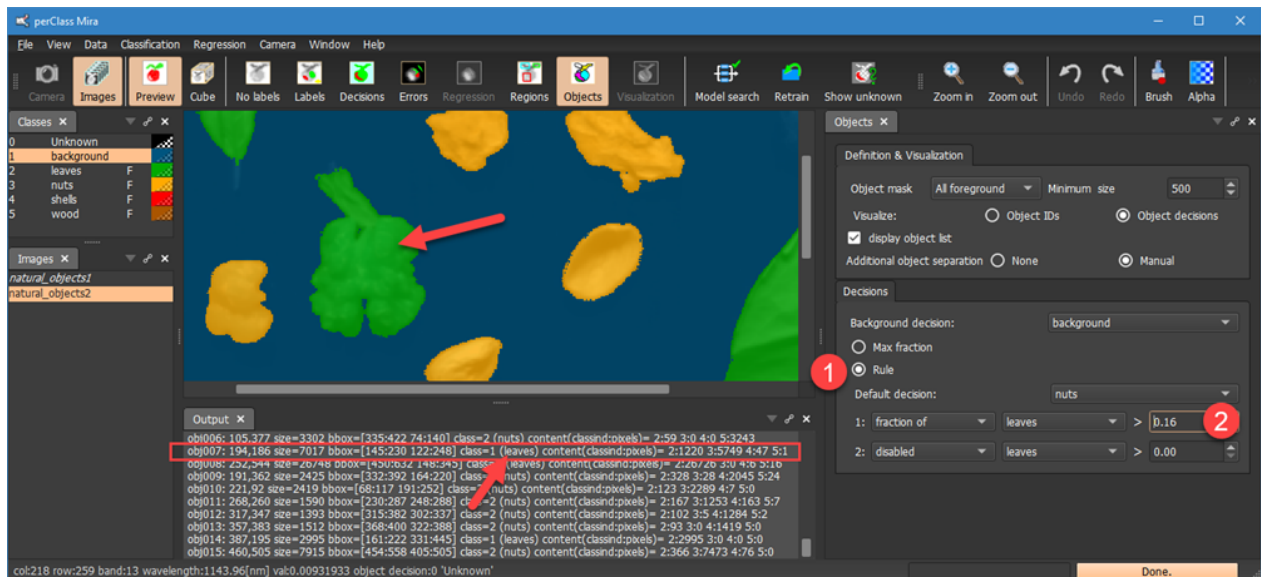
Note, that while the object decision visualization seems very similar to pixel decisions in our example, there are many differences such as removed small objects along the edges.



In the *All foreground* mode, classification is performed based on **object content**. Below, we can see that the compound object composed of the walnut touching the leaf in the center is classified as a nut. By default the classification is based on maximum fraction (majority voting). Each object in *All foreground* mode provides information on pixel counts of each foreground class. See the highlighted line in the object output for object 7. Because the majority of pixels is classified into *nuts* class, the entire object is as well.



Alternative classification rule can be defined in the *Decisions* section. In the example below, we see the rule such that if a fraction of leaves class is higher than 16%, the object is classified as *leaves*. That's what happens to object 7. Currently, two rules can be defined based on fraction or absolute number of pixels.



## Regions

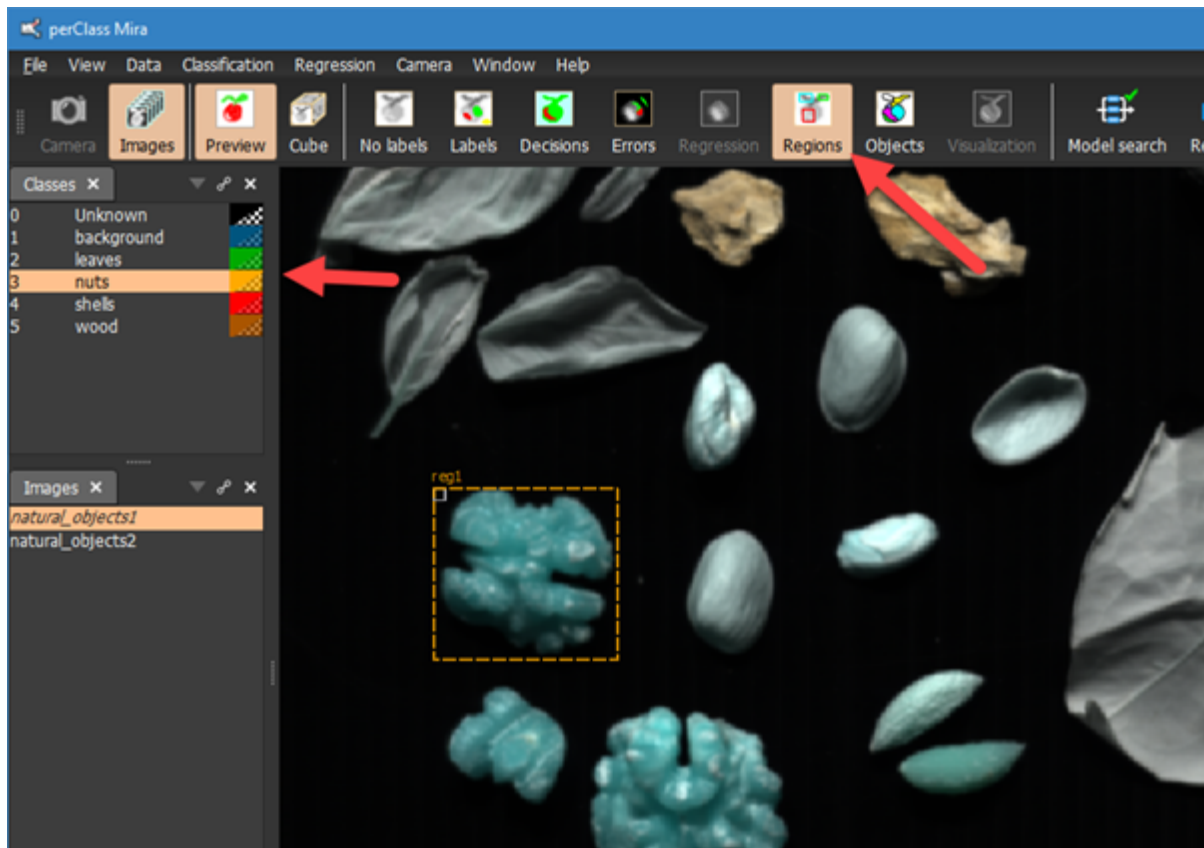
Regions tool in perClass Mira provides annotation of image areas. It is used for several purposes:

1. It is always available as an [area annotation tool](#)
2. It allows **extraction of information** from specific image areas
3. It provides us with **ground truth information for object classification**
4. It enables localized annotation of specific objects for regression

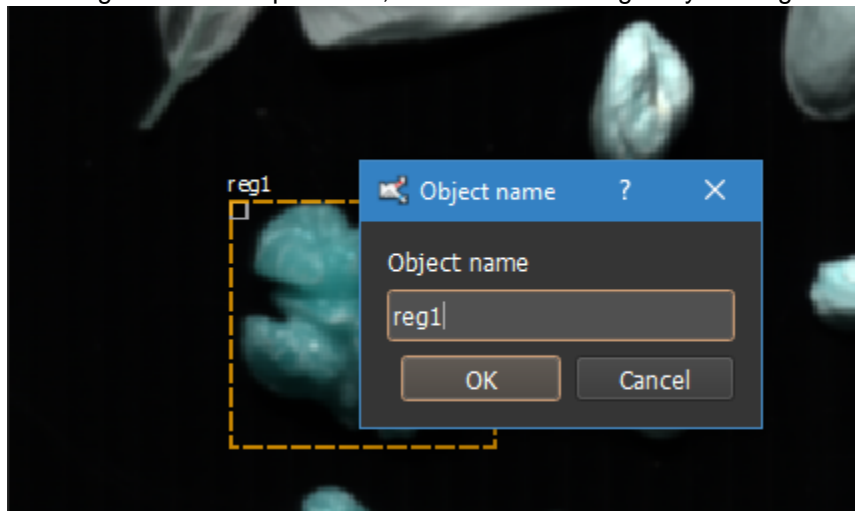
### Region annotation

Region annotation tool is always available by selecting the *Regions* toolbox button. We may then draw a rectangle anywhere in the image. This tool is useful, for example, for a **quick annotation of image content by domain experts**.

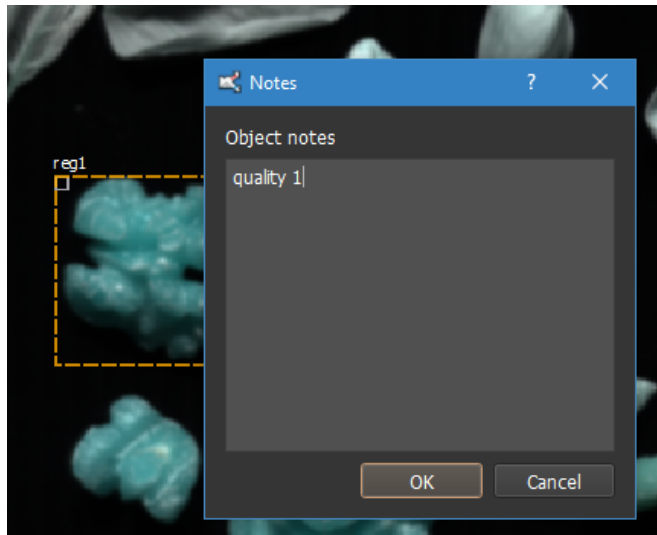
The current class, selected in the class list, is defining the color/class of the region.



Each region has a unique name, which can be changed by clicking:



User can attach arbitrary text content to the region by clicking the left upper rectangle:



Regions with existing content are highlighted using green rectangle. When hovering over, the content is displayed in a tool-tip:



## Confusion matrix

When designing a robust classification algorithm, we need to understand its performance and robustness in detail. In perClass Mira, we can use the *Confusion matrix* tool to understand and fine-tune pixel classification performance.

In this section, we use the potato virus data set to illustrate *Confusion matrix* tool <sup>1</sup> and its use in performance understanding and fine-tuning. When a pixel classifier is trained, the confusion matrix on training set is always available in the *Training set* tab of the *Confusion matrix* panel.

Confusion matrix shows detailed report on classifier performance. In rows, it provide information on all labeled examples in the training set (the ground truth). In the columns, it captures the classifier decisions on these examples. Ideally, all labeled examples are allocated to the same categories. The confusion matrix would show only diagonal elements (displayed in green). In practice, some examples are misclassified. These show up off-diagonal and are rendered in red color.

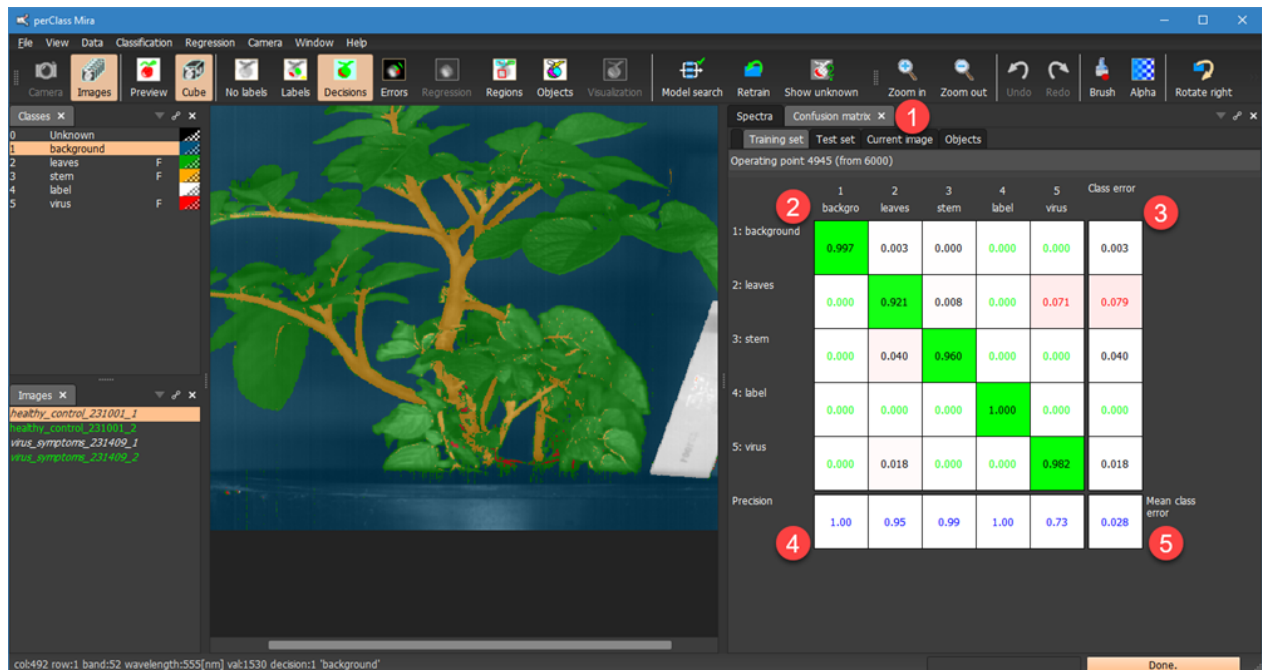
By default, the confusion matrix is normalized by each row. Thie means that the entries represent accuracies on-diagonal and error rates off-diagonal. The sum of the off-diagonal errors i.e. the **class error**

is displayed in the right-most column <sup>3</sup>. Class error reflects what percentage of the ground truth pixels is misclassified.

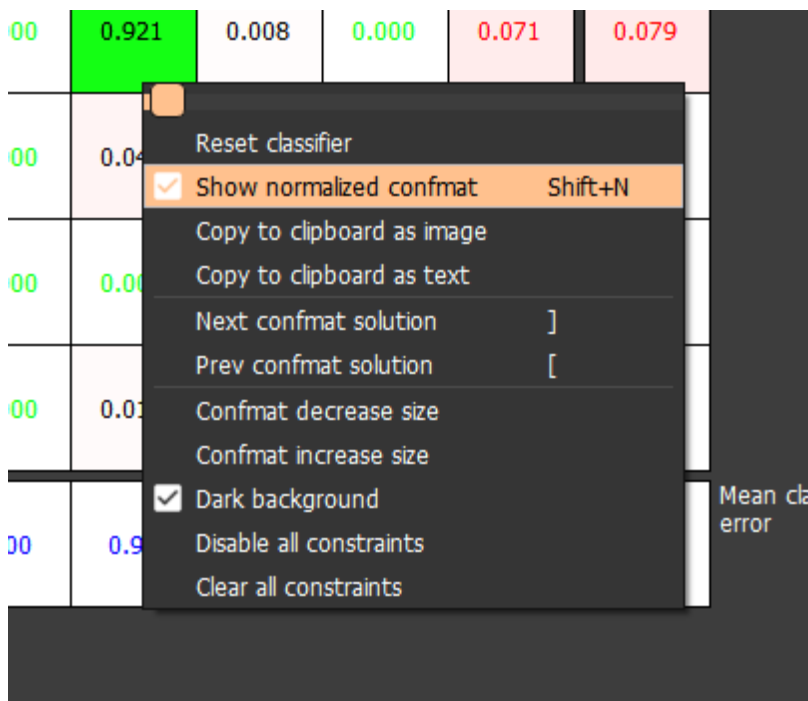
Similarly, the **per-class purity** is displayed in the bottom row <sup>4</sup>. This denotes the fraction of each class decisions that is actually classified correctly. This allows us to quickly judge if a specific classifier decision is trustworthy. For example, when our classifier provides decisions on *leaves*, it is correct in 95% of cases. However, when classifying the *virus* it is correct only on 73% of labeled pixels in our training set.

Finally, the right-bottom corner provides one summary performance indicator: The mean error over classes - the average of per-class error rates.

The value of confusion matrix is in providing detailed understanding of classifier behaviour. While 2.8% mean error does not seem too high, confusion matrix allows us to learn quickly, that 7% of healthy leaves is being misclassified as a virus (the are false positives).



Instead of normalized matrix, we may wish to display the absolute pixel counts in each field. This is possible by disabling the normalization in the context menu:



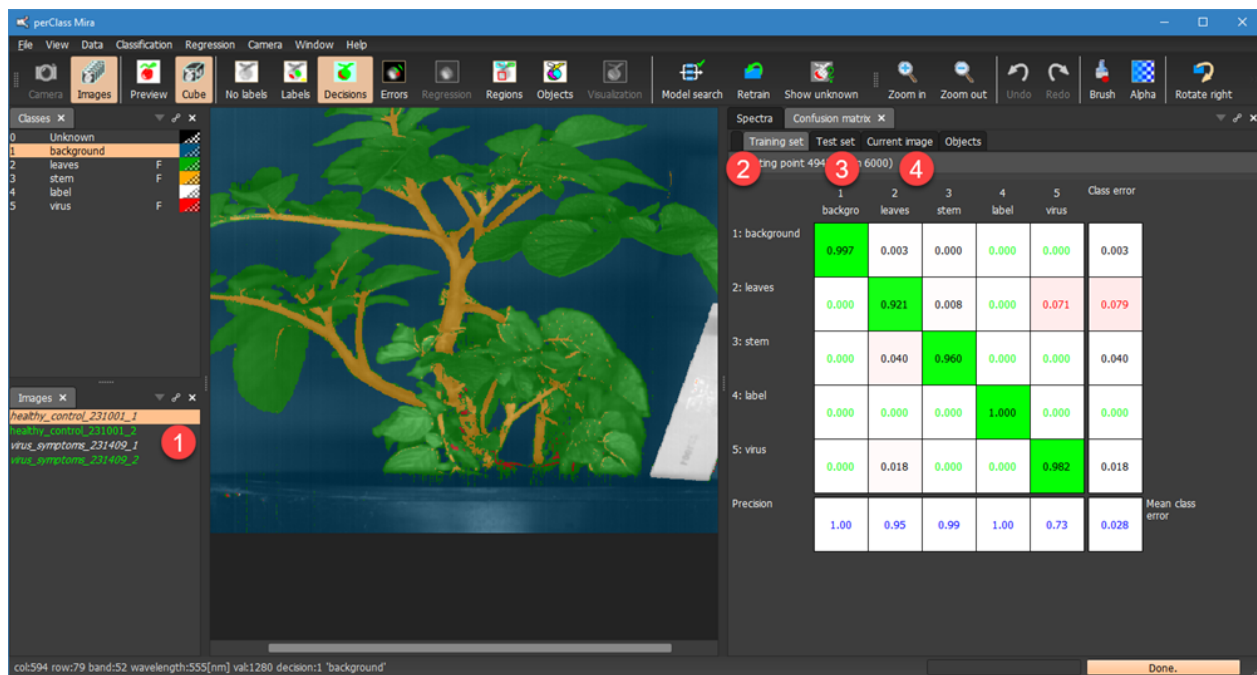
The not-normalized confusion matrix may highlight that some of our classes are undersampled. For example, while our *virus* class contains only 165 labeled pixels, the background contains almost four thousand.

Spectra Confusion matrix ×						
Training set Test set Current image Objects						
Operating point 4945 (from 6000)						
	1 backgro	2 leaves	3 stem	4 label	5 virus	True sample count
1: background	3904	11	1	0	0	3916
2: leaves	0	790	7	0	61	858
3: stem	0	26	628	0	0	654
4: label	0	0	0	579	0	579
5: virus	0	3	0	0	162	165
Decision count	3904	830	636	579	223	6172
						Total sample count

### Test set confusion matrix

By default, the confusion matrix <sup>2</sup> is displayed that is estimated from training set. This means from all images <sup>1</sup> that are not flagged as a test set.





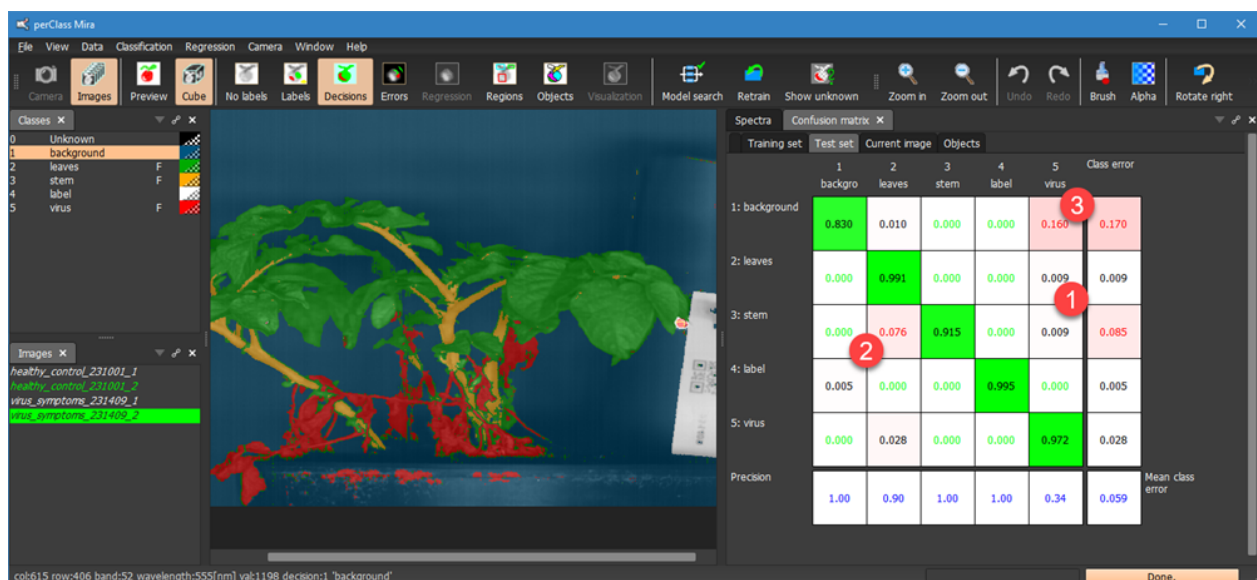
In order to properly evaluate any classification solution, we need **good performance on the independent test set**. The reason is that we need confirmation of **generalization capability** of our classifier on example unseen in training. In perClass, testing is defined on the level of images. Images flagged as a test set are not used for training the classifier.

**NOTE: In order to take any change in test image flags into account, we need to retrain the model!**

To estimate the test set confusion matrix, we switch to the *Test set* tab <sup>3</sup>. In our example, we will observe that the confusion matrix is not complete. This is because not all classes were represented (labeled) in our test scans.

Spectra Confusion matrix X						
	1	2	3	4	5	Class error
	backgro	leaves	stem	label	virus	
1: background	0.937	0.009	0.000	0.000	0.055	0.063
2: leaves	---	---	---	---	---	nan
3: stem	---	---	---	---	---	nan
4: label	---	---	---	---	---	nan
5: virus	---	---	---	---	---	nan
Precision	1.00	0.00	nan	nan	0.00	nan
						Mean class error

We add relevant labeling to the two test scans and switch to *Training set* confusion matrix and back to the *Test set* matrix to update it:



Note, that we do not observe the same misclassification of *leaves* into *virus* <sup>1</sup> as in the training set. In addition, we can see higher errors in two new fields, misclassifying some stem as leaves <sup>2</sup> and background as virus <sup>3</sup>.

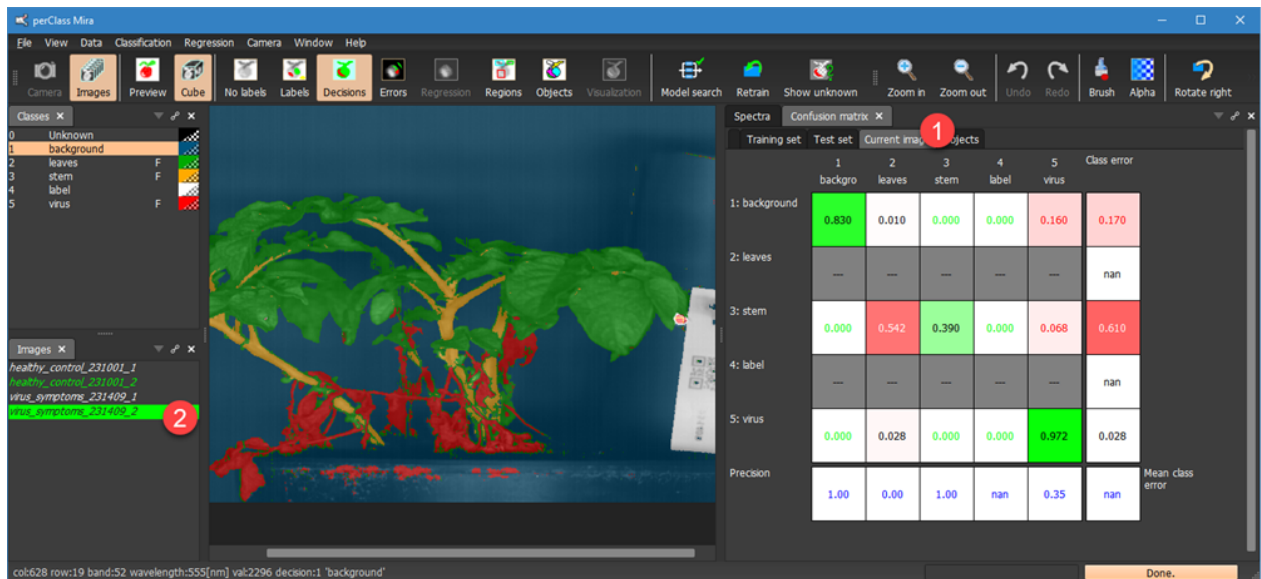
It is very difficult from the performance estimates only to judge to what extent are these relevant errors. We need to understand what pixels these misclassifications represent in the image. That is greatly simplified by



perClass Mira *Errors* tool and the *Current image* confusion matrix.

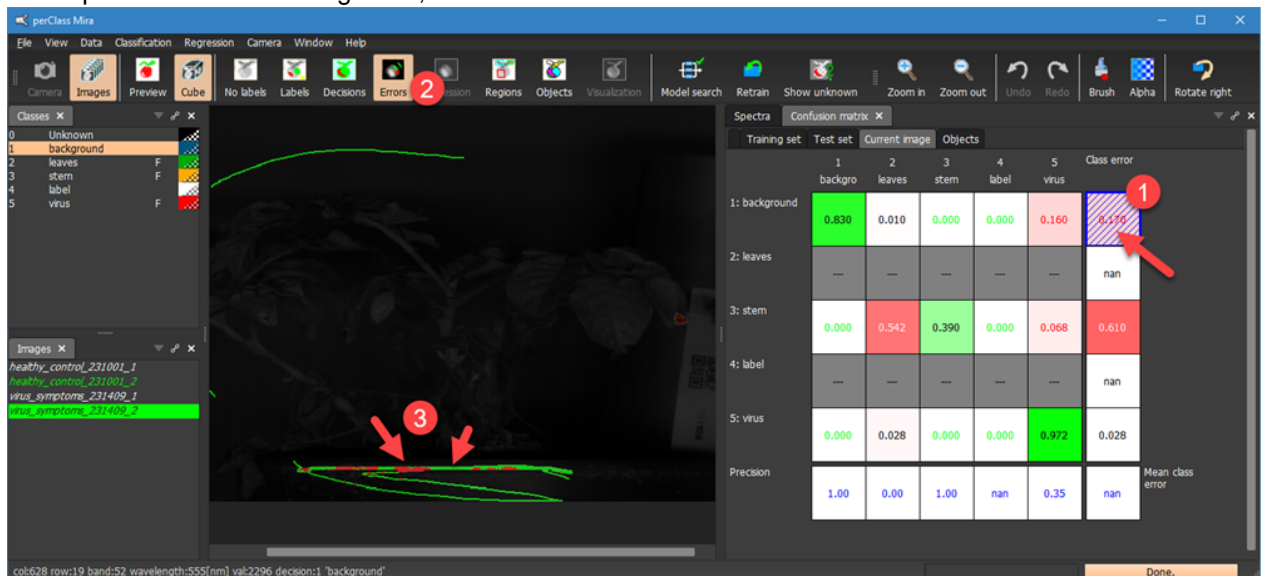
## Current image confusion matrix

When we switch to *Current image* confusion matrix, we can easily introspect how individual confusion matrix entries map to image pixels. The confusion matrix <sup>1</sup> is displayed on for the pixels labeled in the selected image <sup>2</sup>. Note, that again, we may miss some of the classes. The respective rows of the confusion matrix then remain empty.

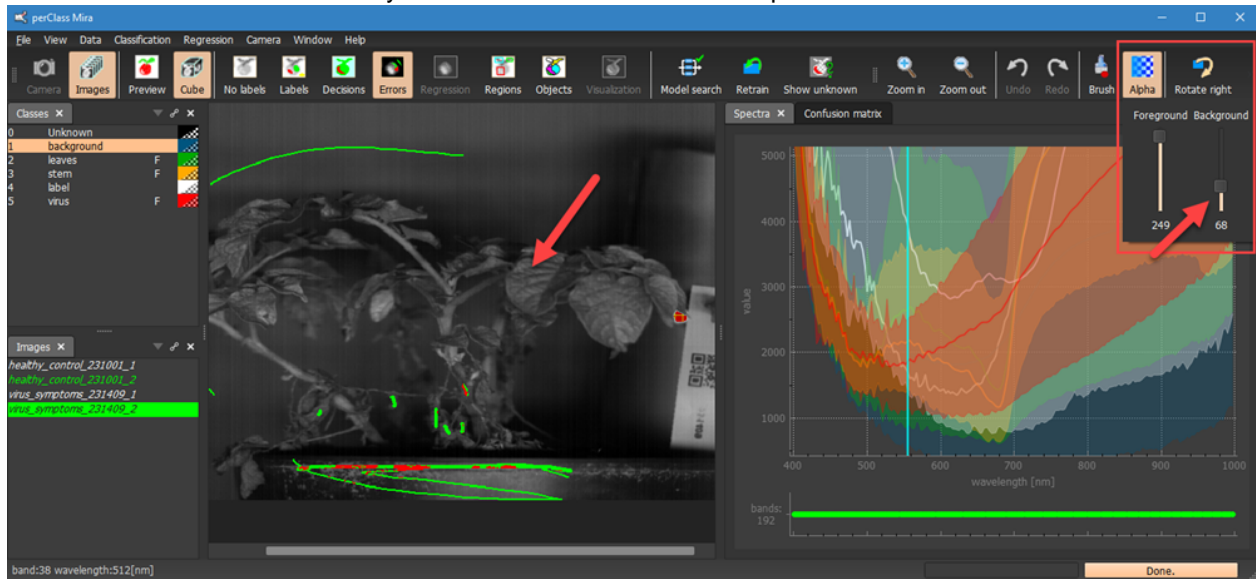


The *Current image* confusion matrix is fully interactive. When we hover over the confusion matrix entries, errors at the pixel level are visualized over the image.

For example, moving the mouse over the *background* class error <sup>1</sup>, The *Errors* mode <sup>2</sup> is enabled. We can see only the labeled pixels falling into the specific field of the confusion matrix. The pixels correctly classified by our model are rendered in green and the misclassified pixels in red. In our example, we can see all pixels labeled as background, some misclassified into *virus* and *leaves* classes.



We may wish to adjust transparency of foreground and background layers using the *Alpha* toolbar button. This allows us to see more clearly what structures do the errors represent.



## Optimizing classifier performance

In perClass Mira, classifier performance may be further fine-tuned and optimized using the confusion matrix tool.

Tuning the classifier performance to application-specific requirements is a basic necessity in any practically deployed machine learning system. It is because default way that statistical models make decisions largely depend on the class abundance in the training set which typically does not correspond to application needs.

perClass Mira provides two ways to tune classifier performance in the confusion matrix:

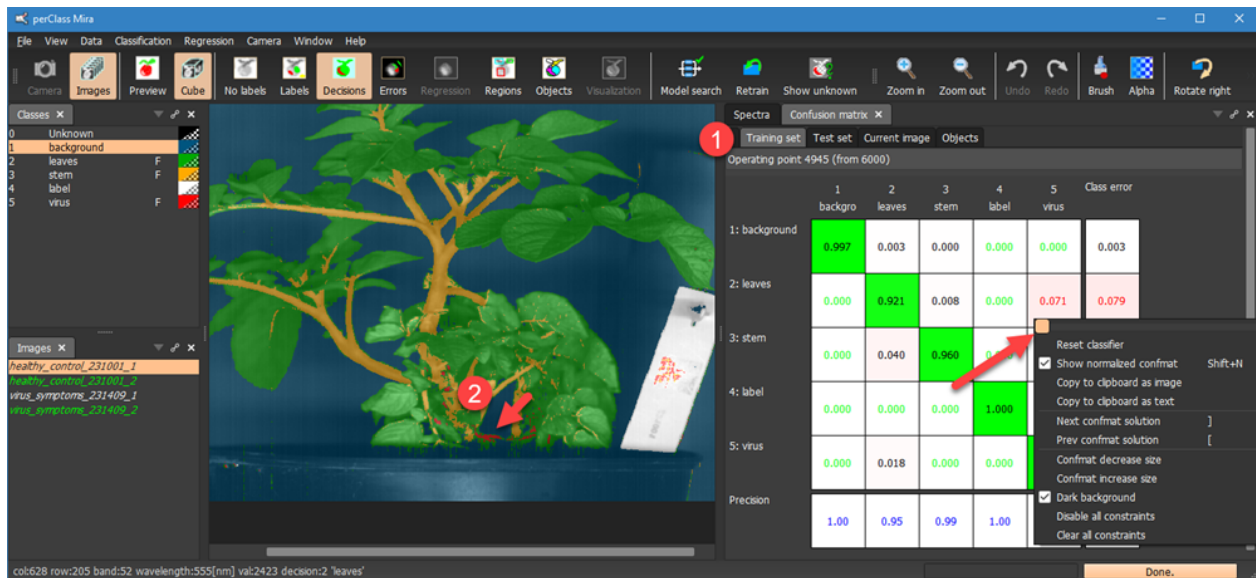
1. [Cost sensitive optimization](#)
2. [Performance constraints](#)

Note, that all classifier optimization in perClass Mira happens on the training set, NOT the test set. This is very important point. Test set, in perClass Mira, is considered only for performance evaluation, not for any form of model tuning.

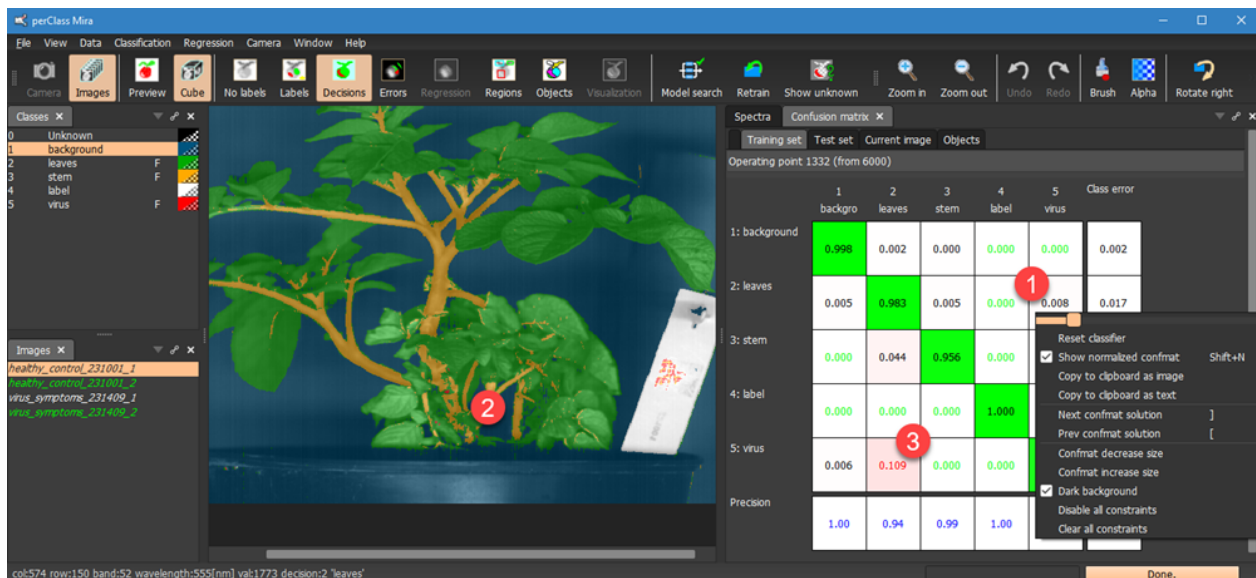
### Cost sensitive optimization

In the *Training set* confusion matrix <sup>1</sup>, we right-click on the field that we wish to optimize. In our example, we wish to lower the 7% of leaves misclassified as virus. Some of these pixels are pointed by

arrow <sup>2</sup> The slider in the context menu allows us to increase the cost for this entry:



Below, we can see that, by adjusting the slider <sup>1</sup>, we can directly see classifier decisions changing. The virus misclassifications in the area <sup>2</sup> disappeared. However, another entry in the confusion matrix shows error increase <sup>3</sup>. It represents the virus pixels, misclassified as leaves.

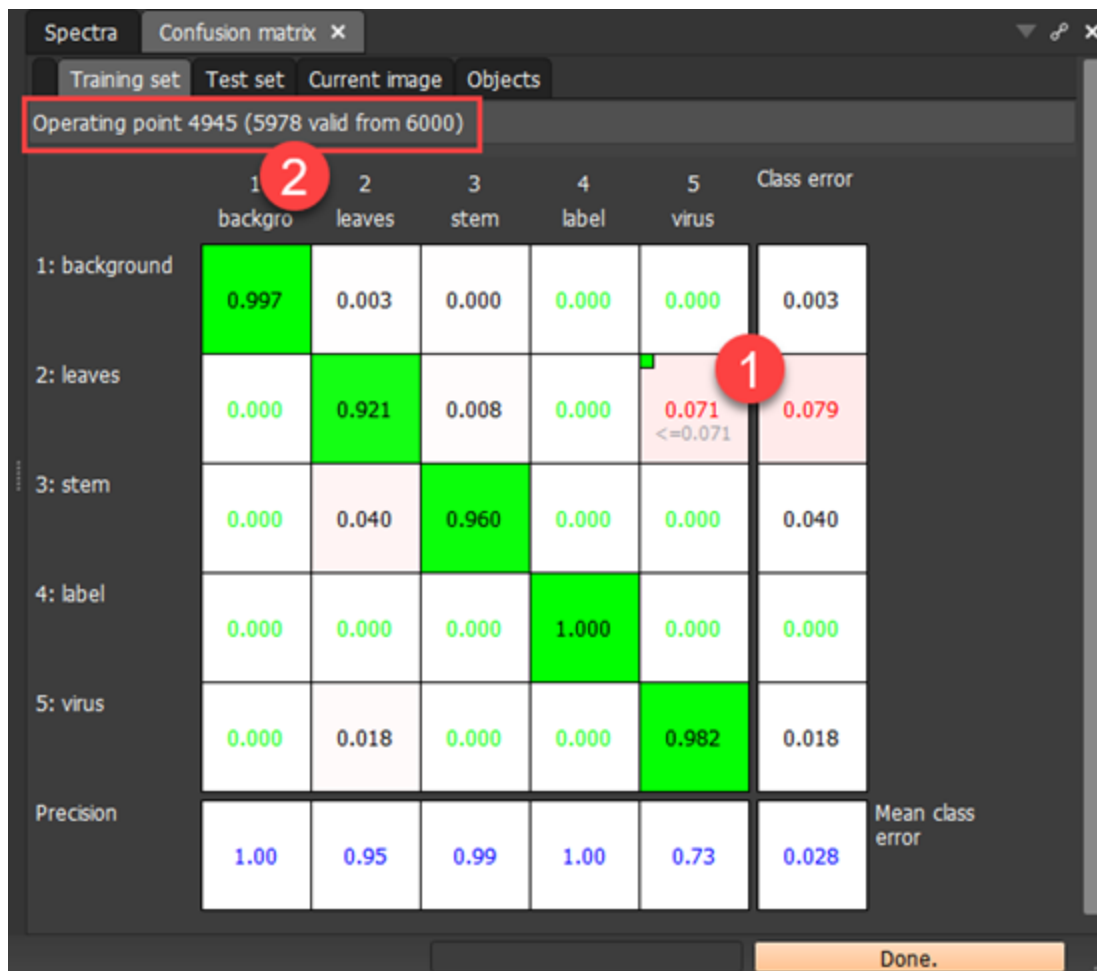


We cannot see these pixels on the current image of the healthy (control) plant as it does not contain any virus infection. We can switch to training image with virus symptoms to investigate impact of this adjustment on true virus class.

### Performance constraints

The alternative way of fine-tuning the classifier performance is by defining performance constraints. This means that we limit the error or accuracy on certain field or fields. We can do that by double clicking any field of the matrix.

In our example, we double click on the *leaves* misclassified as *virus* <sup>1</sup>. A small green box appears in its corner and the current value is listed below estimated error of our solution. The total number of solutions <sup>2</sup> is decreased by each new constraint.



In order to tune the constrain, move mouse over the constrained field, hold Ctrl and use mouse wheel to adjust the constrain. The new solution is displayed.

Spectra Confusion matrix ×						
Training set Test set Current image Objects						
Operating point 1332 (5975 valid from 6000)						
	1 backgro	2 leaves	3 stem	4 label	5 virus	Class error
1: background	0.998	0.002	0.000	0.000	0.000	0.002
2: leaves	0.005	0.983	0.005	0.000	0.008 ≤0.061	0.017
3: stem	0.000	0.044	0.956	0.000	0.000	0.044
4: label	0.000	0.000	0.000	1.000	0.000	0.000
5: virus	0.006	0.109	0.000	0.000	0.885	0.115
Precision	1.00	0.94	0.99	1.00	0.95	0.036
						Mean class error

Multiple constraints can be defined on both errors and accuracy fields:

Spectra Confusion matrix ✕						
Training set Test set Current image Objects						
Operating point 4763 (2536 valid from 6000)						
	1	2	3	4	5	Class error
	backgro	leaves	stem	label	virus	
1: background	0.999	0.001	0.000	0.000	0.000	0.001
2: leaves	0.010	0.966	0.002	0.000	0.021 ≤0.061	0.034
3: stem	0.000	0.064 ≤0.064	0.936 ≥0.896	0.000	0.000	0.064
4: label	0.000	0.000	0.000	1.000	0.000	0.000
5: virus	0.012	0.073 ≤0.079	0.000	0.000	0.915	0.085
Precision	1.00	0.93	1.00	1.00	0.89	0.037
						Mean class error

By clicking the tiny squares in top-left corners of the constrained fields we may disable / enable individual constraints.

Spectra

Confusion matrix X

Training set

Test set

Current image

Objects

Operating point 4763 (4845 valid from 6000)

	1	2	3	4	5	Class error
	backgro	leaves	stem	label	virus	
1: background	0.999	0.001	0.000	0.000	0.000	0.001
2: leaves	0.011	0.966	0.002	0.000	0.021 ≤0.061	0.034
3: stem	0.000	0.064 ≤0.064	0.936 ≥0.896	0.000	0.000	0.064
4: label	0.000	0.000	0.000	1.000	0.000	0.000
5: virus	0.012	0.073 ≤0.079	0.000	0.000	0.915	0.085
Precision	1.00	0.93	1.00	1.00	0.89	0.037
						Mean class error

## Object confusion matrix

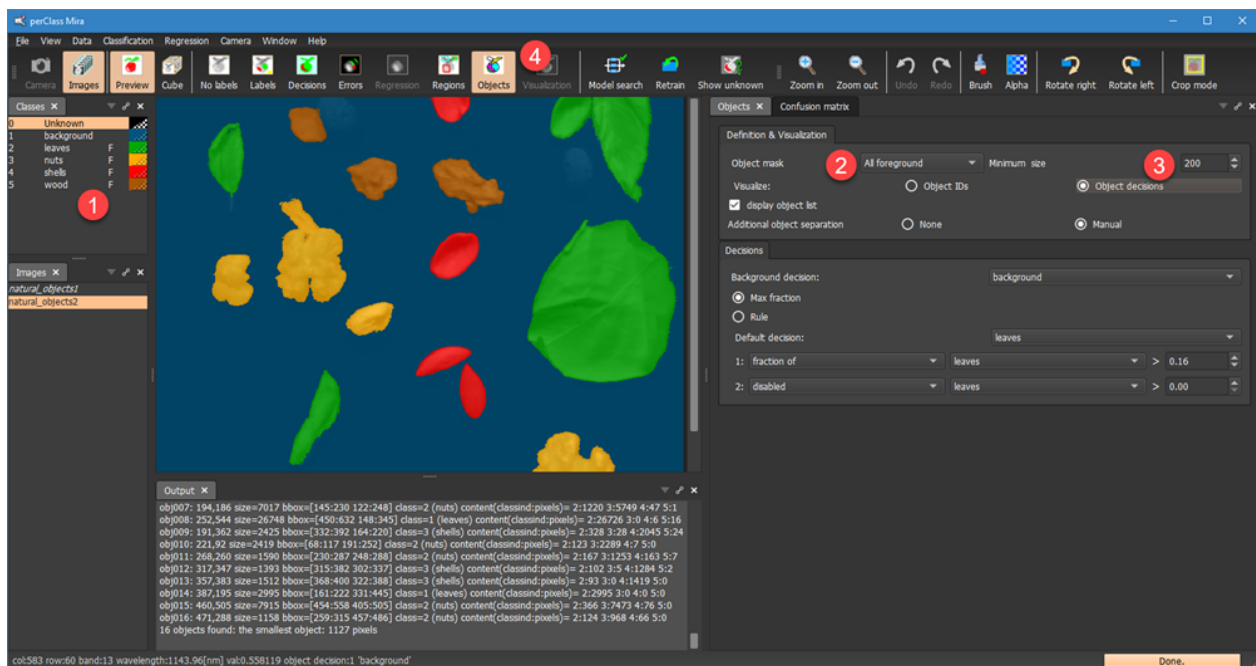
In the same way pixel confusion matrix allows us to understand performance of pixel classifier, perClass Mira provides object confusion matrix to characterize object classification performance.

In order to estimate object confusion matrix, we need

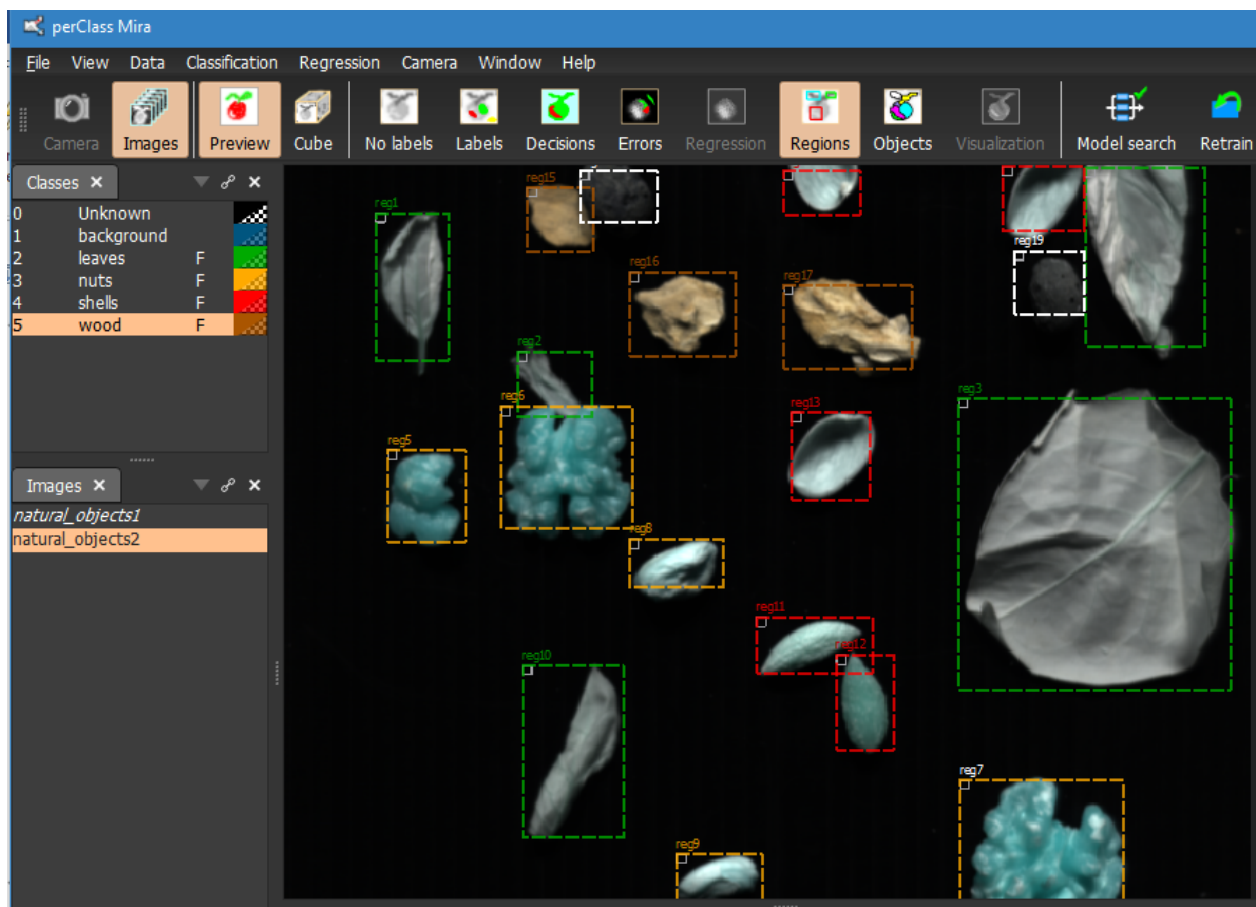
1. **object classifier** and
2. **object ground truth**, defined by image regions.

In our example, we built a pixel classifier and flagged several classes as foreground <sup>1</sup>. We use the *All foreground* mode <sup>2</sup> and minimum size of 200 pixels <sup>3</sup>. By clicking *Objects* toolbar button <sup>4</sup>, we perform object segmentation followed by object classifier:



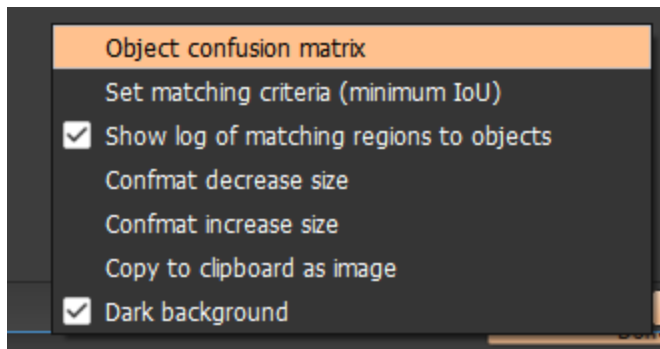


In order to define object ground truth, we use the *Regions* tool and [annotate individual objects](#) assigning them to their respective classes:

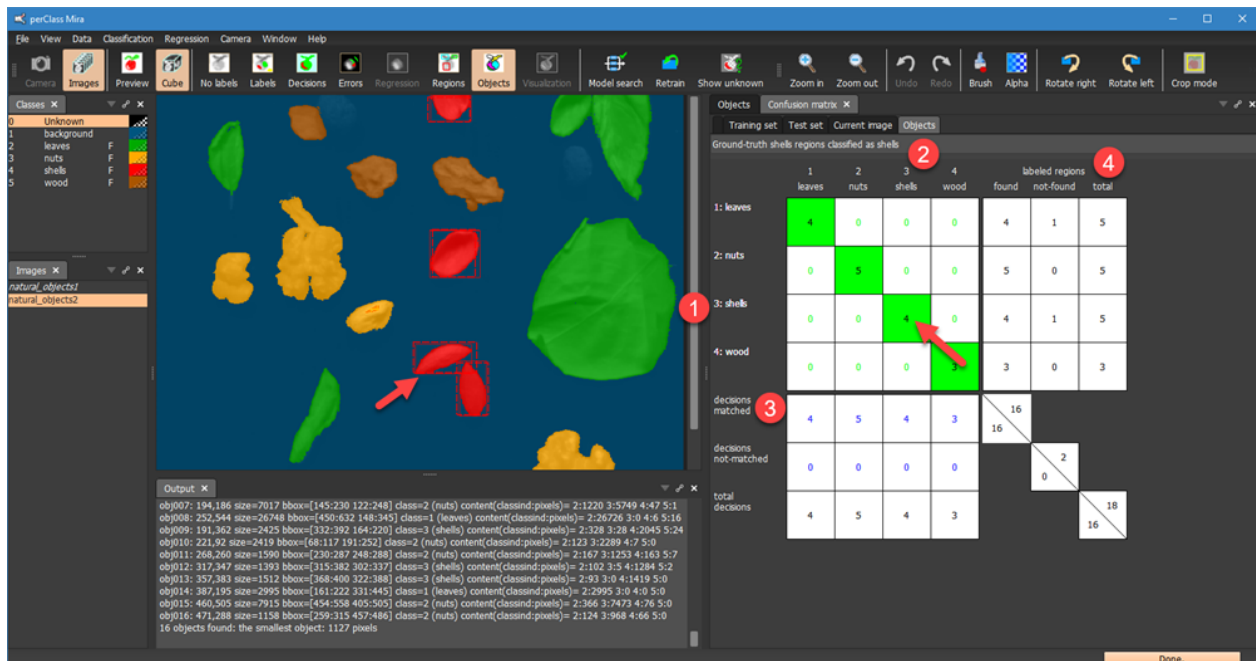


Now we can estimate the object confusion matrix by switching to the *Confusion matrix* panel, selecting *Objects*. We right-click to open context menu and select *Object confusion matrix*:





The object confusion matrix collects information from labeled regions (our ground-truth) in rows <sup>1</sup> and classifier decisions in columns <sup>2</sup>.



Note, that when only a single image is selected (like in our case), moving over the confusion matrix will highlight regions and object bounding boxes represented by the respective field. In our example, we can see both ground truth *shells* and *shell* decisions (object bounding boxes).

**TIP:** Note textual explanation available for each confusion matrix field in the top of the panel.

## Detailed information on object matching

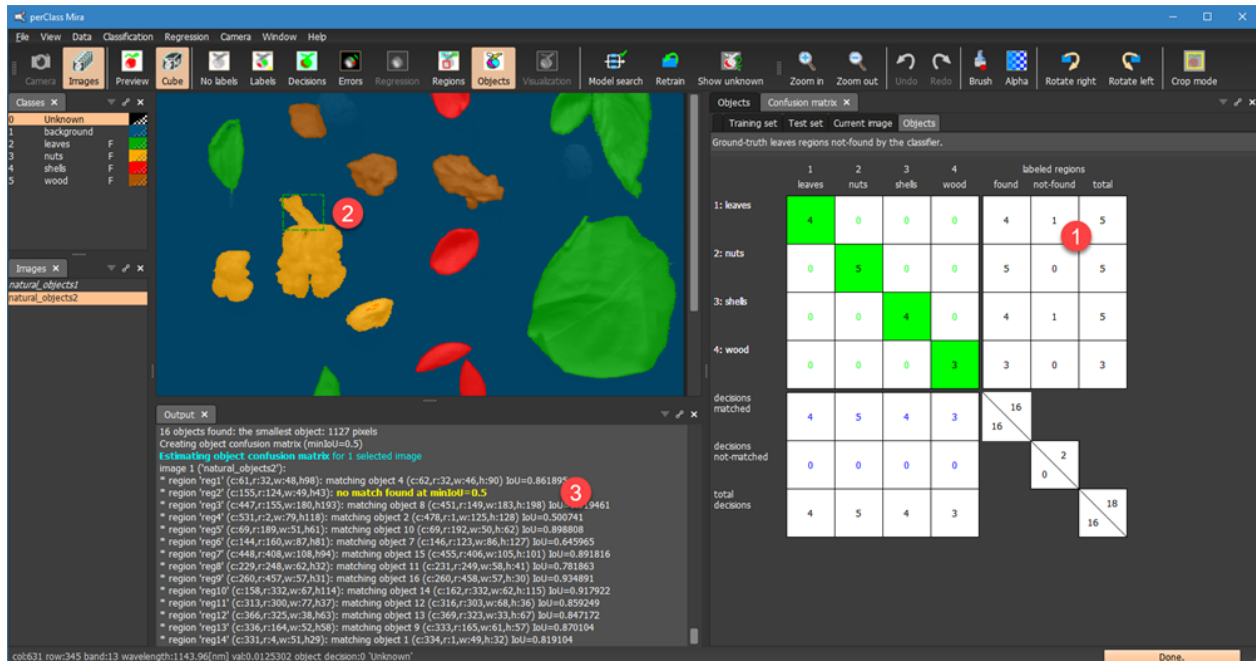
Apart of the square confusion matrix part summarizing all object decisions matched to the ground truth, the object confusion matrix also collect information on **all object decisions** <sup>3</sup> and **all labeled regions** <sup>4</sup>.

The decisions section <sup>3</sup> clarifies how many object decions are not matched to the ground truth. This is important to understand **false detections** that may pose significant burden in sorting applications. The

labeled regions section <sup>4</sup> provides extra insight on labeled regions that were not identified by the object classifier.

## Example of interactive inspection

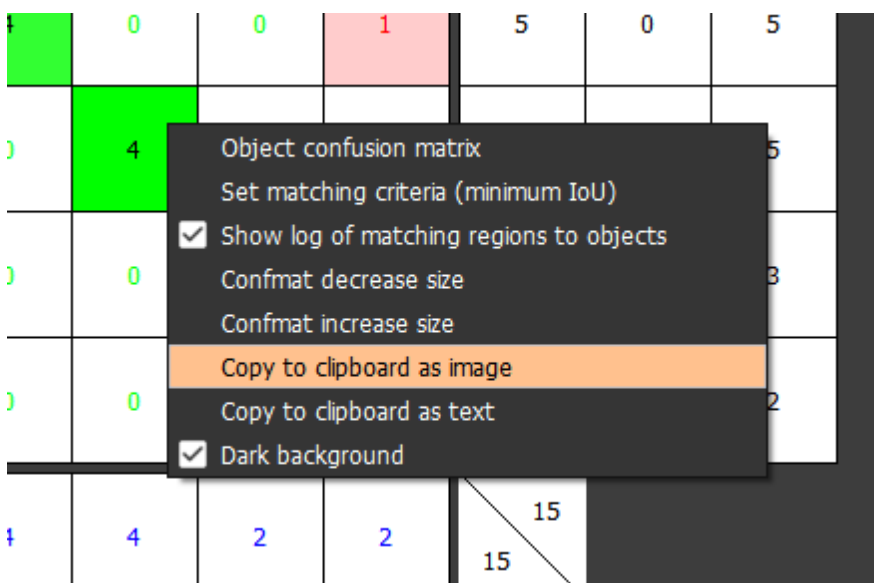
In the example below, we hover with the mouse pointer on the field <sup>1</sup>. This highlights single region from *leaves* class that was not found <sup>2</sup>. The *Output* window shows a log of all matches between the ground truth regions and object bounding boxes. We can see that the **reg2** region could not be matched as the default 0.5 level of the IoU (intersection over union) measure to any of the bounding boxes.



## Copying confusion matrix

Object confusion matrix can be copied as image or as text.

## Copying object confusion matrix as image



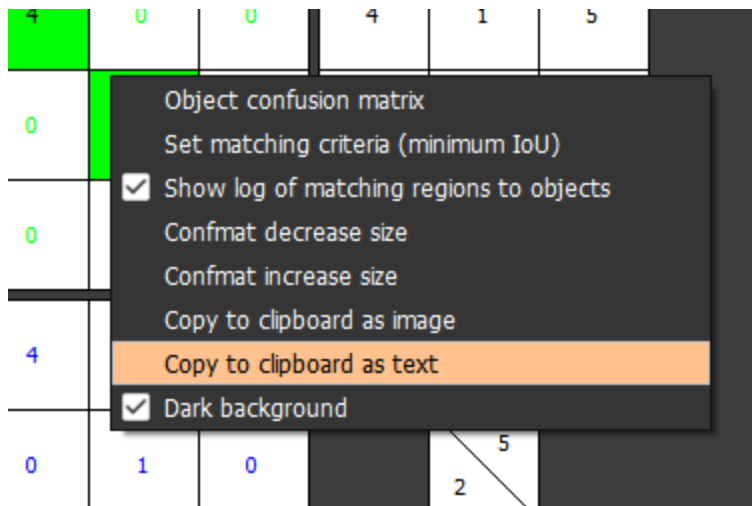
When copying as image for presentations, you may wish to disable dark background via context menu:

	1	2	3	4	5	labeled regions		
	leaves	nuts	shells	wood	Unknown	found	not-found	total
1: leaves	3	0	0	0	0	3	2	5
2: nuts	0	4	0	0	1	5	0	5
3: shells	0	0	4	0	0	4	1	5
4: wood	0	0	0	2	0	2	1	3
5: Unknown	0	0	0	0	1	1	1	2
decisions matched	3	4	4	2	2	15		
decisions not-matched	1	0	0	1	0		5	
total decisions	4	4	4	3	2		2	20
							17	

	1	2	3	4	5	labeled regions		
	leaves	nuts	shells	wood	Unknown	found	not-found	total
1: leaves	3	0	0	0	0	3	2	5
2: nuts	0	4	0	0	1	5	0	5
3: shells	0	0	4	0	0	4	1	5
4: wood	0	0	0	2	0	2	1	3
5: Unknown	0	0	0	0	1	1	1	2
decisions matched	3	4	4	2	2	15		
decisions not-matched	1	0	0	1	0		5	
total decisions	4	4	4	3	2		2	20
							17	

## Copying object confusion matrix as text

Copying confusion matrix content as text is useful, when you wish to perform further analysis of the results e.g. in Excel.



Book2 - Excel

File Home Insert Page Layout Formulas Data Review View Help Tell me what you want to do

Paste Clipboard Font Alignment Number Style

B1 Decisions

	A	B	C	D	E	F	G	H	I	J
1		Decisions					Labeled regions			
2		1:leaves	2:nuts	3:shells	4:wood	5:Unknown	Found	Not-found	Total	
3	1:leaves	3	0	0	0	0	3	2	5	
4	2:nuts	0	4	0	0	1	5	0	5	
5	3:shells	0	0	4	0	0	4	1	5	
6	4:wood	0	0	0	2	0	2	1	3	
7	5:Unknown	0	0	0	0	1	1	1	2	
8	Decisions matched	3	4	4	2	2	15/15			
9	Decisions not-match	1	0	0	1	0		5-Feb		
10	Total decisions	4	4	4	3	2			17/20	
11										
12										

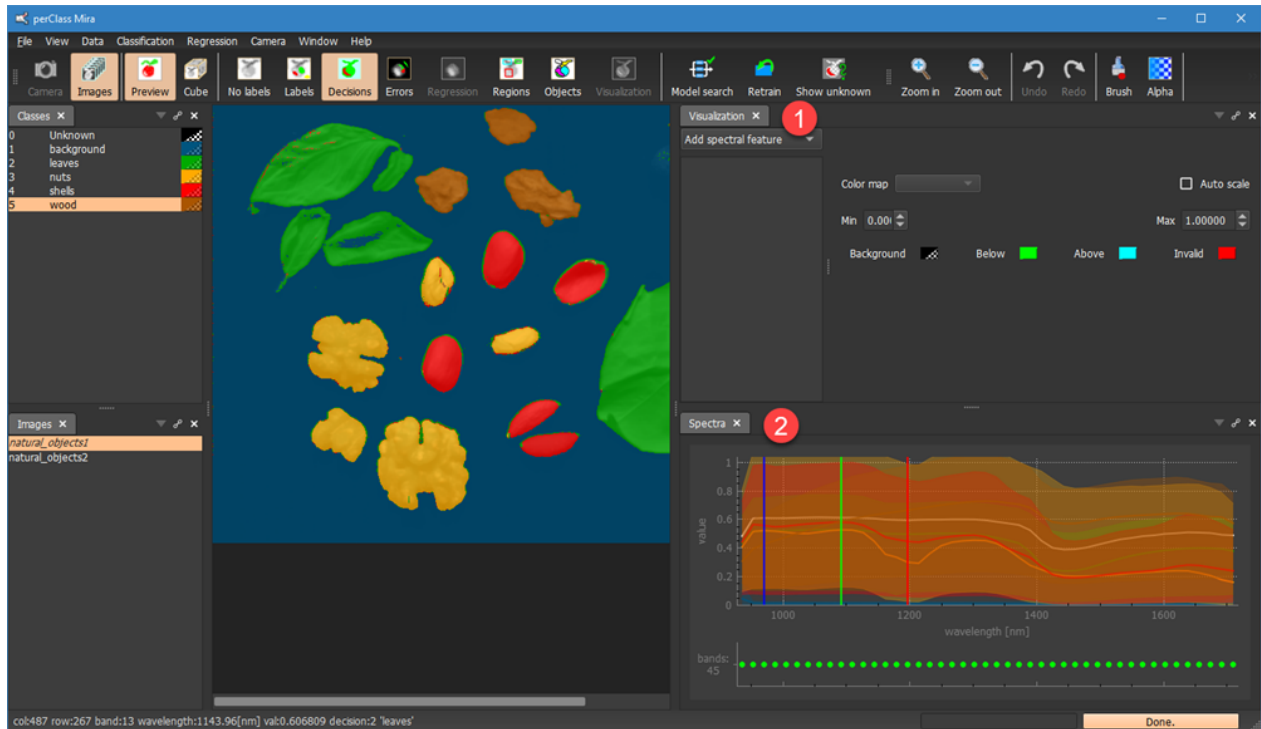
## Visualization (spectral indices)

perClass Mira provides an interactive visualization tool enabling the user to define custom spectral indices highlighting different aspects of spectral data. By a spectral index, we mean a quantity computed from spectral information at a pixel level.

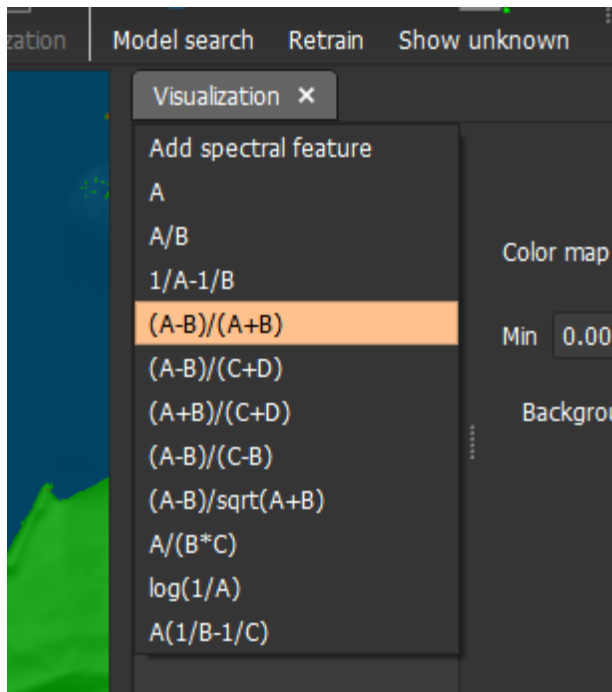
For example, one commonly-used spectral index is NDVI which, in broad terms, highlights a difference between visible and near-infrared reflectance. It is used in remote sensing in order to estimate vegetation coverage  $NDVI = (NIR - RED) / (NIR + RED)$ . In the NDVI equation, the RED and NIR terms corresponds to integrated reflectances in Red (visible) and near-infrared areas of the spectrum.

In perClass Mira, visualization or spectral feature extraction tools allow the user to define her own spectral indices either based on wavelength specification or interactively. While the former lets us work with spectral indices defined in literature, the later provides a powerful way of discovering hidden signal in spectral data by interactive experimentation with a direct visual feedback.

In order to use the interactive visualization, we organize perClass Mira screen in the following way: We position the *Visualization* panel <sup>1</sup> above the *Spectra* <sup>2</sup>.



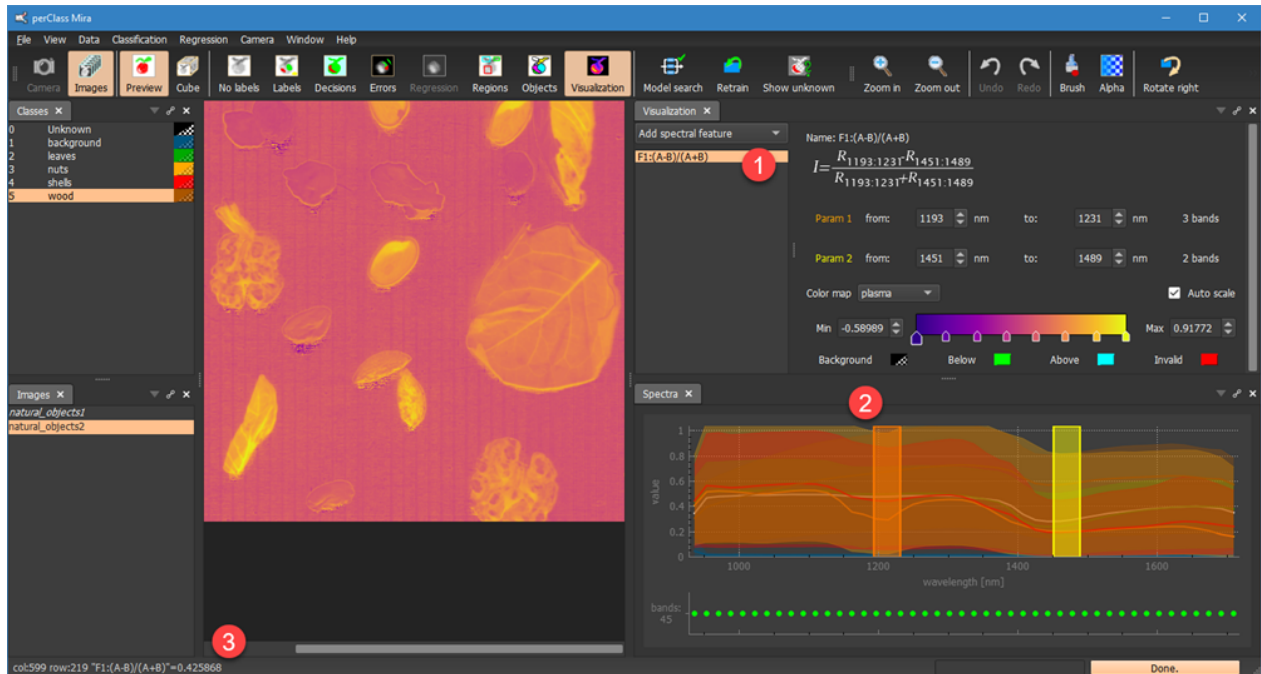
We may select the type of spectral feature (generic equation) in the combo box located in the *Visualization* panel.



We select the  $(A-B)/(A+B)$  equation covering the NDVI type of index discussed above. A new spectral

feature is created called "F1:(A-B)/(A+B)". We click on the new entry <sup>1</sup> in the list box. The spectral index is then computed for every pixel for the image using default definition of A and B spectral ranges. Pixel intensities are integrated (summed) in the specified A and B spectral ranges. For example,  $R_{\{1451:1489\}}$  (in LaTeX notation) means integrated reflectance between 1451 and 1489 nm. Each of the spectral feature

parameters (in our case A and B) are also highlighted by colored bars in the spectral plot <sup>2</sup>. Note that the equation is displayed in the *Visualization* panel using wavelength definition in nanometers. The parameters listed can be also changed directly by specifying the wavelengths. When hovering over the image, the floating point spectral feature value is displayed in the status bar <sup>3</sup>.



## Adjusting spectral features

Spectral features may be adjusted in two ways:

1. Interactively
2. By specifying the wavelength ranges

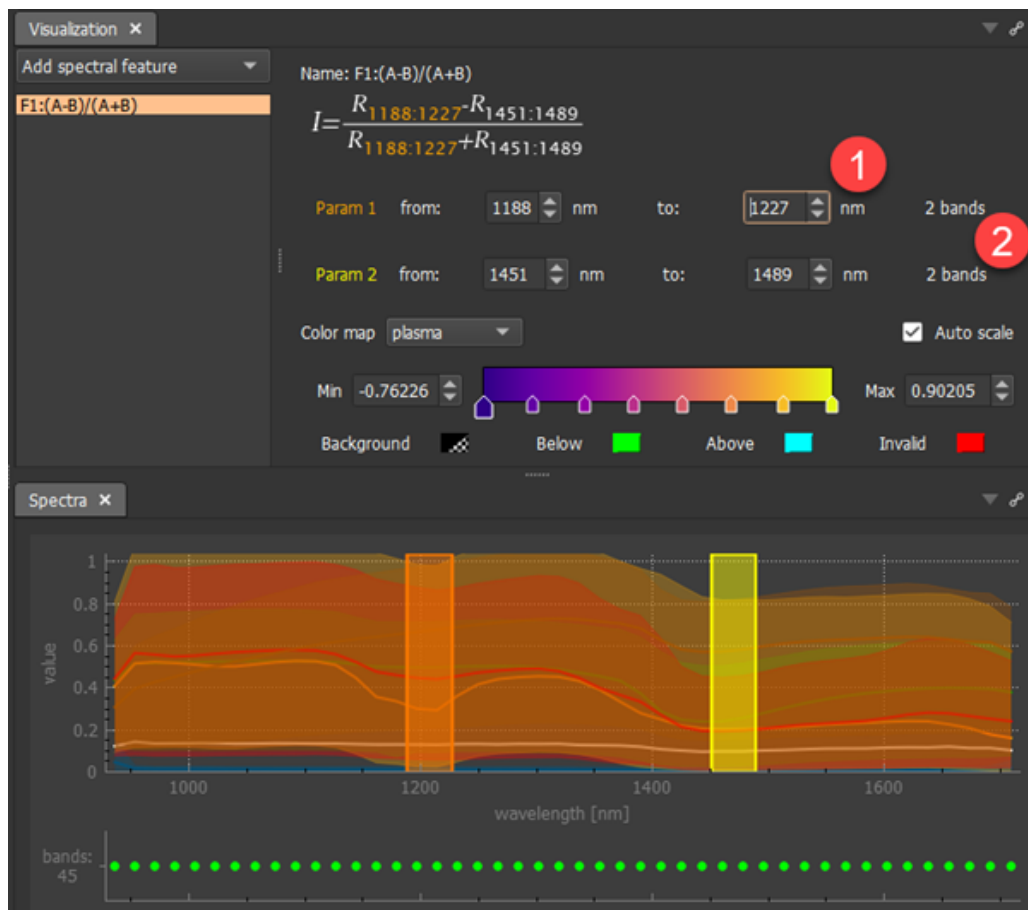
### Interactive spectral feature definition

We may adjust the current spectral feature, selected in the list box, by dragging the color bars, corresponding to its parameters, in the spectral plot. When dragging the bar we change its position. Alternatively, we may control its boundaries by dragging the bar borders.

**TIP:** Hold Ctrl to always drag the bar. This is useful if the bar is narrow and simple click-and-drag would result in change of the bar boundaries

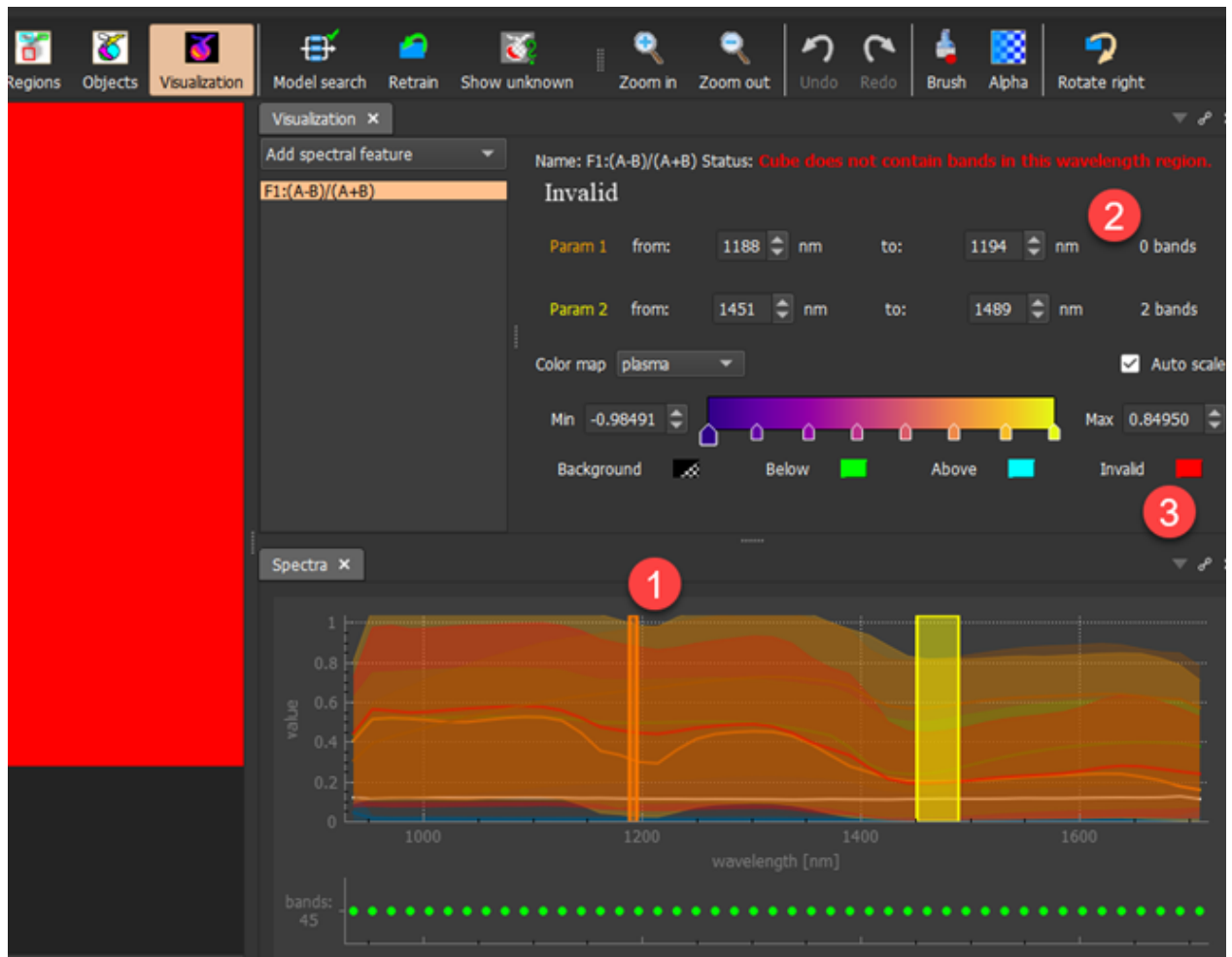
### Specification of wavelength ranges

When the spectral index definition is provided in literature by wavelength ranges, we may directly specify it for each of the parameters <sup>1</sup>.



Note, that the actual number of spectral bands is displayed next to each parameter <sup>2</sup>. In our example, each of the two parameters is computed by integrating content of two spectral bands.

In the following example, our wavelength range for parameter A <sup>1</sup> does not cover any existing spectral bands in our image. This is indicated by zero bands <sup>2</sup> and a warning in red. The entire image is then also displayed in red - the default "invalid" color <sup>3</sup>.



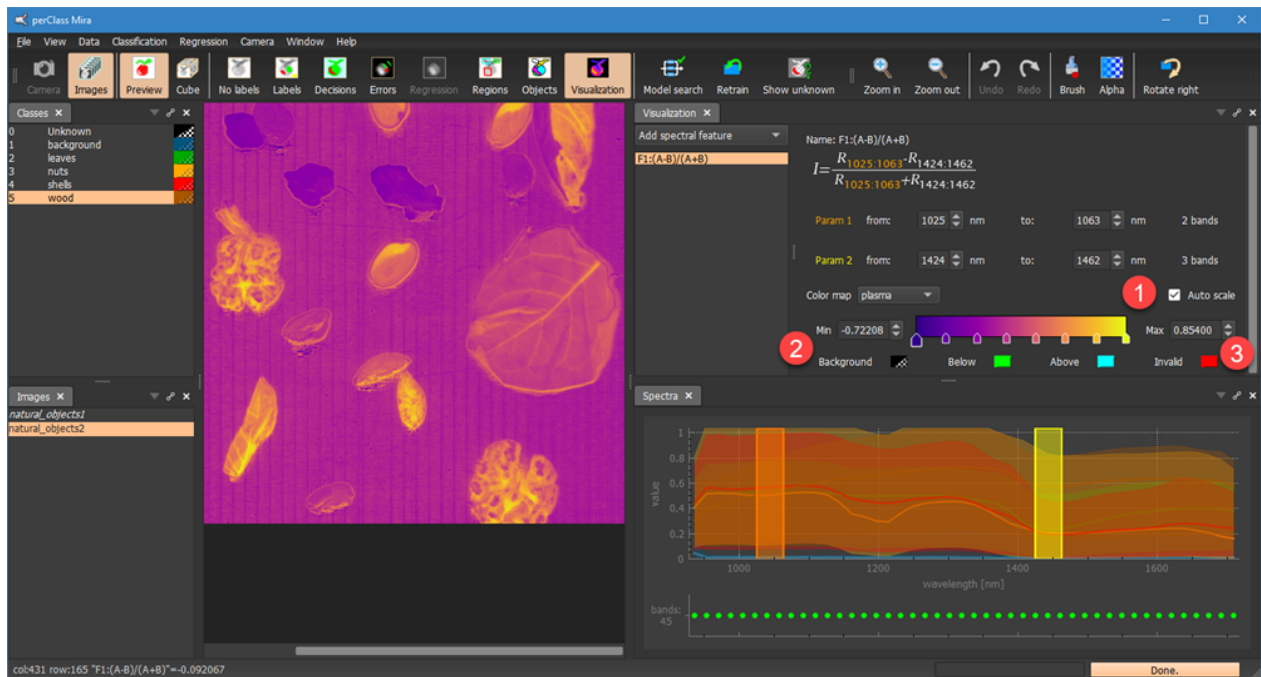
## Scaling spectral features

### **Auto-scaling**

By default, visualization of spectral features is auto-scaled. This is indicated by the *Auto scale* checkbox

1



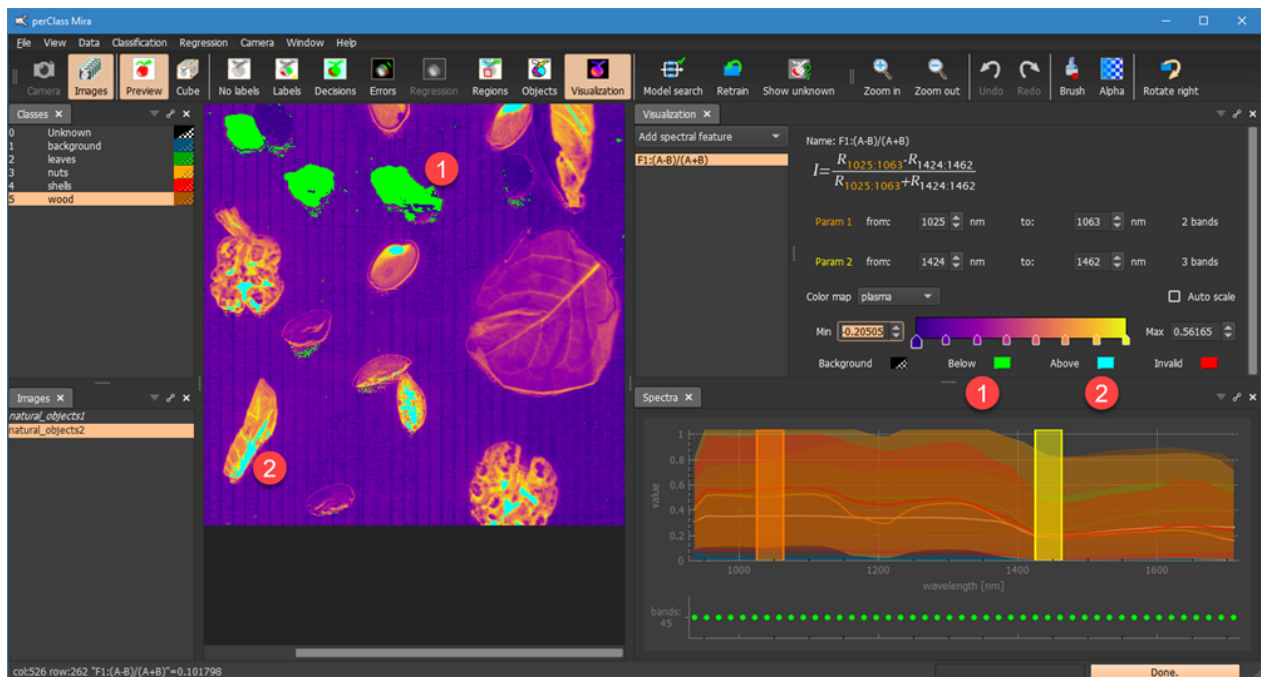


When switching between different images, the minimum <sup>2</sup> and maximum <sup>3</sup> value is adjusted based on the current image. While this is useful for a quick understanding of the data, we may want to fix the visualization to a single range. This makes the visualization comparable for all images. We may do just that either by disabling the *Auto scale* checkbox or by adjusting the *Min* and *Max* edit fields.

### Manual scaling

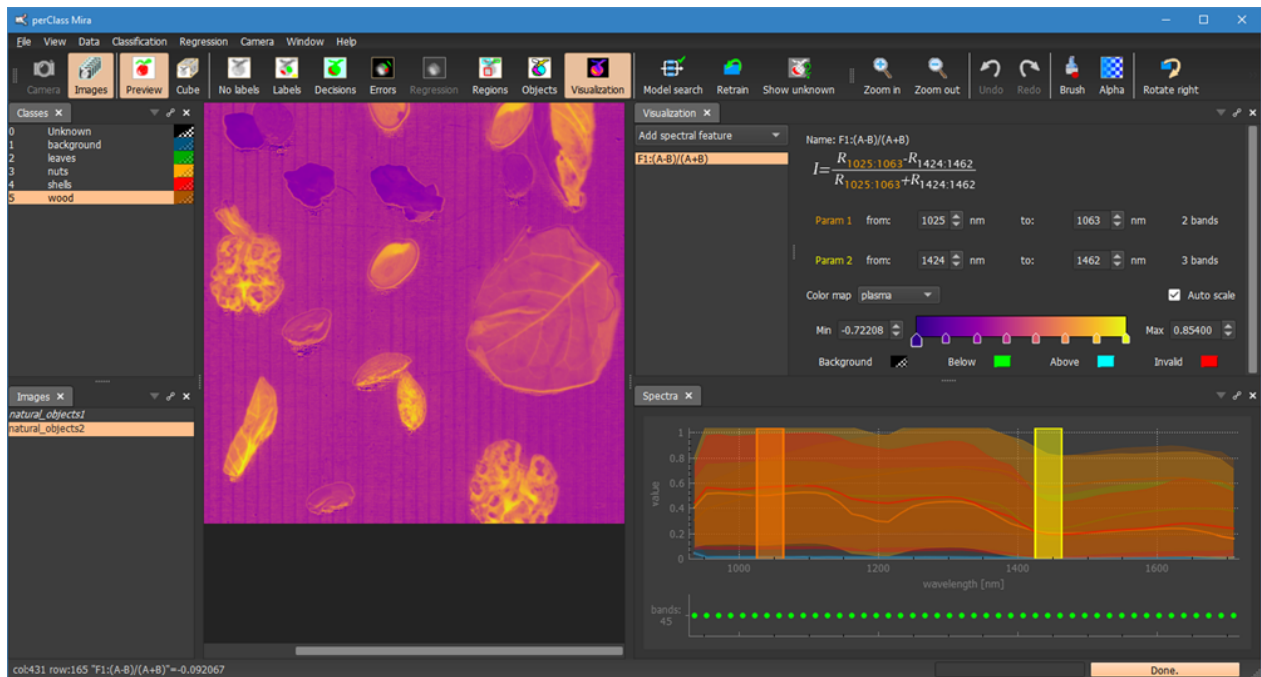
When adjusting minimum and maximum range values manually, we may notice that some pixels are

displayed in green <sup>1</sup> or cyan <sup>2</sup>. These correspond to pixels below or above the current visualization range. You may adjust the colors by the respective color swatches.



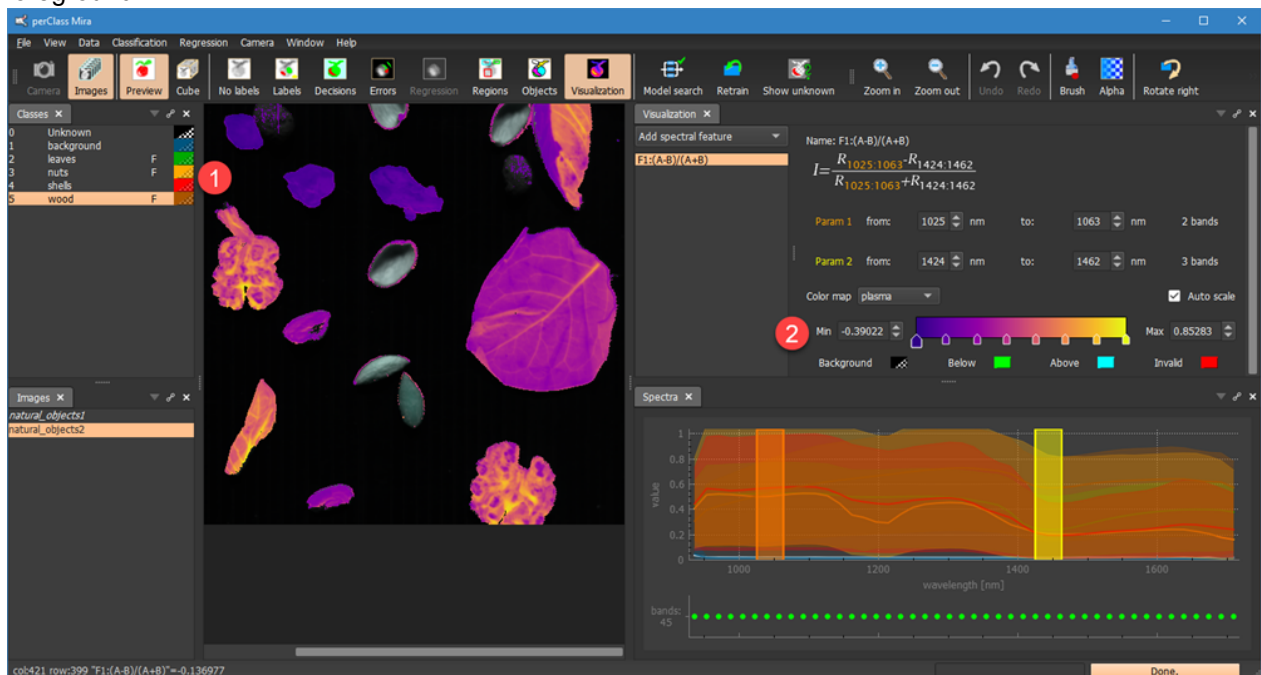
### Applying feature extraction to foreground

By default, spectral feature extraction is applied to all image pixels:



If a pixel classifier is defined, we may focus the analysis only to specific classes by selecting these as

foreground <sup>1</sup>:

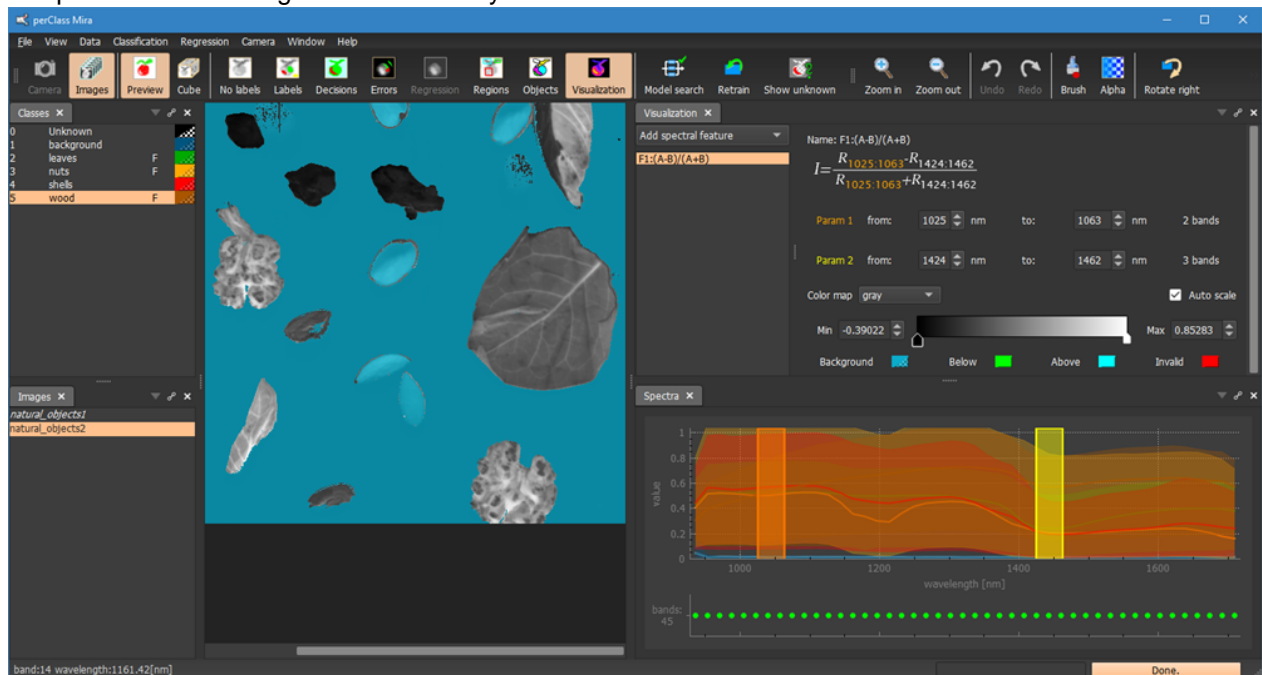


Note, that, when applying the visualization only to the foreground, the auto-scaling adjusts the min and max boundaries based on the foreground pixels.

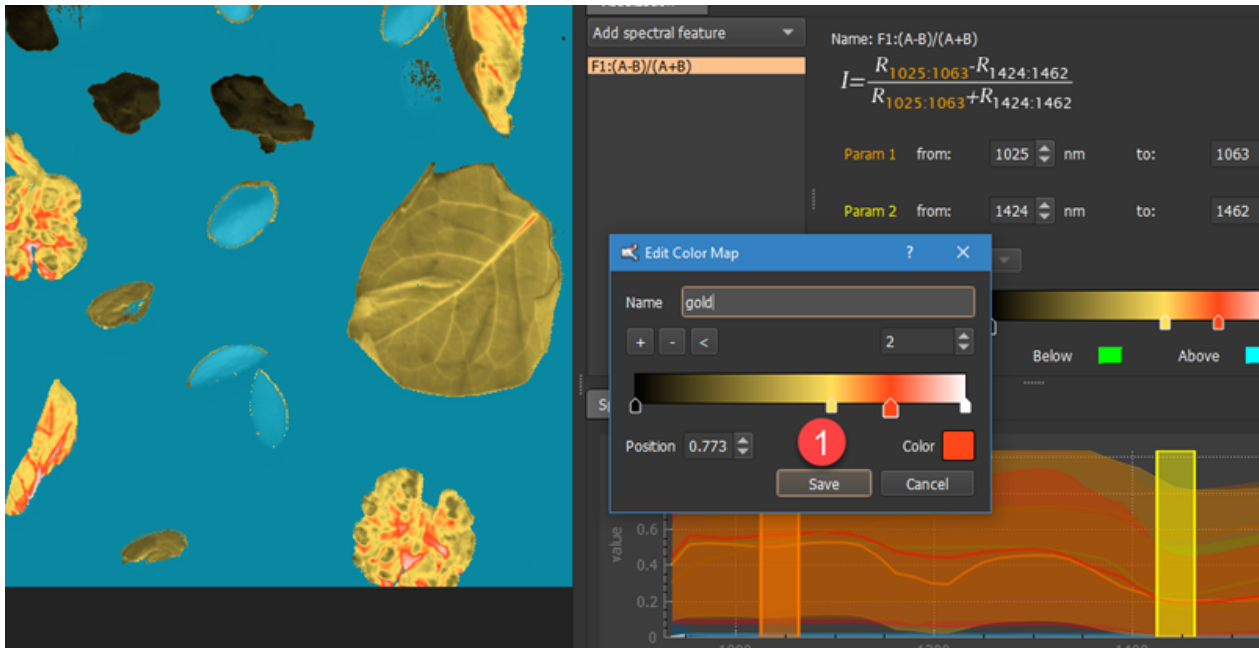
## Colormaps

The color map of spectral feature visualization can be selected using the combo box <sup>1</sup>

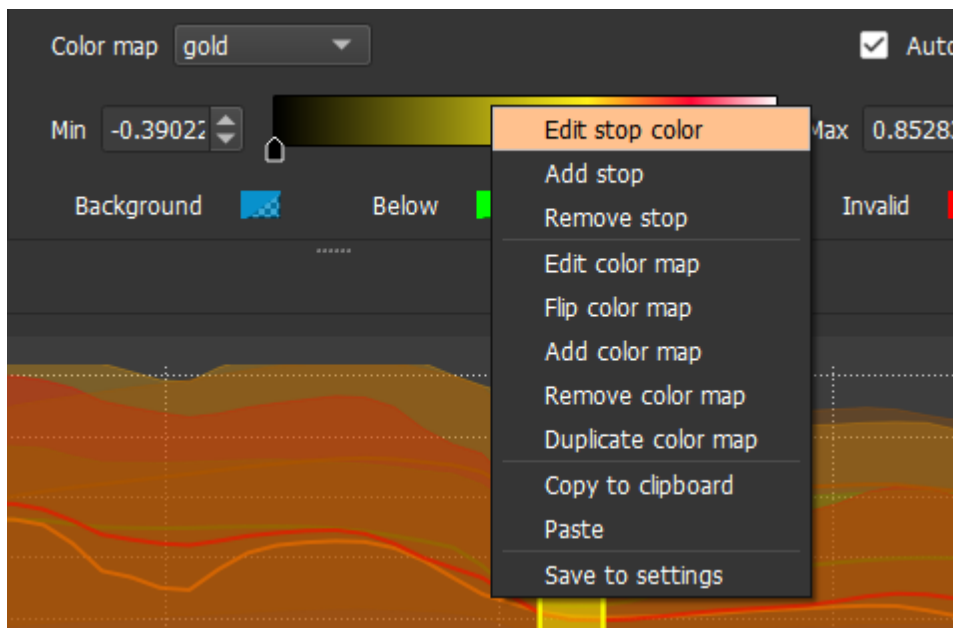
In this example, we set the gray level color map and make the background distinct. This allows us to spot fine patterns in the foreground more easily.



Example of custom color map definition: Select *New* in *Color map* combo box. Define the colormap steps and colors interactively <sup>1</sup> and name the color map.



The new color map will be saved in mira.ini file. When clicking on the color map widget, a context menu provides number of additional options:



perClass Mira represents color maps in a simple string format that can be copied, edited by the user and pasted back in the application.

The mira.ini file will contain our new color map definition as a text string.

```
[colormap]
gold="gold;0 0,0,0;0.625 255,240,24;0.829 255,7,52;1 255,255,255;"
```

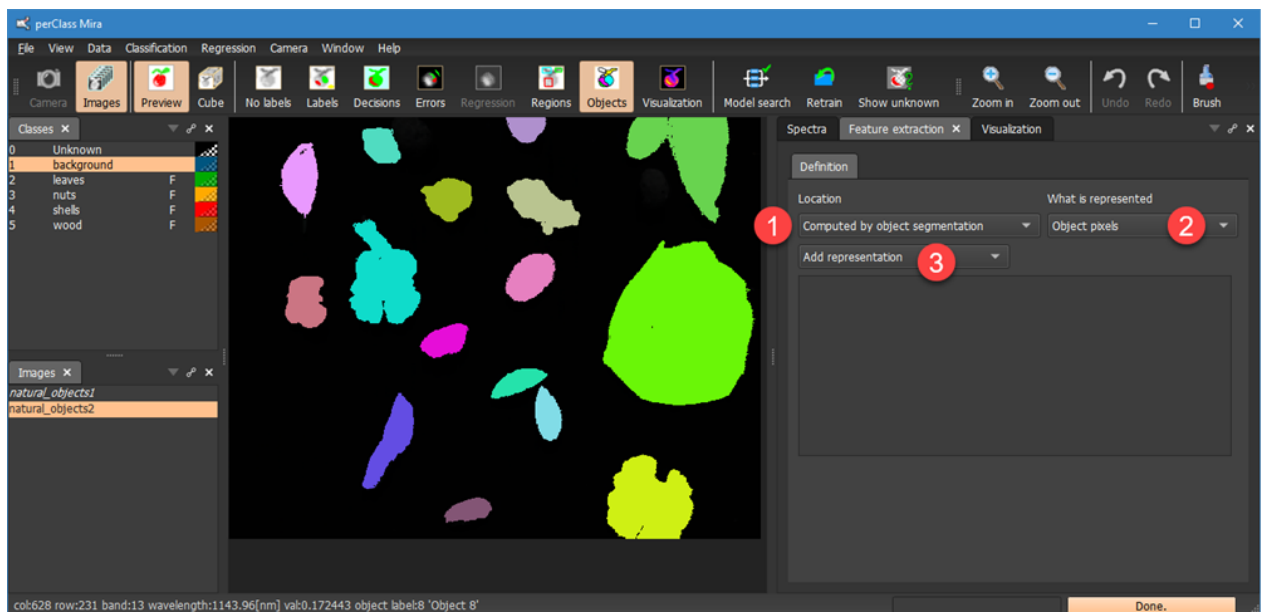
The content in double quotes specifies fully the color map. It is a list separating fields by semicolon. First the color map name is listed. Then, each of the color map steps is defined. Each step contains a 0.0 to 1.0 relative position followed by RGB definition of the color.

## Feature extraction (exporting)

perClass Mira provides numerous ways how to extract and export information from a single or multiple images. The use case is to define classifier, segment objects or specify regions of interest and export user-defined features to external file (Excel .xls or XML). This data is the used for custom data analysis or further research.

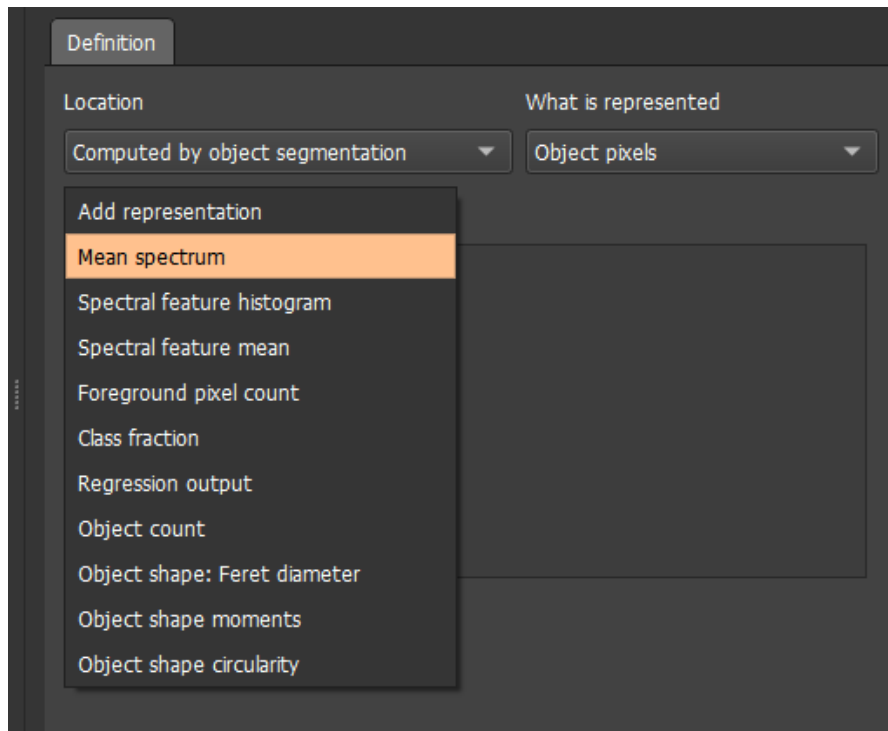
Let us walk through a basic example using *Feature extraction* panel. In order to extract data, we need to specify

- Where the data is extracted from <sup>1</sup>
- What pixels are included in the extraction <sup>2</sup>
- What is being extracted <sup>3</sup>

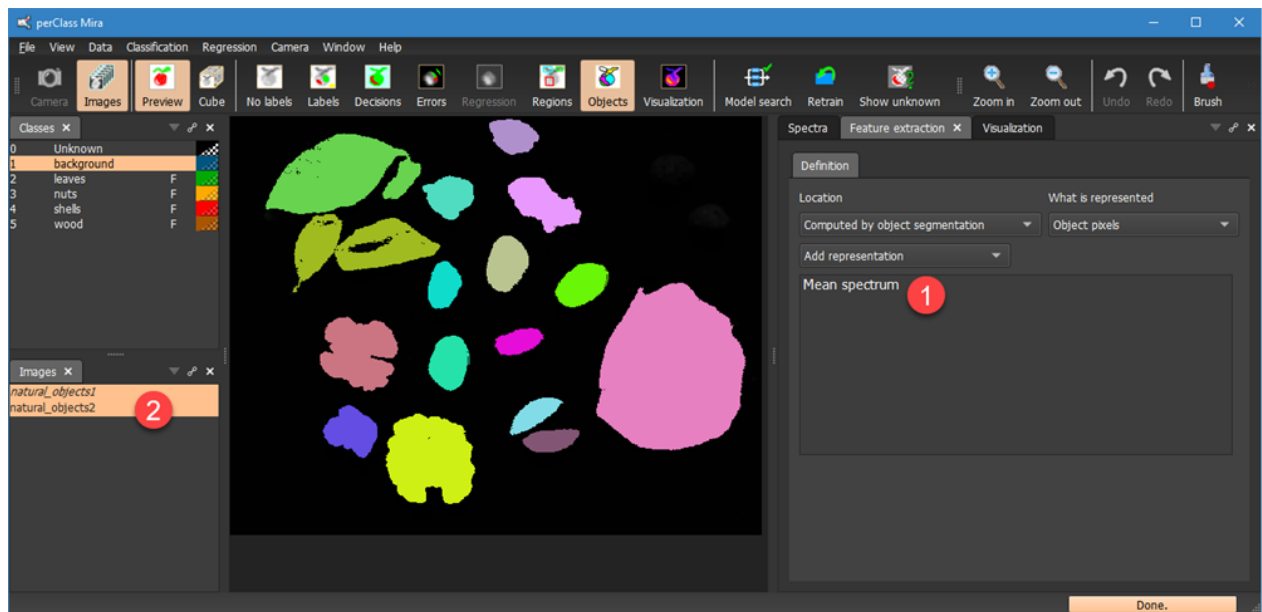


In the first example, we want to extract mean spectra from objects. Therefore, we select *Mean spectrum* from the *Add representation* combo box.

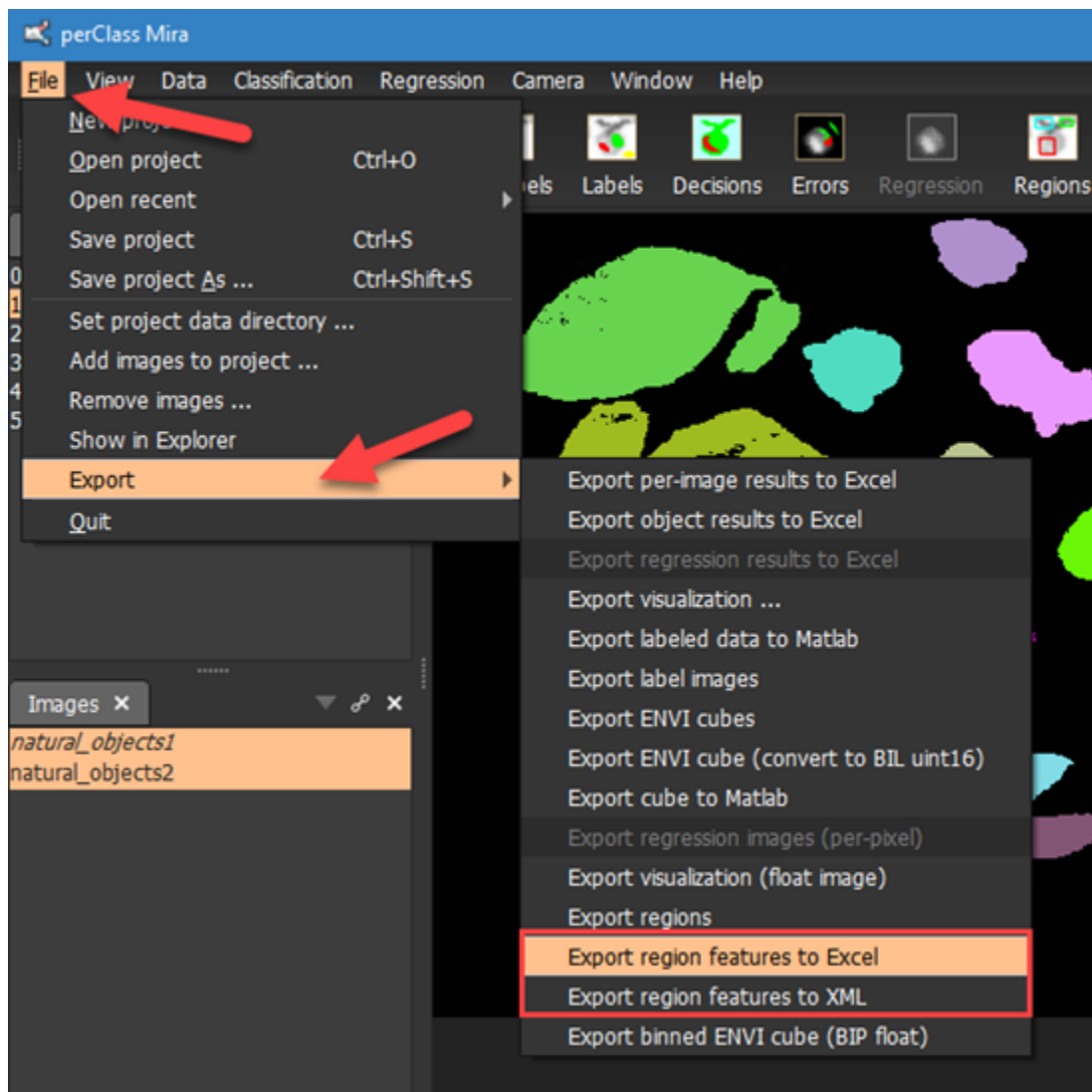




You may select multiple representations <sup>1</sup>



Select one of more images in *Images* list <sup>2</sup> and then *File* menu / *Export* and *Export region features to Excel*. Note you may also export the same data into XML. That option is more convenient if you wish to programmatically post-process data analysis.



You will be asked for a name of a file to save. By default, perClass Mira exports to XLSX format. This allows for more than 256 columns which is useful if we're exporting a lot of features per object, for example mean spectra. If you prefer the legacy XLS format, you may choose it in the export dialog.

In the screenshot below we can see the structure of the exported data:



Image name	Object name	Bounding box Column	Row	Width	Height	Object decision	Content present	pixels	band index	Mean spectrum 1	Mean spectrum 2	Mean spectrum 3	Mean spectrum 4	Mean spectrum 5	Mean spectrum 6	Mean spectrum 7
natural_objects2	obj1	334	1	49	32	shells	TRUE	1127		0.418884	0.61078	0.620182	0.613909	0.616726	0.621849	0.625797
natural_objects2	obj2	478	1	125	128	leaves	TRUE	8360		0.359187	0.461895	0.4688	0.469906	0.473073	0.476322	0.478664
natural_objects2	obj3	161	13	47	43	wood	TRUE	1537		0.188756	0.258818	0.280343	0.29866	0.316507	0.334265	0.351406
natural_objects2	obj4	62	32	46	90	leaves	TRUE	2739		0.257662	0.363636	0.370373	0.372552	0.374962	0.37812	0.379556
natural_objects2	obj5	232	73	62	53	wood	TRUE	2199		0.252241	0.328854	0.353295	0.373342	0.392379	0.410843	0.428108
natural_objects2	obj6	336	75	88	67	wood	TRUE	3302		0.247954	0.328802	0.354568	0.375289	0.395374	0.415124	0.434162
natural_objects2	obj7	146	123	86	127	nuts	TRUE	7017		0.312839	0.404573	0.409493	0.406818	0.401605	0.395843	0.395225
natural_objects2	obj8	451	149	183	198	leaves	TRUE	26748		0.368264	0.442807	0.447981	0.449507	0.4514	0.453307	0.45472
natural_objects2	obj9	333	165	61	57	shells	TRUE	2425		0.352831	0.482109	0.485118	0.47924	0.480726	0.484611	0.487408
natural_objects2	obj10	69	192	50	62	nuts	TRUE	2419		0.294806	0.395407	0.40122	0.398492	0.39295	0.387056	0.386051
natural_objects2	obj11	231	249	58	41	nuts	TRUE	1590		0.365468	0.508502	0.518456	0.516615	0.514616	0.512984	0.51402
natural_objects2	obj12	316	303	68	36	shells	TRUE	1393		0.386104	0.53286	0.52853	0.522129	0.526111	0.533979	0.541446
natural_objects2	obj13	369	323	33	67	shells	TRUE	1512		0.240948	0.379026	0.371125	0.364197	0.369597	0.379022	0.38753
natural_objects2	obj14	162	332	62	115	leaves	TRUE	2995		0.245307	0.37675	0.381109	0.3837	0.386949	0.390507	0.393138
natural_objects2	obj15	455	406	105	101	nuts	TRUE	7915		0.370605	0.49079	0.49658	0.491898	0.483799	0.475624	0.474846
natural_objects2	obj16	260	458	57	30	nuts	TRUE	1158		0.372948	0.527443	0.537544	0.53582	0.532799	0.529806	0.530455
natural_objects1	obj1	313	4	60	43	wood	TRUE	1837		0.19693	0.275119	0.298537	0.317987	0.336933	0.355497	0.373527
natural_objects1	obj2	44	21	188	97	leaves	TRUE	11140		0.35793	0.430531	0.435068	0.43767	0.440422	0.443469	0.445498
natural_objects1	obj3	233	72	62	53	wood	TRUE	2231		0.264209	0.335987	0.360361	0.380526	0.399792	0.418379	0.435768
natural_objects1	obj4	336	74	88	67	wood	TRUE	3356		0.260755	0.338152	0.364111	0.385229	0.405561	0.425548	0.444614
natural_objects1	obj5	77	118	177	94	leaves	TRUE	7627		0.265864	0.332569	0.334761	0.336302	0.338404	0.340751	0.342429
natural_objects1	obj6	310	144	50	68	shells	TRUE	2513		0.316587	0.435052	0.437351	0.431732	0.433359	0.436658	0.439768
natural_objects1	obj7	239	175	41	58	nuts	TRUE	1708		0.386804	0.570502	0.584084	0.583087	0.581207	0.579673	0.581098
natural_objects1	obj8	392	177	64	54	shells	TRUE	2379		0.372631	0.495365	0.49744	0.49143	0.493097	0.497032	0.499977
natural_objects1	obj9	442	201	180	201	leaves	TRUE	27240		0.376576	0.442941	0.446655	0.447628	0.449241	0.450948	0.452186
natural_objects1	obj10	110	242	93	89	nuts	TRUE	5960		0.332146	0.415901	0.419814	0.416104	0.409281	0.402252	0.401151
natural_objects1	obj11	320	255	58	33	nuts	TRUE	1328		0.488548	0.658507	0.671276	0.668415	0.665356	0.662282	0.663227
natural_objects1	obj12	240	264	49	67	shells	TRUE	2411		0.360104	0.476716	0.477188	0.470221	0.471511	0.475606	0.47879
natural_objects1	obj13	338	339	64	45	shells	TRUE	1451		0.382345	0.534721	0.529953	0.522595	0.527041	0.535618	0.543633
natural_objects1	obj14	113	348	66	63	nuts	TRUE	2651		0.368257	0.480393	0.485677	0.487028	0.475577	0.468517	0.467581

Objects of each selected image are described by rows. For each object, we can see the scan name followed by an object name. When exporting object segmentation, the object name is automatically assigned in the segmentation procedure. When exporting content of user-defined regions, the region name

is used. This can be [user-assigned](#). For each object, its bounding box and per-object decision

is provided. The *Content present* column shows whether there is content represented in this object/region and if so, how many pixels. When exporting objects, the content is always present. When exporting regions, this may not be the case.

Finally, the section contains the exported data. In our case, the columns correspond to individual wavelengths of the mean spectra extracted for each object.

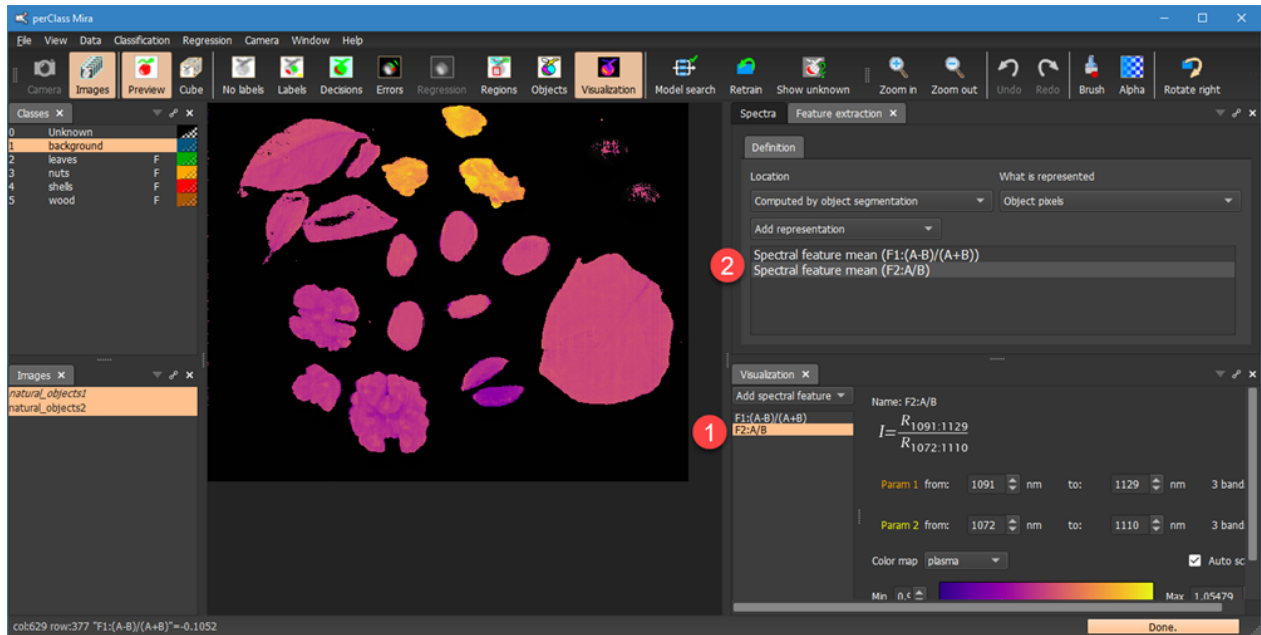
## Extracting multiple features

We may specify multiple features to be extracted from each object/region.

Say, we wish to extract two spectral indices and some shape representation of each object. We have

[defined two spectral indices](#). We need to select a specific index and then choose the desired

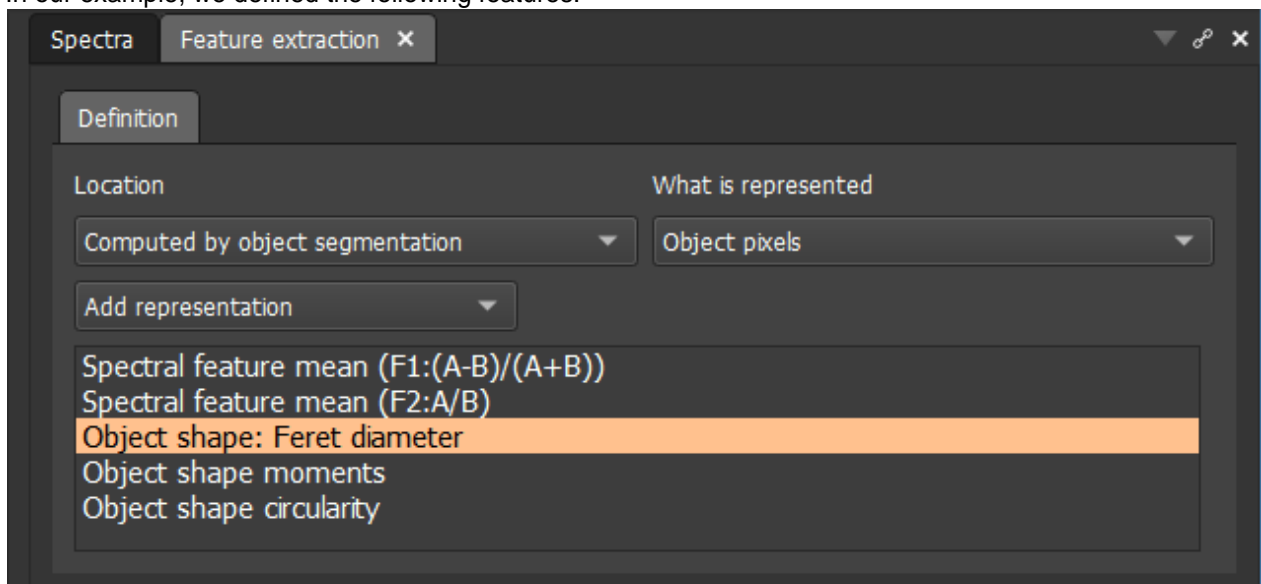
representation in the *Add representation* combo to include it in the list.



### Available feature types

- **Mean spectrum** - mean spectrum computed using all pixels specified
- **Spectral feature histogram** for the selected spectral feature. The min and max boundaries, defined in the *Visualization* panel, are used and split into 20 bins.
- **Spectral feature mean** for the selected spectral feature
- **Foreground pixel count**
- **Class fraction** for the class selected in the *Class list*
- **Regression output** for the selected regression variable
- **Object count** within the region
- **Object shape: Feret diameter** - shape representation providing minimum and maximum caliper distance for the object mask
- **Object shape moments** - a set of 7 Hu shape moment invariants and an object eigenvalue ratio
- **Object shape circularity** describes how far from a circle is certain object shape. Three features are provided, namely Circularity, the Area/Perimeter ratio and the Perimeter.

In our example, we defined the following features:



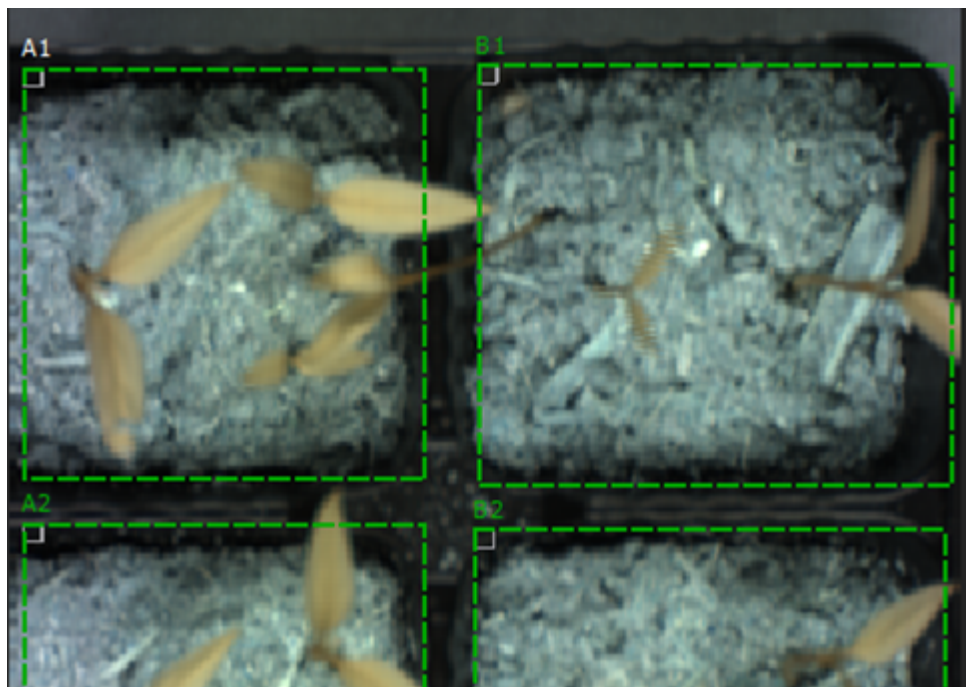
Below screenshot of the exported Excel file with indicated 5 feature groups.

Mean of spectral feature (F1:(A-B)/(A+B))

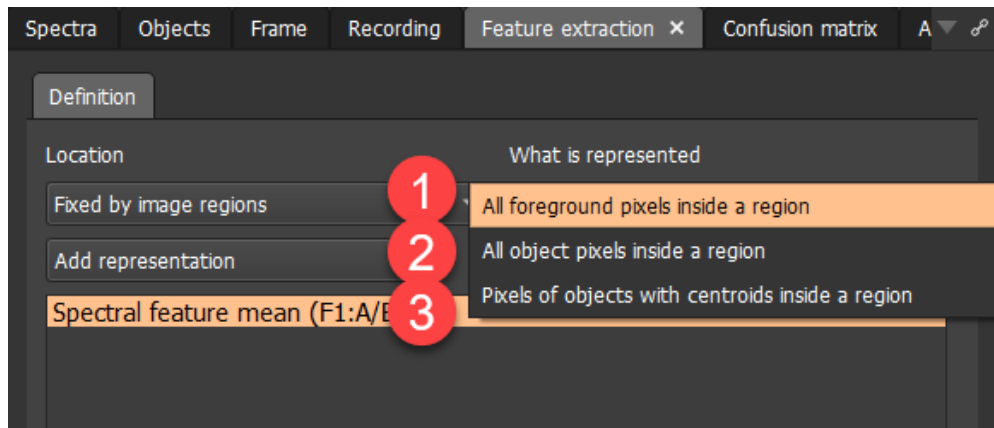
	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	AA	AB	AC	AD
1																						
2																						
3																						
4																						
5																						
6																						
7																						
8																						
9																						
10																						
11																						
12																						
13																						
14																						
15																						
16																						
17																						
18																						
19																						
20																						
21																						

## Extracting from region grid

In some applications, we may wish to extract data from user-defined regions. For example, in plant phenotyping, each plant seedling may be defined by a region. In perClass Mira, we can use the [region annotation](#) to drive feature extraction. The advantage of feature extraction from regions is, that we may detect absence of data in a cell (for example, when the seed did not germinate).



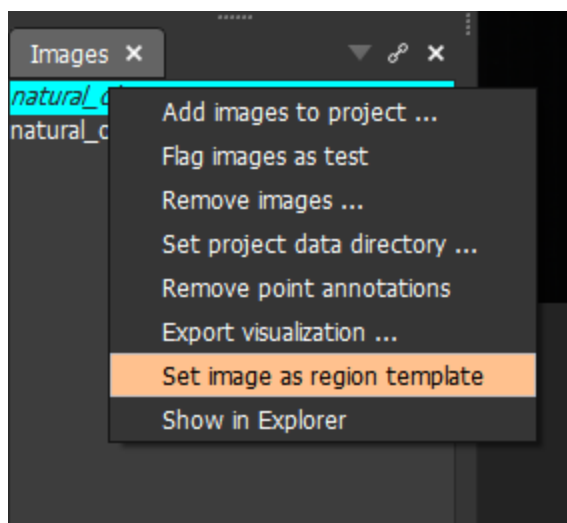
When using regions for feature extraction, we have few options how to define what pixels are included in the processing:



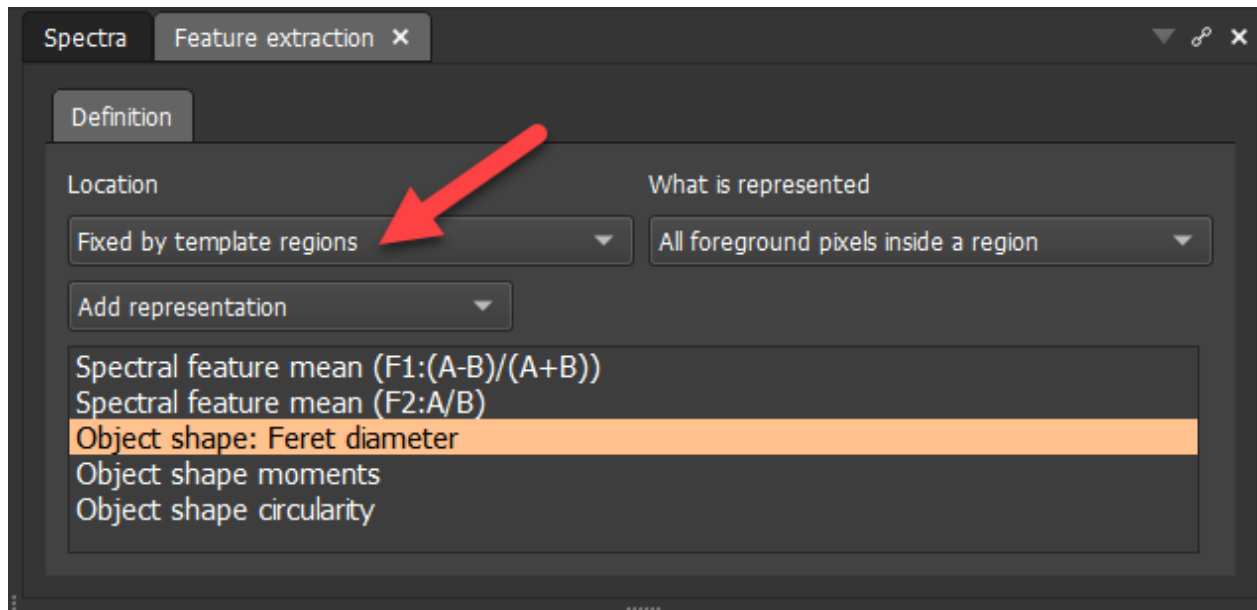
The option <sup>1</sup> simply includes all foreground pixels within the region. The option <sup>2</sup> only considers the object pixels. Therefore, pixels of small objects (with size smaller than the defined minimum object size) are not included. Finally, the option <sup>3</sup> includes only pixels of objects with centroids within the region. This excludes e.g. a leaf extending from a neighboring germination cell into our region.

### Defining region extraction template

If the regular region grid is applicable to multiple scans, we may set one of the images with the desired region definition as a template. Select the *Set image as region template* in the *Images* context menu. The image is then marked with the light blue/cyan color.



The *Location* in *Feature extraction* panel can then be *Fixed by template regions*.



The same set of regions from the template image is then used when exporting data. Only one template can be selected in a project.

## Exporting into XML

When exporting into XML using *File / Export / Export region into XML* menu command, we obtain an XML file with the following structure:

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <object_representation xmlns="http://perclass.com/mira" version="perClass Mira 4.2 (28-feb-2023)">
3    <location>Computed by object segmentation</location>
4    <what_is_represented>All object pixels</what_is_represented>
5    <representations>
6      <representation index="1" name="Spectral feature mean (F1:(A-B)/(A+B))" type="spectral_feature_mean"/>
7      <representation index="2" name="Spectral feature mean (F2:A/B)" type="spectral_feature_mean"/>
8      <representation index="3" name="Object shape: Feret diameter"/>
9      <representation index="4" name="Object shape moments"/>
10     <representation index="5" name="Object shape circularity"/>
11   </representations>
12   <images count="2">
13     <image index="2" name="natural_objects2">
14       <objects count="16">
15         <object index="1" type="computed" name="obj1" bbox="334 1 49 32">
16           <representations>
17             <representation index="1" name="Spectral feature mean (F1:(A-B)/(A+B))" present="true" pixels="1127">0.204476</representation>
18             <representation index="2" name="Spectral feature mean (F2:A/B)" present="true" pixels="1127">0.997617</representation>
19             <representation index="3" name="Object shape: Feret diameter"/>
20             <representation index="4" name="Object shape moments"/>
21             <representation index="5" name="Object shape circularity"/>
22           </representations>
23         </object>
24         <object index="2" type="computed" name="obj2" bbox="478 1 125 128">
25           <representations>
26             <representation index="1" name="Spectral feature mean (F1:(A-B)/(A+B))" present="true" pixels="8360">0.217082</representation>
27             <representation index="2" name="Spectral feature mean (F2:A/B)" present="true" pixels="8360">0.996487</representation>
28             <representation index="3" name="Object shape: Feret diameter"/>
29             <representation index="4" name="Object shape moments"/>
30             <representation index="5" name="Object shape circularity"/>
31           </representations>
32         </object>
33         <object index="3" type="computed" name="obj3" bbox="161 13 47 43">
34           <representations>
35             <representation index="1" name="Spectral feature mean (F1:(A-B)/(A+B))" present="true" pixels="1537">-0.304859</representation>
36             <representation index="2" name="Spectral feature mean (F2:A/B)" present="true" pixels="1537">1.03535</representation>

```

The section <sup>1</sup> lists the features (representations) used. Then, for each image <sup>2</sup>, each object <sup>3</sup> is described with extracted data for each representation <sup>4</sup>.

## Regression

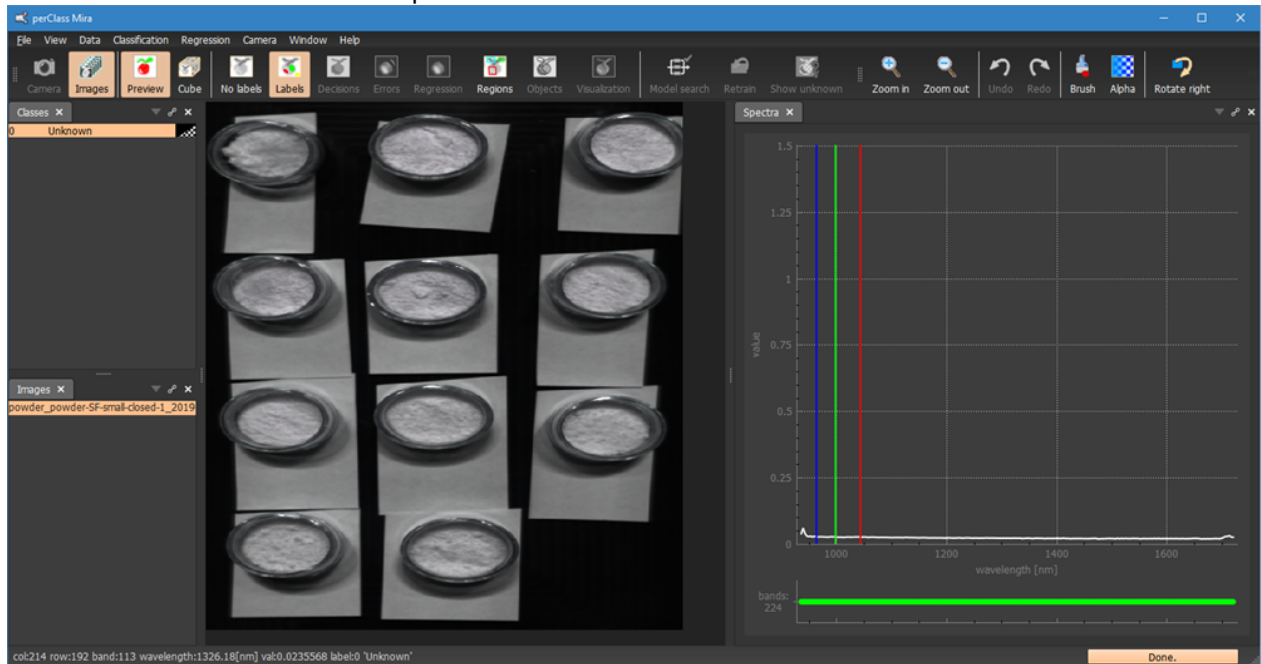
Regression modeling allows us to estimate numerical quality parameters from spectral data. For example, we may wish to estimate sugar content in a tomato or mixing proportion of powders. In perClass Mira, regression is performed at object level. Therefore, we need to define pixel classifier and one or more classes of interest. Then, we can assign external numerical values to each object and build a regression model. This model is then applicable to objects detected in a new image and can provide e.g. an estimate of



sugar content per tomato.

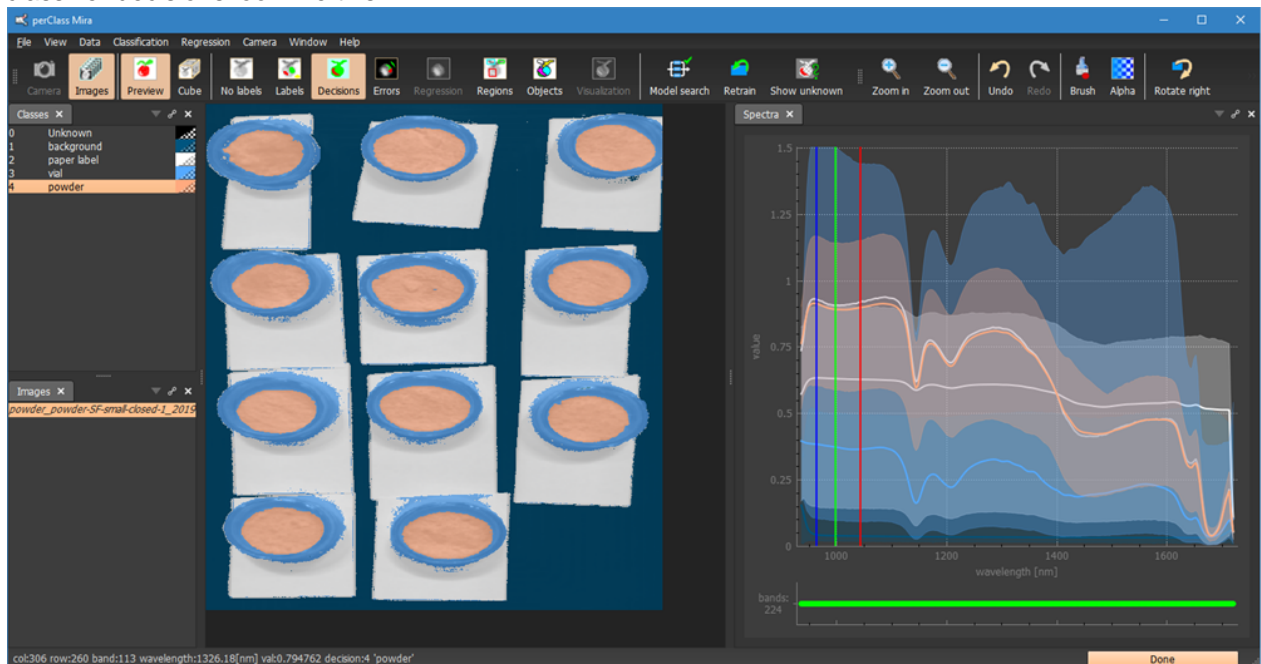
In this example, we use the powder data set with vials containing mixtures of two powders, namely flower and soda. Our goal is to train a model that will be able to estimate the mixing proportion for a new powder mix.

We have loaded the first scan with powders:



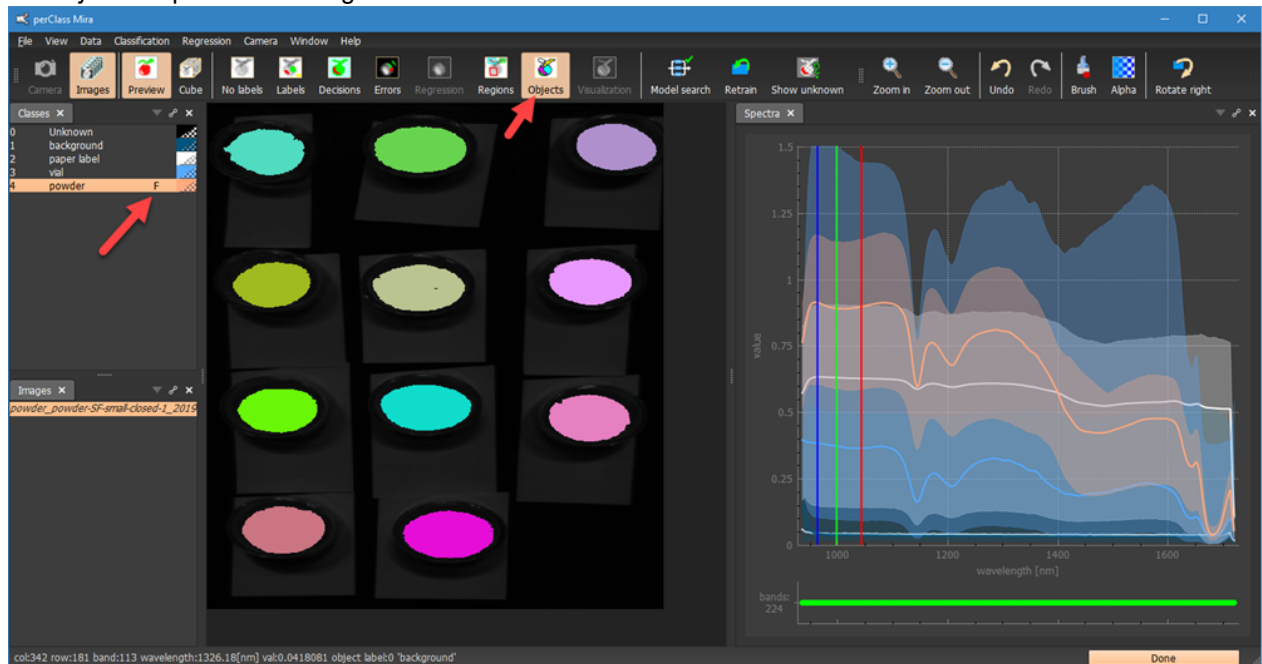
## Step 1: Pixel classification

In the first step, we create a pixel classifier. We only care about good quality segmentation of the powder content at this step. In our example, we define classes of background, paper label, vial and powder. Our classifier decisions look like this:



## Step 2: Object segmentation

In the second step, we create an object segmentation. We flag the *powder* class as foreground and click on *Objects* to perform the segmentation.



You may use more than one foreground class in regression. For example, you may build a model with separate pixel classes for white and dark grapes, flag both as foreground and use both in regression modeling.

## Step 3: Object annotation

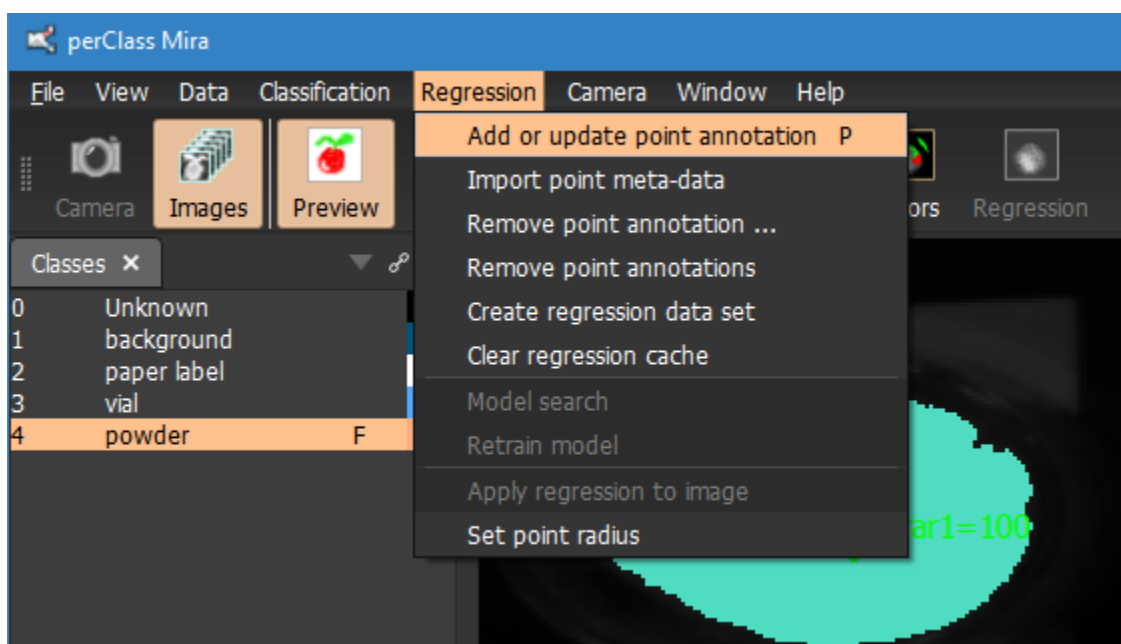
In the third step, we annotate individual objects with numerical values denoting the true mixing proportion of powders. We use the following image describing the ground-truth:





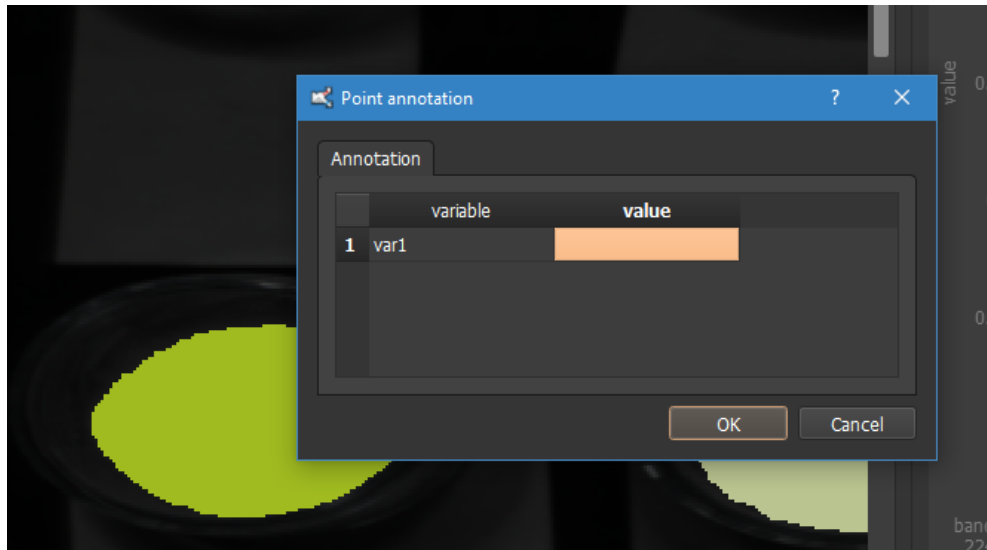
For each powder object, we add a single number that corresponds to the percentage of soda in the sample.

We can add an annotation from *Regression* menu with *Add or update point annotation* command:

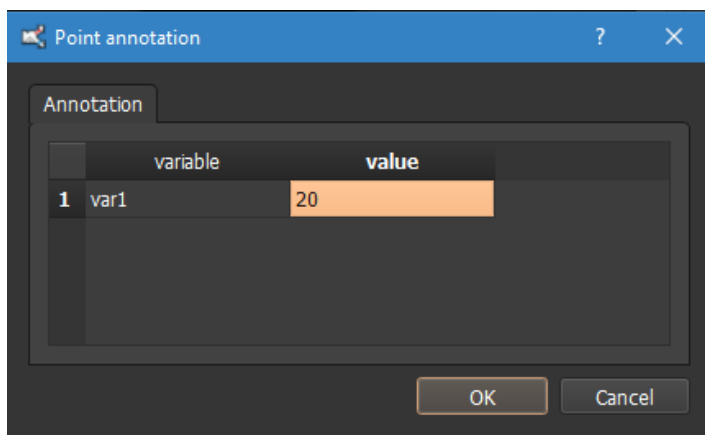


More convenient is to use the keyboard shortcut. We position the mouse pointer on top of the desired

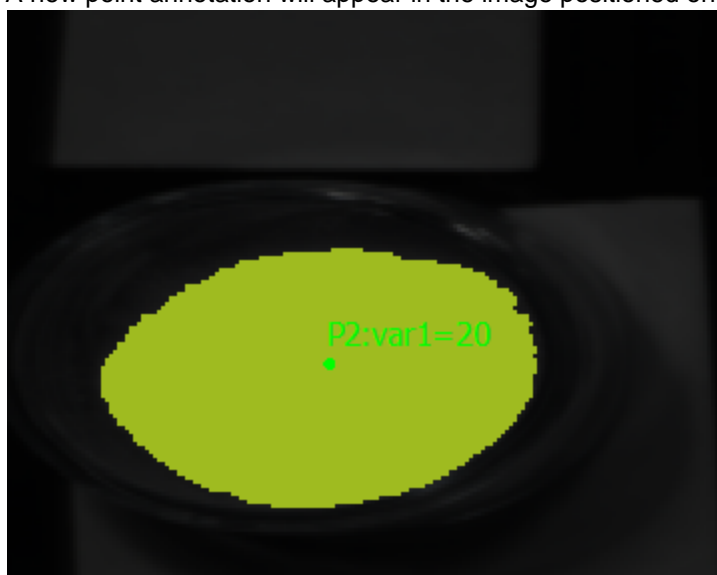
object and press *P* (for point). A dialog will appear:



We may directly type in the numerical ground truth value, in our case 20 and press Enter twice (the first time to confirm entering the value, the second time to confirm the dialog):

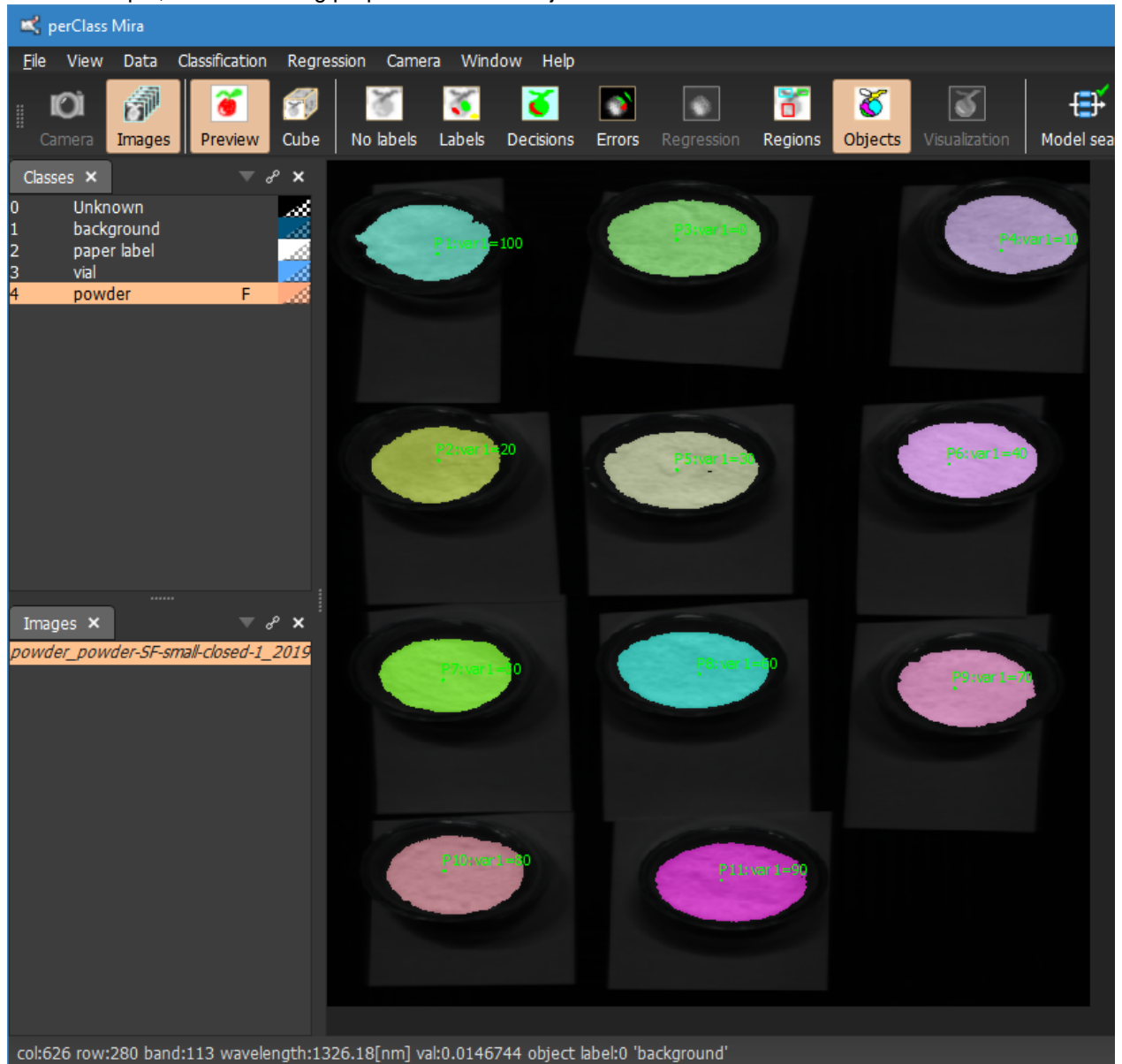


A new point annotation will appear in the image positioned on our original mouse pointer location



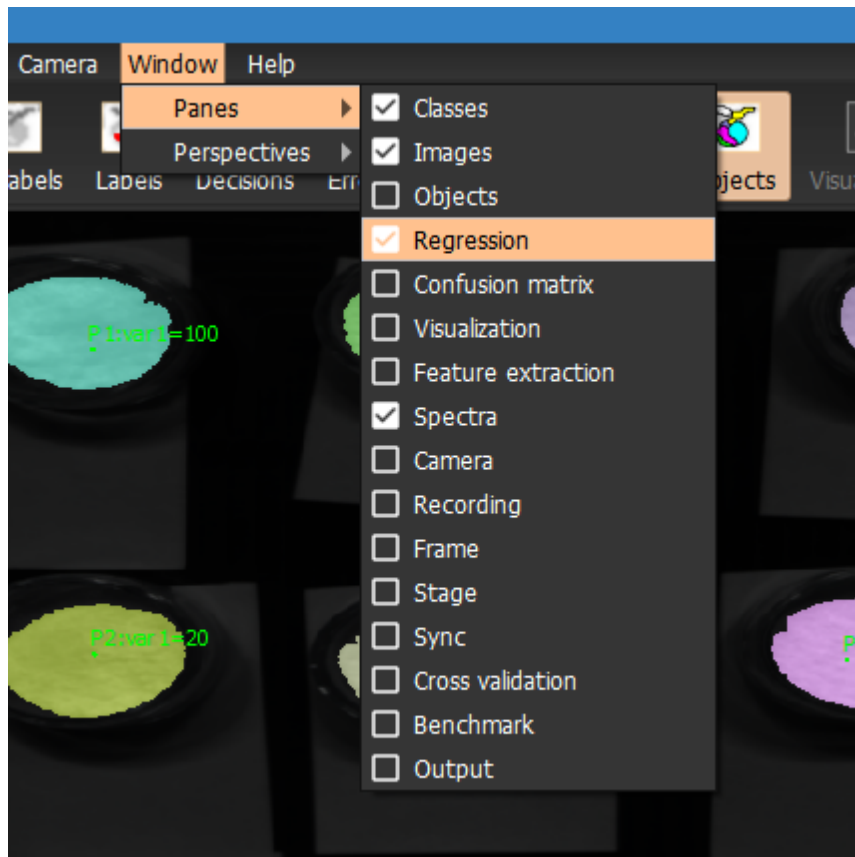
Each point has a unique number in the project which is assigned automatically. We may move the point around. By double clicking the point, we may edit the attached values.

In our example, we fill in mixing proportions for all objects:

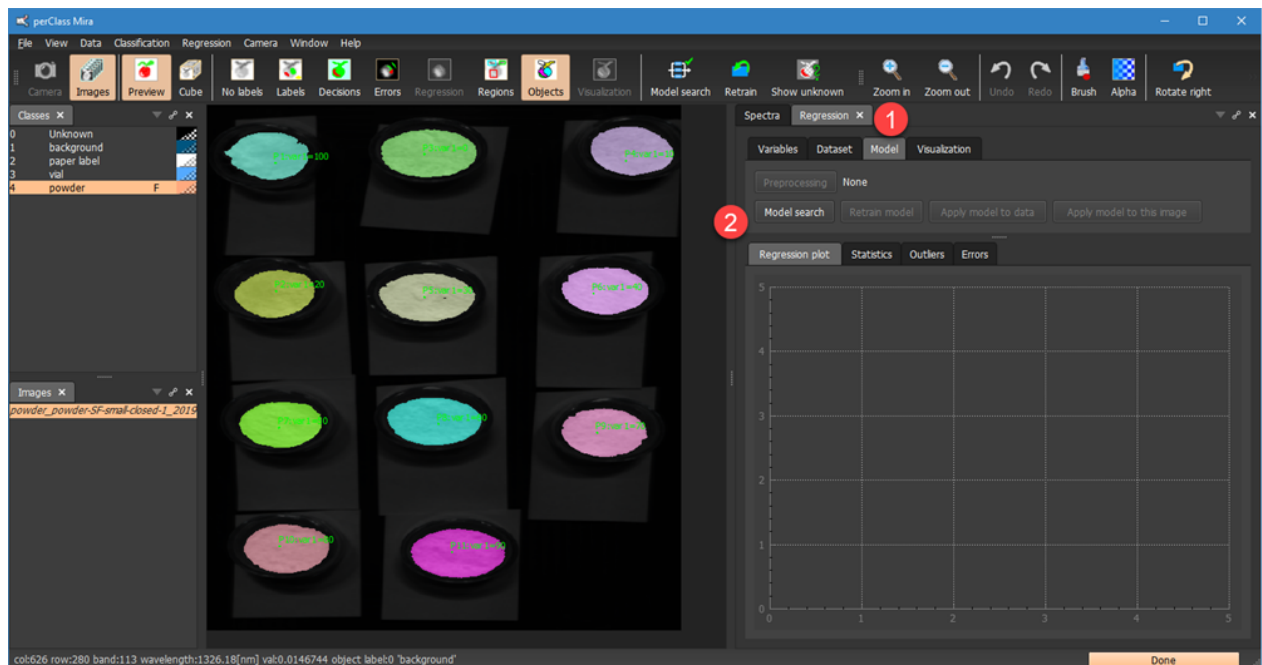


#### Step 4: Regression modeling

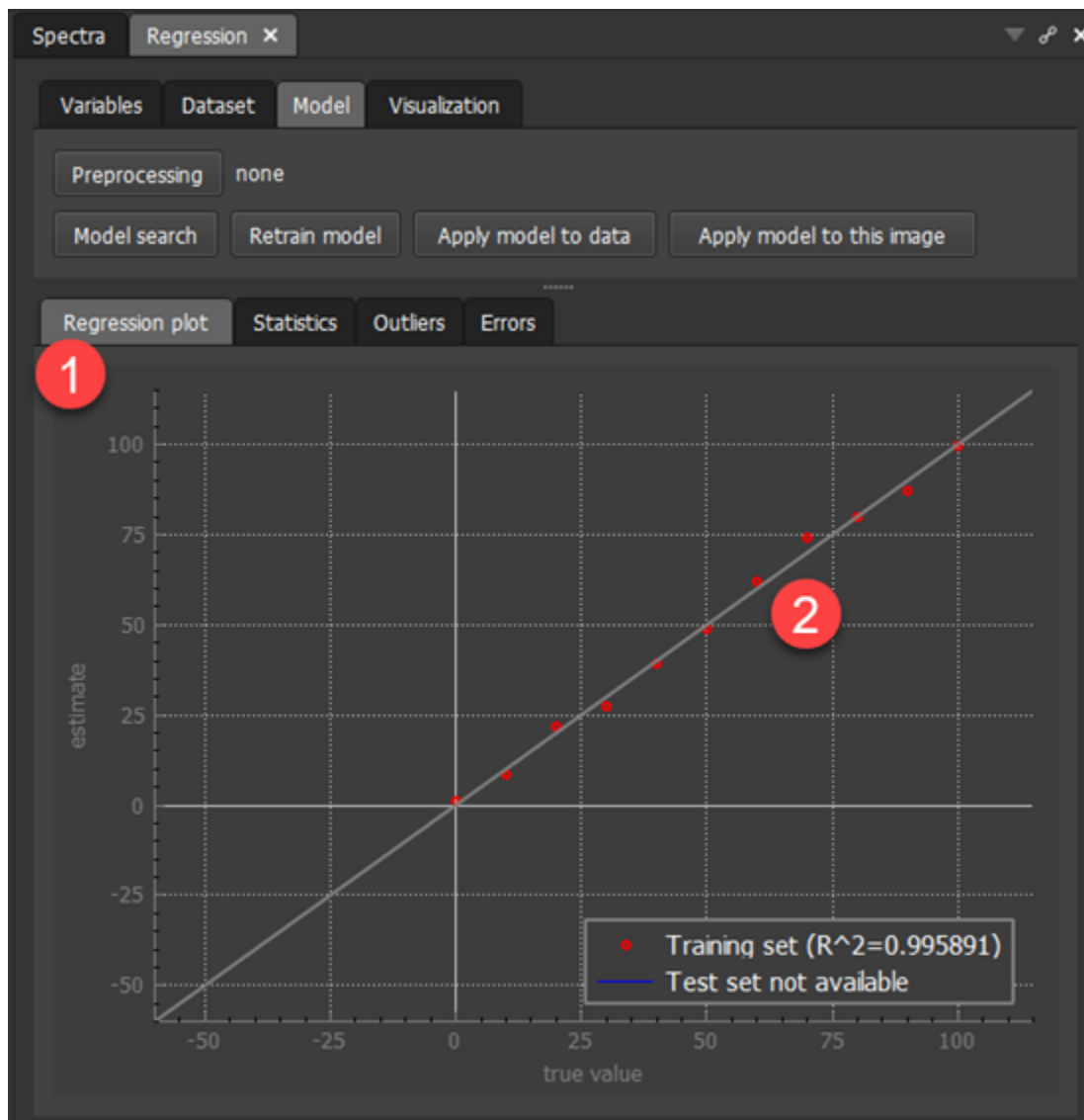
In the fourth step, we will build a regression model. We open the *Regression* panel (if not visible, enable it in the *Window / Panes* menu)



We can now perform regression *Model search* using the button 2



Similarly to the classification, the regression model is automatically built and the results are reported in the *Regression* panel:

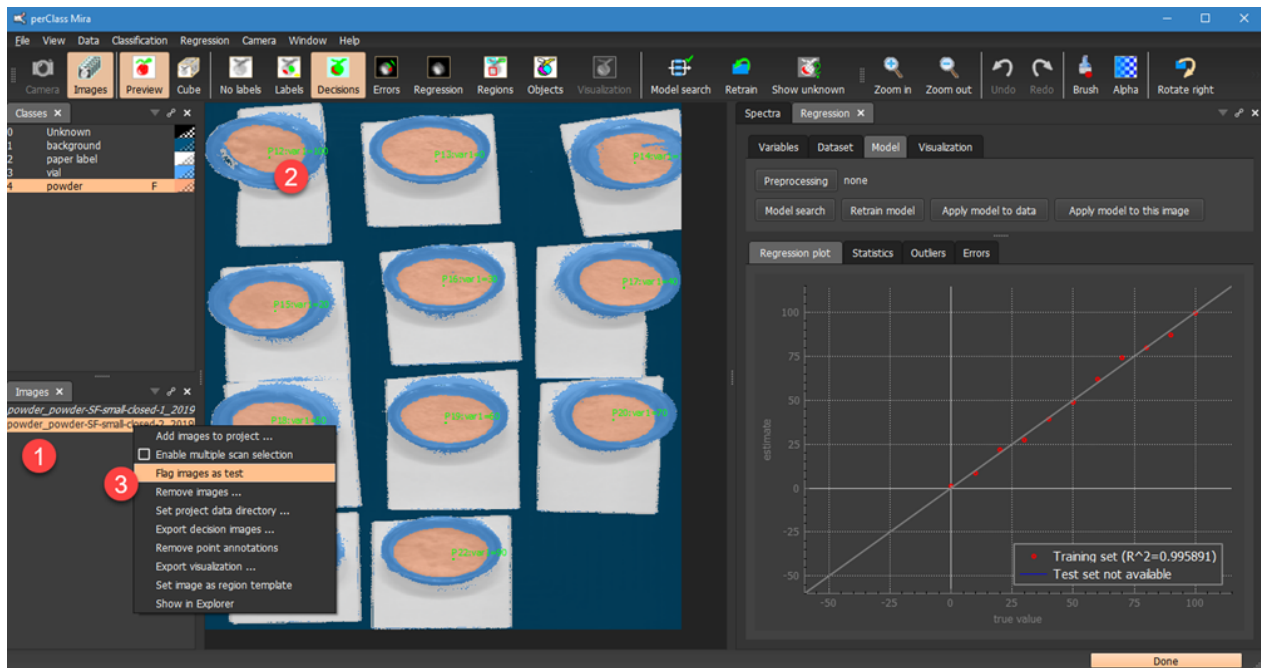


Specifically, the regression plot <sup>1</sup> will show number of red points <sup>2</sup>. Each point corresponds to one object. The red color denotes training objects.

### Step 5: Defining test data set

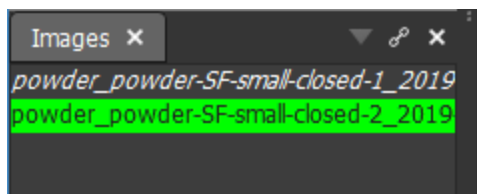
Similarly to classification, in regression modeling it is very important that we build sufficiently large and representative test set. We need the test set to estimate performance of our regression model on example unseen in its training.

In perClass Mira, test flag applies to entire images. Therefore, we need to include at least one more image and annotate its objects with the numerical ground truth.

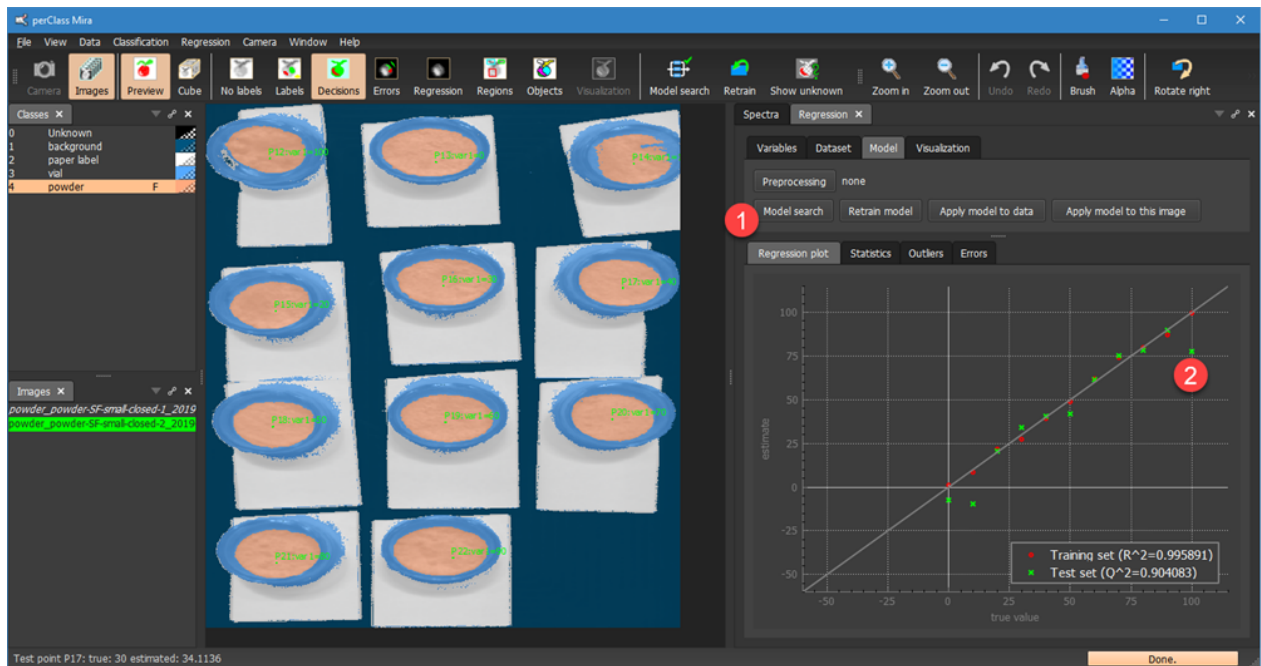


We have added a new scan using the context menu in *Images* list and *Add images to project...* command

1. We apply the classifier and annotate the new objects 2 with the respective ground truth taken from our external annotation source (not shown). We follow the process in the [Step 3](#) to annotate the new objects. Finally, we flag the new image as test via the context menu and *Flag image as test* command 3. The image color will change to green to indicate its test status.



We can now rebuild our regression model by pressing *Model search* in the *Regression / Model* tab 1:



The red points do not change, because identical information is used in training. However, we will see a new set of green points <sup>2</sup> referring to our new test objects.

### Step 6: Improving regression model

There are number of ways we can understand performance of a regression model and improve it.

In order to understand performance, we can

- Visually judge regression results in the [regression plot](#)
- Inspect number of commonly used [performance measures](#)
- Use the [outlier plot](#) visualizing difference of the data from the regression model (for any any detected as it does not need ground truth)
- Use the [error plot](#) to visualize differences of estimated values from the ground-truth

We may improve the regression model by

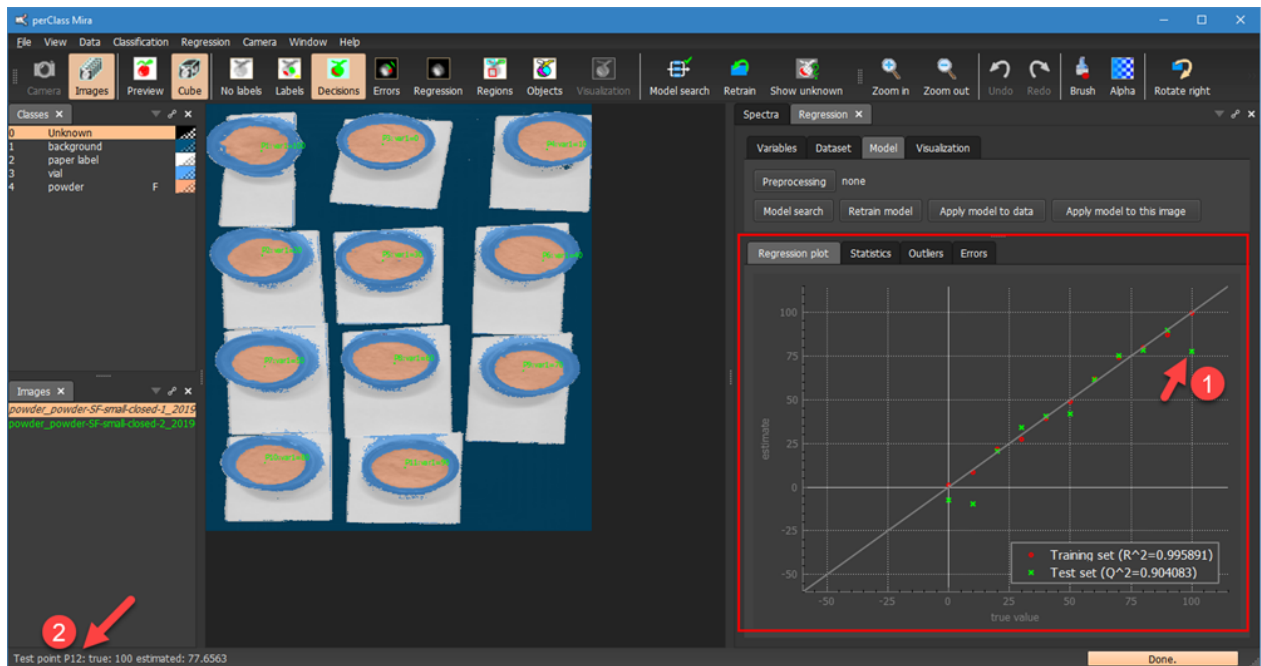
- Using only subset of spectral bands
- Using specific data preprocessing
- Curating the training set by removing suspicious samples or outliers

### Regression plot

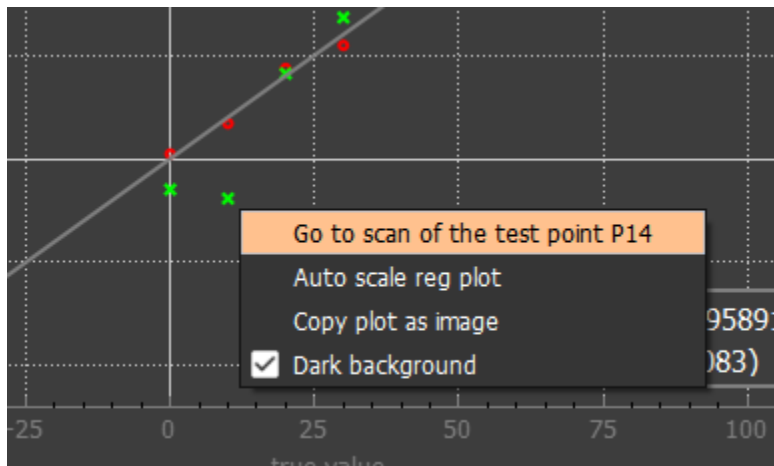
Regression plot serves for quick visual overview of the regression model performance. In the X-axis, the ground-truth for each annotated point is given. The Y-axis represents the estimated value. Red points correspond to the training set and gree to the test set.

When hovering over the plot, the details of the closest point <sup>1</sup> are displayed in the status bar <sup>2</sup>.



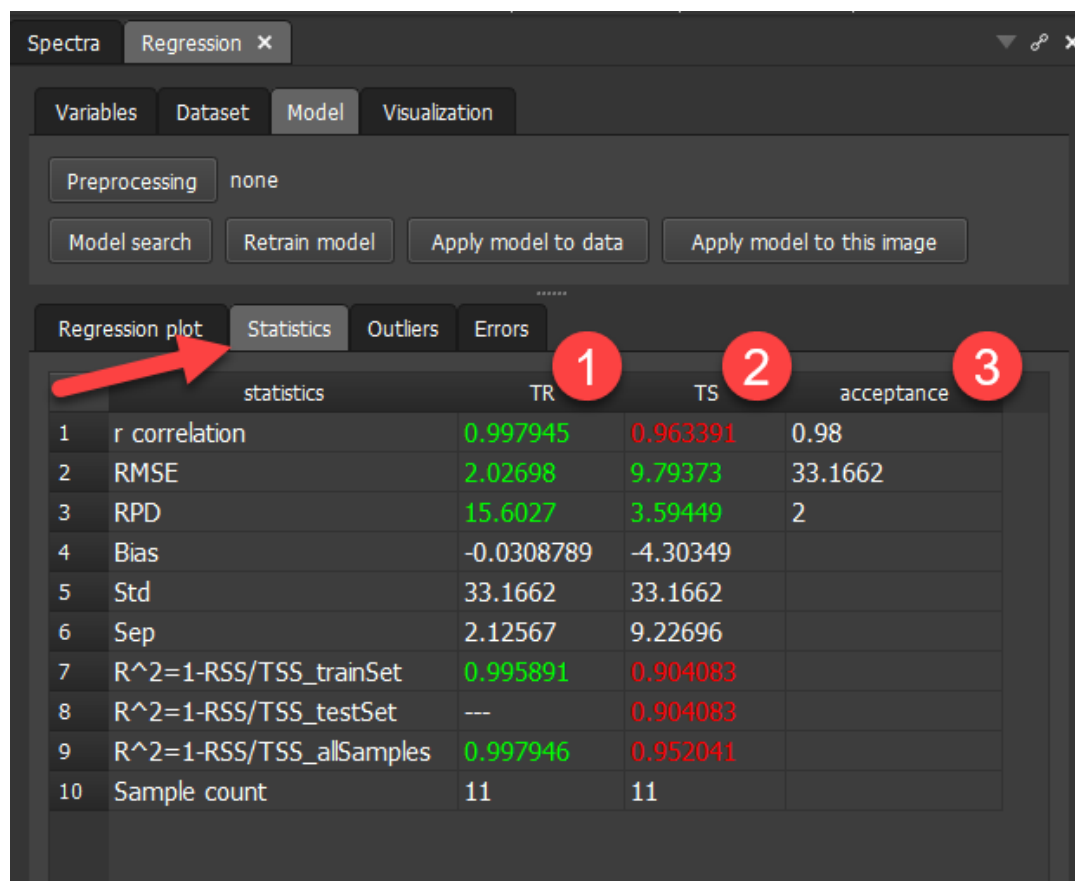


We may jump to the scan of the nearest point using the context menu and *Got to the scan of the point* command:



## Performance statistics

The *Statistics* tab provides a summary of the most common performance measures. Each measure is estimated on the training set <sup>1</sup> and on the test set <sup>2</sup>.



The *Acceptance* column provides user-adjustable acceptance criteria for selected measures. The color of the performance measure reflects the acceptance status. For example, when we set the acceptance for correlation to 0.95, the accepted status in green and not-accepted in red will update.

	statistics	TR	TS	acceptance
1	r correlation	0.997945	0.963391	0.95
2	RMSE	2.02698	9.79373	33.1662
3	RPD	15.6027	3.59449	2
4	Bias	-0.0308789	-4.30349	
5	Std	33.1662	33.1662	
6	Sep	2.12567	9.22696	
7	R <sup>2</sup> =1-RSS/TSS_trainSet	0.995891	0.904083	
8	R <sup>2</sup> =1-RSS/TSS_testSet	---	0.904083	
9	R <sup>2</sup> =1-RSS/TSS_allSamples	0.997946	0.952041	
10	Sample count	11	11	

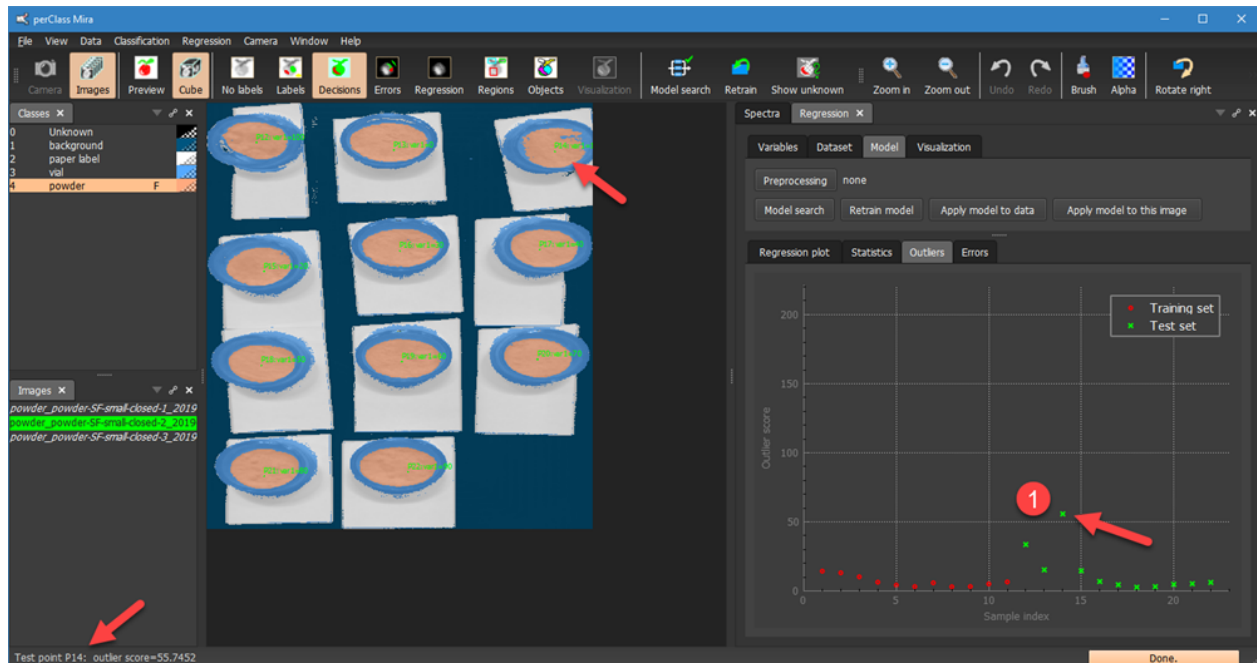
Performance statistics may be copied out as text by selecting specific cells in the table and using *Copy statistics as text* command. These values can be directly pasted into Excel table.

**TIP** Complex cross-validation schemes are easy-to-perform in perClass Mira using the [Cross-validation tool](#)

## Outlier plot

The *Outlier* tab visualizes Outlier score for each object. It reflects the distance from the current regression model.

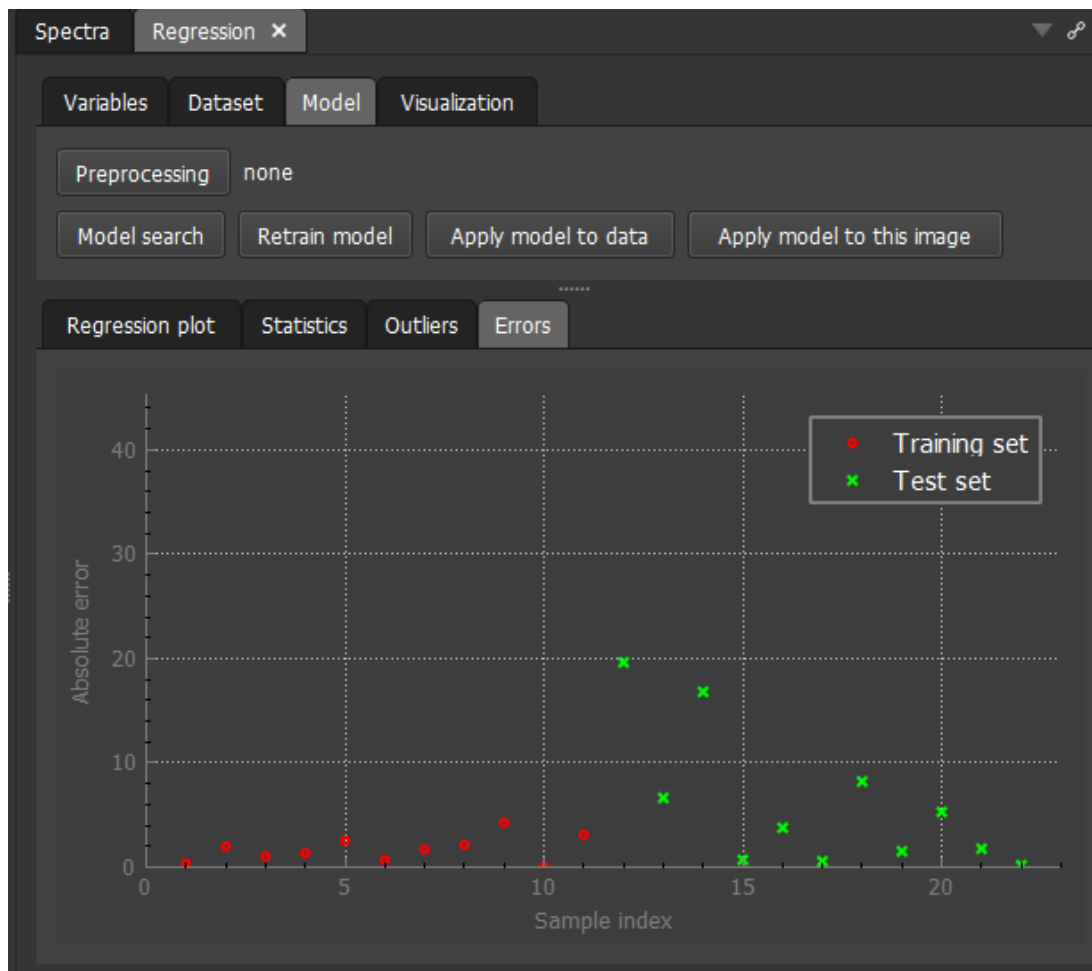
In the following example, we can see the most outlying object <sup>1</sup> is the P14 indicated in the status bar and in the image by arrows.



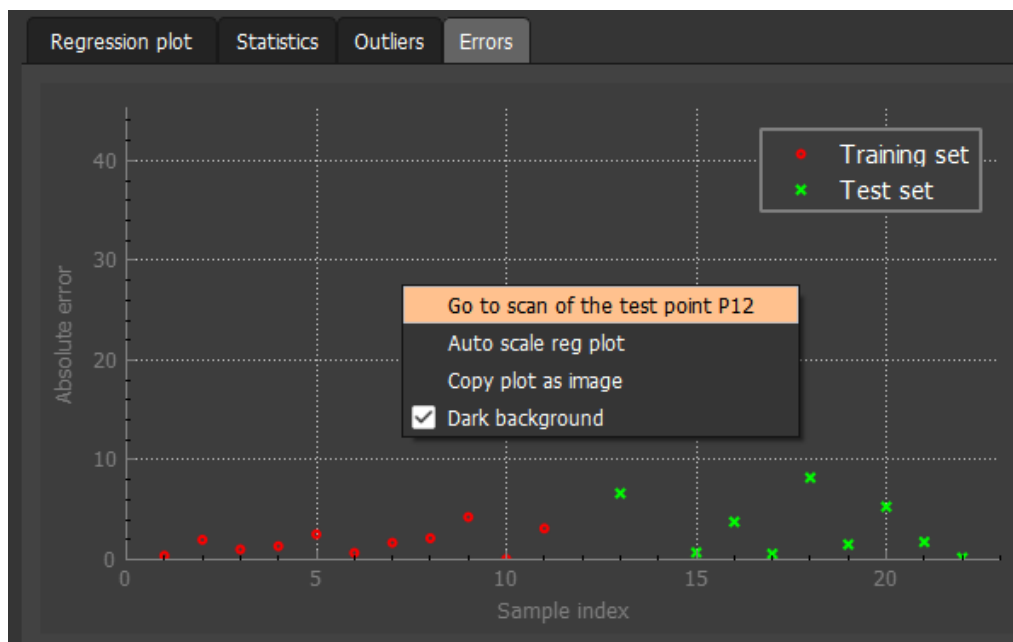
Note, that ground truth information is not needed when computing the outlier score. Therefore, it may help us to understand any new observations.

## Error plot

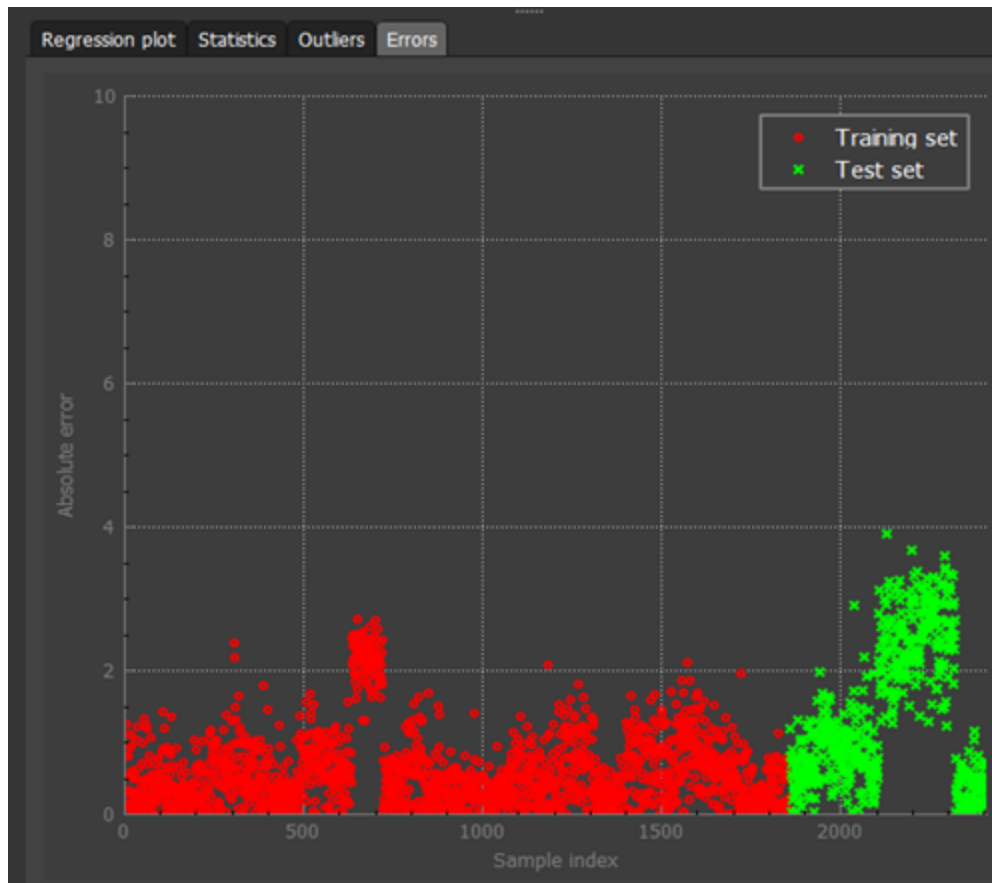
Error plot visualizes the difference between the ground-truth and the estimate value. Similarly to the [Outlier plot](#), the samples are sorted displaying all training samples and then test samples.



Context menu in the error plot provides commands to jump to a scan of a specific point, automatically scaling the plot axis and copying the plot as an image.



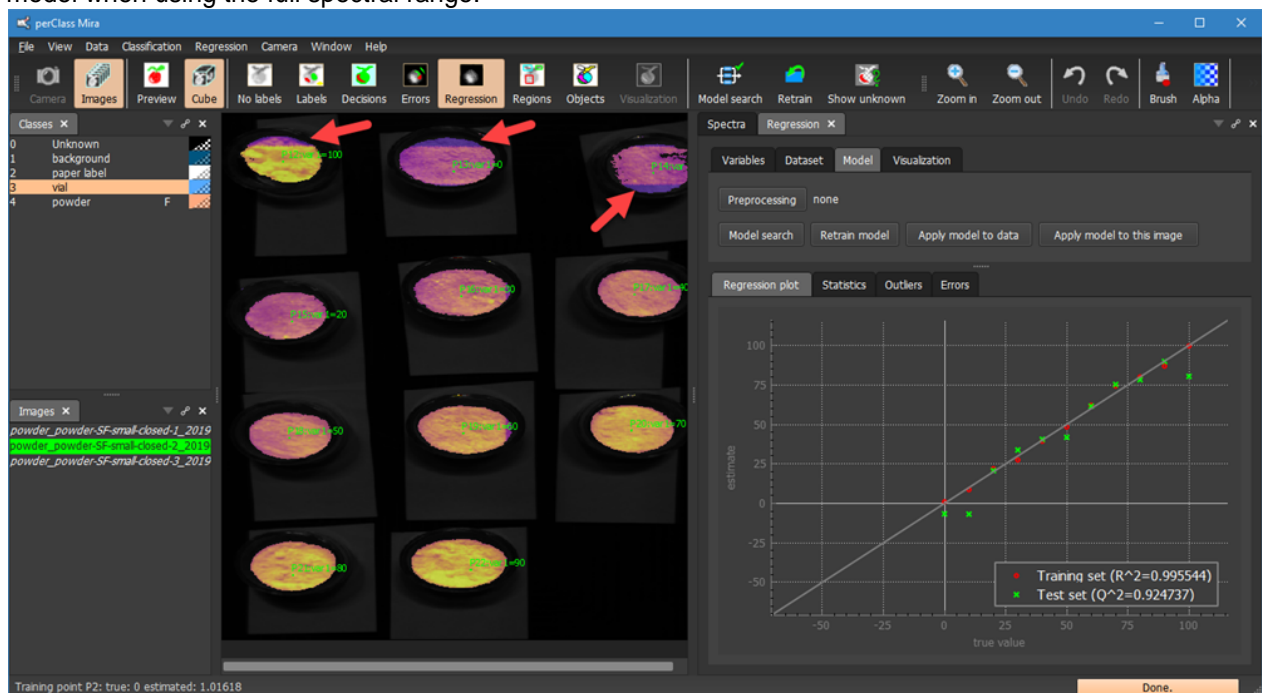
The example below originates from a different real-world project. It shows that we may easily spot systematic errors do to strong grouping information presented in the error (or outlier) plots. The groups with higher error may refer to different fruit varieties, producers, processing setting and similar.



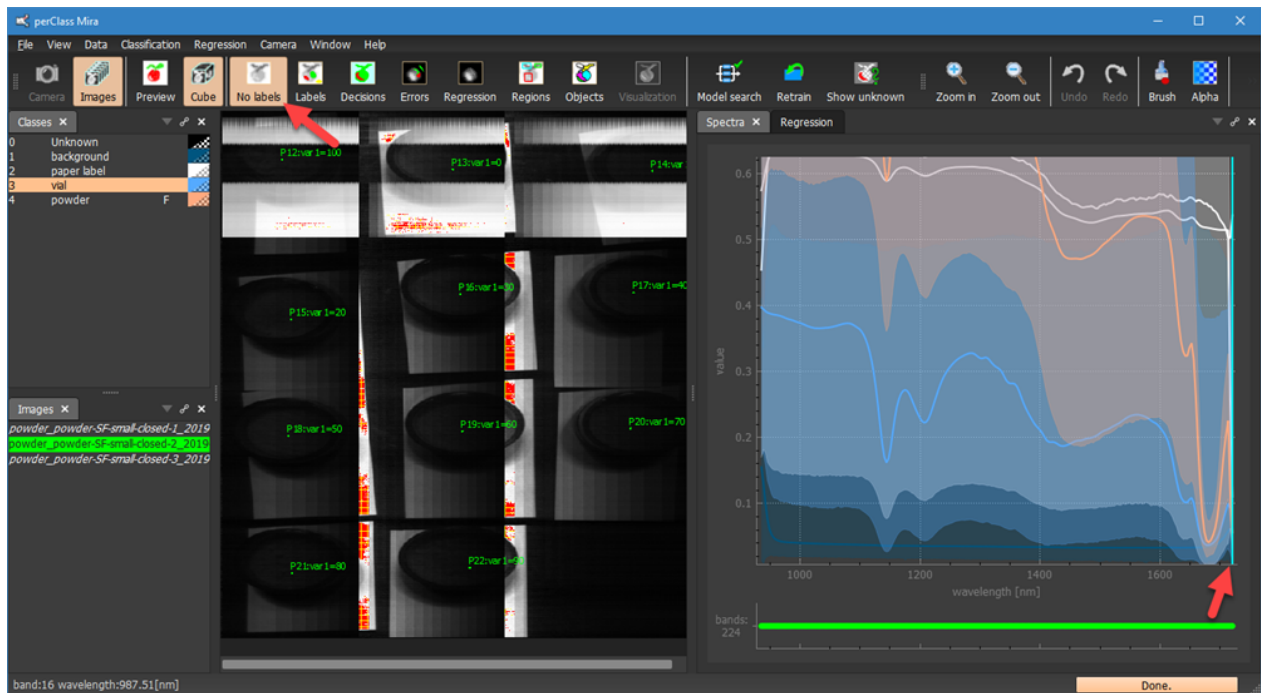
### Regression using subset of bands

By default, all available spectral bands are used when building the regression model. Similarly to classification, we may precisely control the band subset.

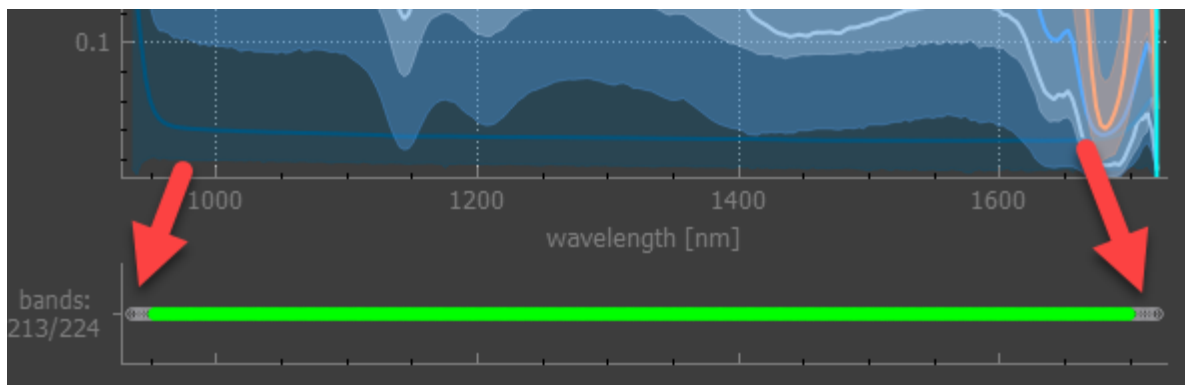
In this example, we use [Pixel visualization of the regression output](#) to show that there is some issue with our model when using the full spectral range:



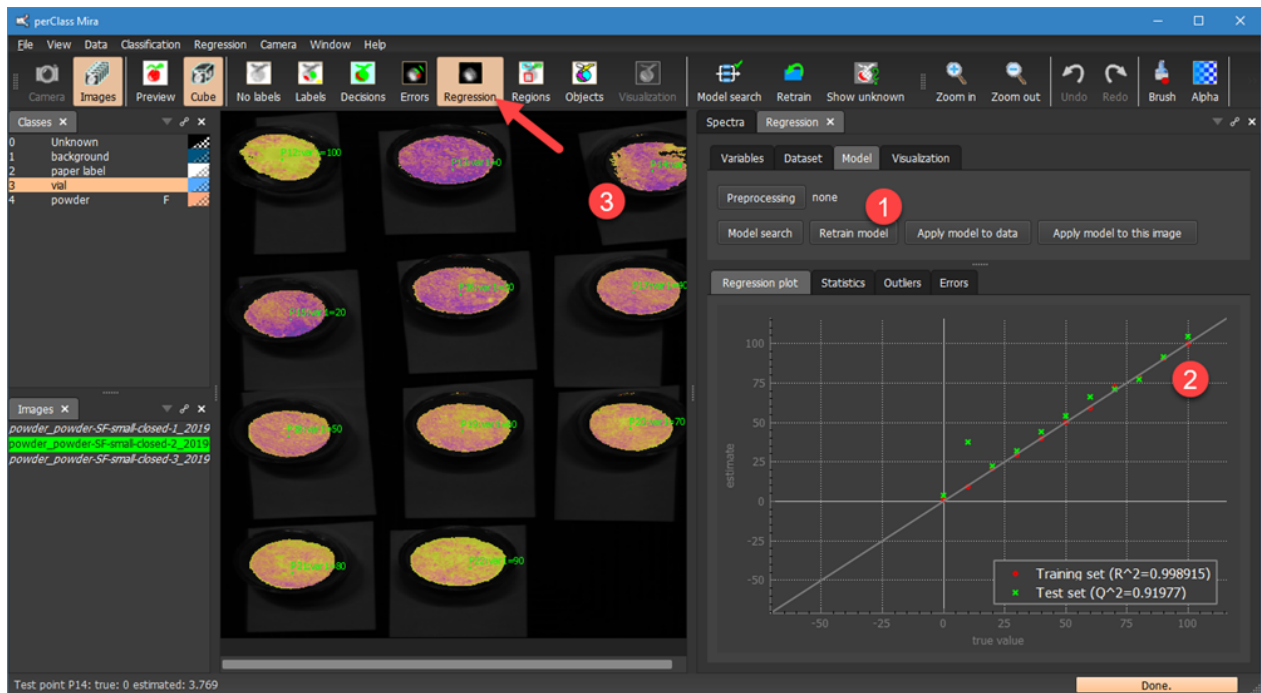
When we explore individual spectral bands, we can see that several first bands is highly noisy (not shown here) and the few last bands contain strong image artifacts:



We may remove the bands at the start and end of the spectral range in the *Spectra* plot...



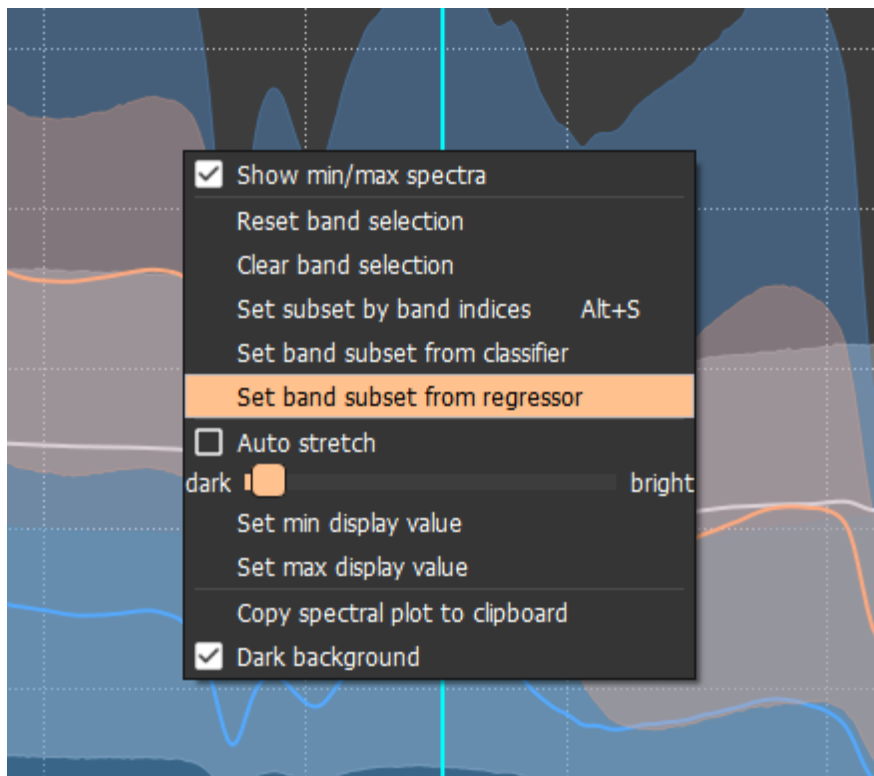
and retrain the regression model using the *Retrain* button <sup>1</sup>. We may observe that the *Regression plot* shows improved performance <sup>2</sup> and pixel regression output <sup>3</sup> lacks any artifacts.



## Regressor and classifier band subsets

In perClass Mira, we may use a separate band subset for the classifier and for the regression analysis.

The context menu in the *Spectra* panel shows the two commands that allows us to select the bands in the spectral plot based on the subset used when the last classifier or the last regression model was trained.



## Preprocessing

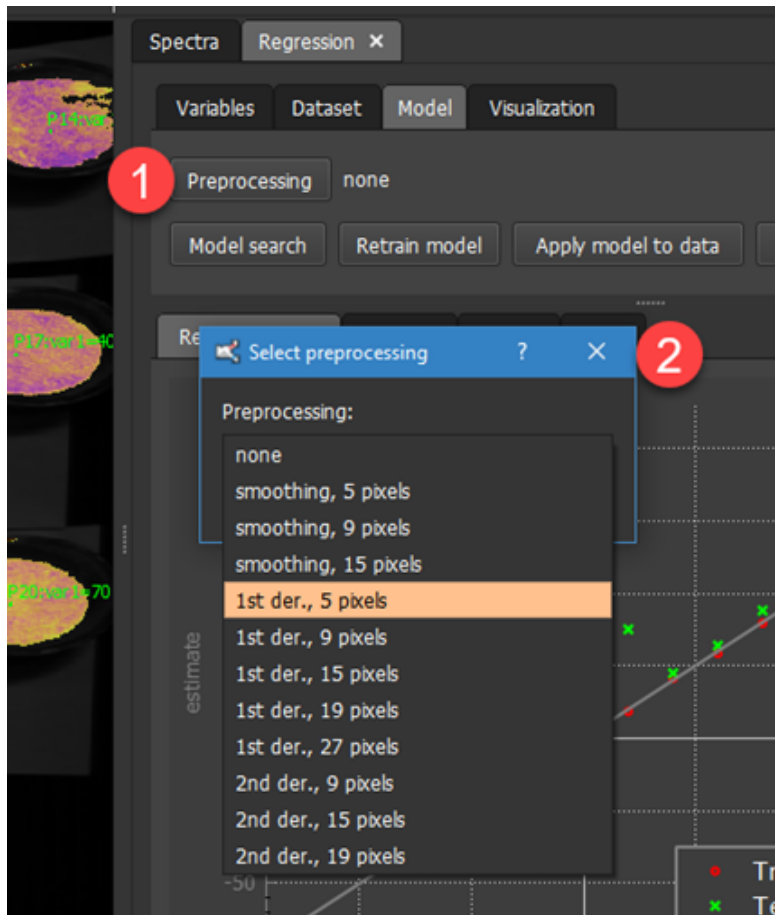
perClass Mira offers user-defined spectral preprocessing. This means that object spectra are not used as is by the regression model but filtered. Three filtering methods are provided:



- Smoothing
- 1st derivative
- 2nd derivative

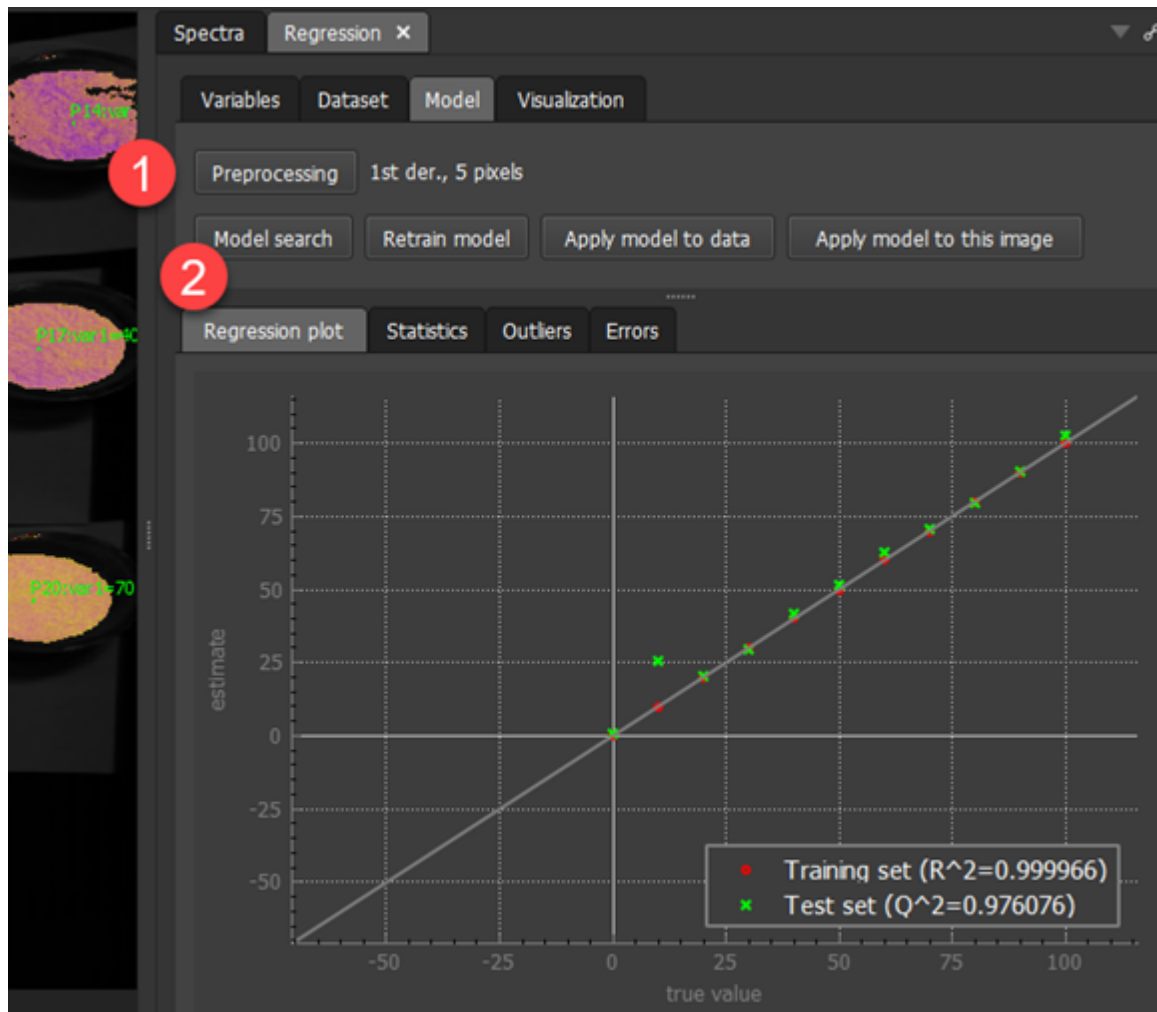
Each of the methods can be applied at a different spectral filter window size.

Preprocessing is set using the *Preprocessing* button <sup>1</sup> in the *Model* tab. When selected a dialog <sup>2</sup> will appear where the preprocessing method can be selected from a combo box.



After confirming the dialog with OK button, the preprocessing is set **and the mode needs to be retrained**

<sup>2</sup>. It is recommended to run full *Model search* and not *Retrain* after changing the preprocessing setting.



In general, we may say that smoothing improves regression if the data set suffers significant noise and the first derivative emphasizes the change of spectral change. However, it is not possible to say which method will work best without proper testing.

### Additional regression tools

#### Model search versus retraining

Similarly to classification, regression modeling offers two ways how to build a model using either *Model search* or *Retrain* buttons.

The *Model search* employs search for optimal complexity followed by model training. The *Retrain* command only retrains the model using the last selected modeling scheme.

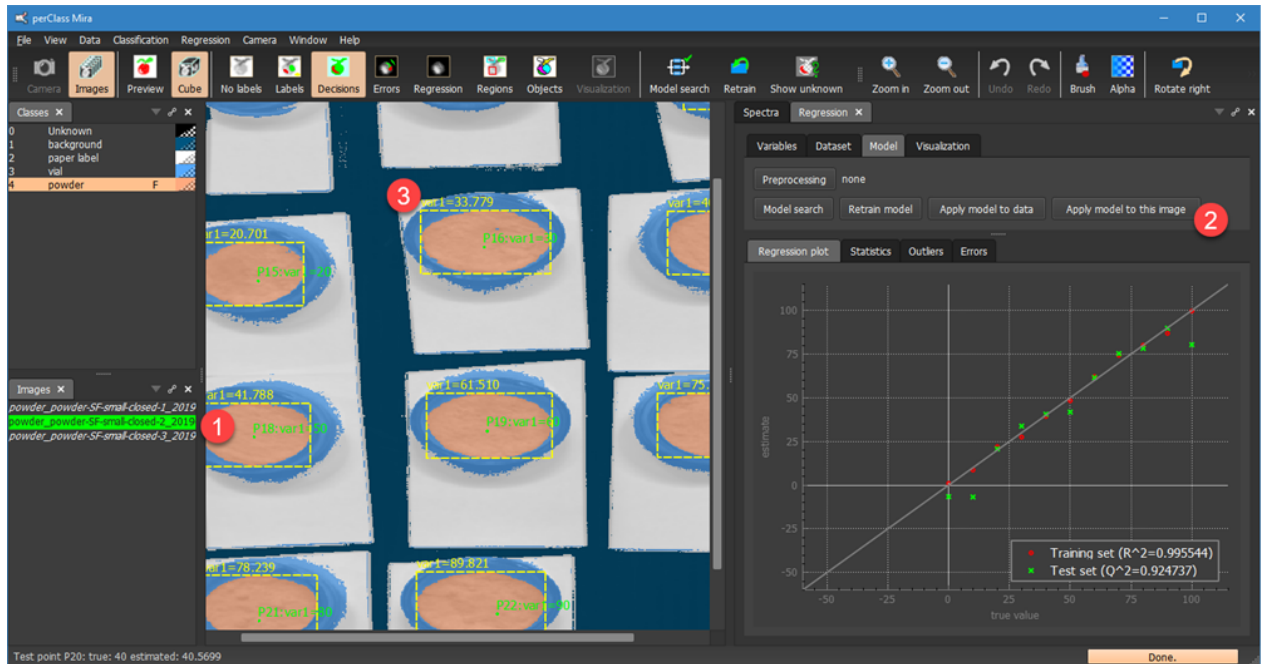
General recommendation is to use the *Model search* in case of large changes. For example, when we select a different preprocessing or different set of spectral bands it is better to perform full model search. If only a small change to our data set is introduced, for example, when we investigate impact of a potential object being removed, *Retrain* is more convenient as it reflects impact of only the single change made.

#### Applying to new images

The regression model can be applied to any scan (with or without ground truth) by selecting the scan 1

and pressing the *Apply model to this image* button 2 in the *Regression / Model* tab. The entire processing pipeline is applied to the scan. This means, that the pixel classifier produces decisions, objects

are segmented and a bounding box <sup>3</sup> is displayed for each object showing the estimated regression output.

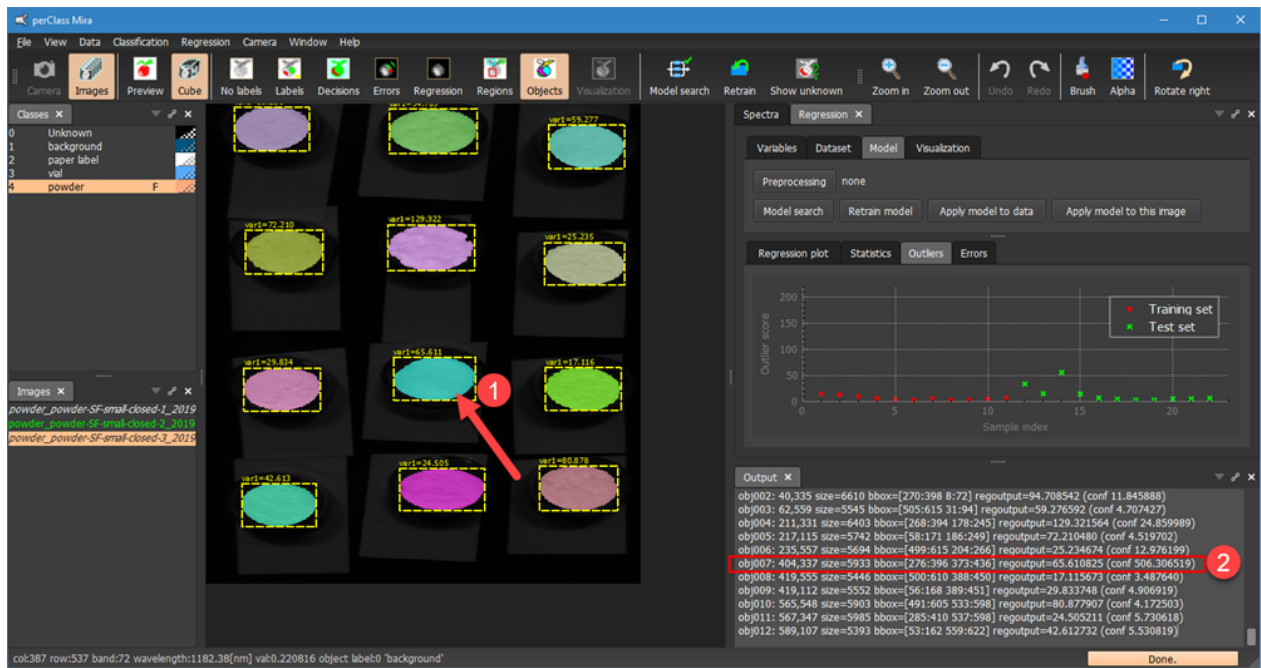


The *Output* panel shows detailed information on regression output. For each object a centroid, size,

bounding box and the regression output <sup>1</sup> are given. In addition, an outlier score <sup>2</sup> is provided as well. Low value of the outlier score (compared to training examples) means that the sample is similar to training set where the model was trained. High score indicates strong deviation from the known training examples. For training and test examples, the outlier score is also displayed in the [Outlier plot](#).



In the following example, we apply the regression model trained on soda/flour mixture to a scan containing also a vial with salt <sup>1</sup> :



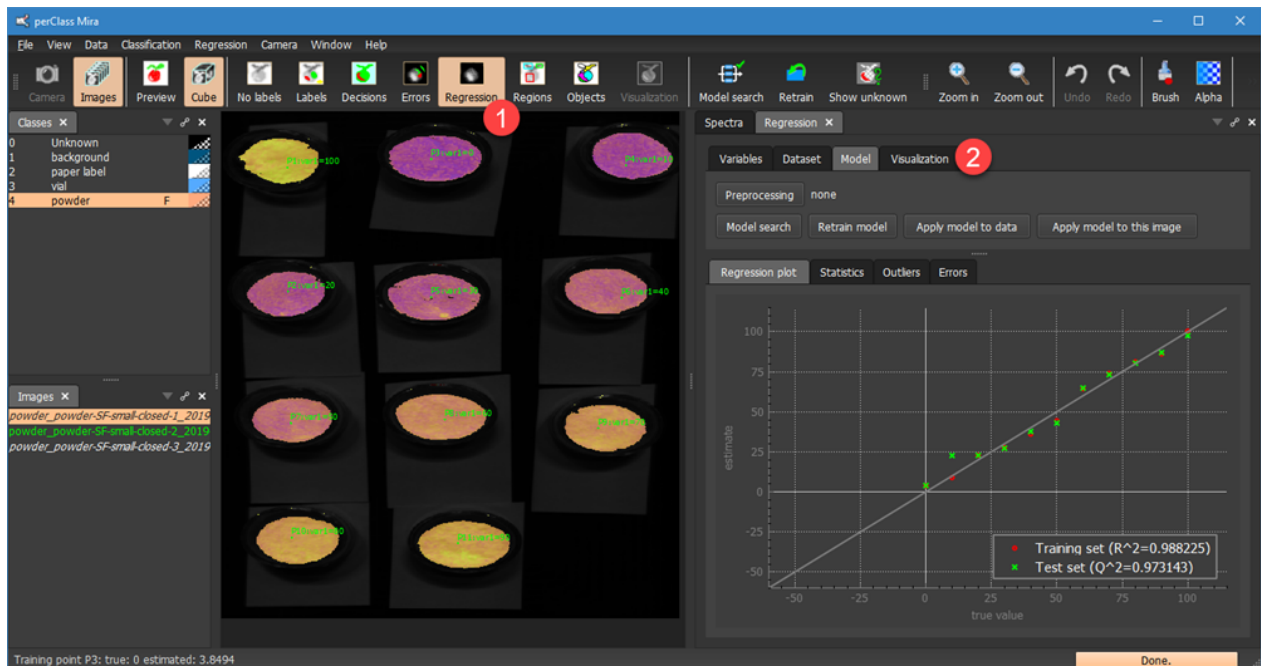
The corresponding object 007 information is highlighted with 2 shows that the outlier score is >500.0 while the other samples show the range of 3.0-12.0.

### Pixel visualization of regression output

When a regression model is built, we may apply not only [at object level](#) but also at pixel level using the

#### Regression toolbar button 1

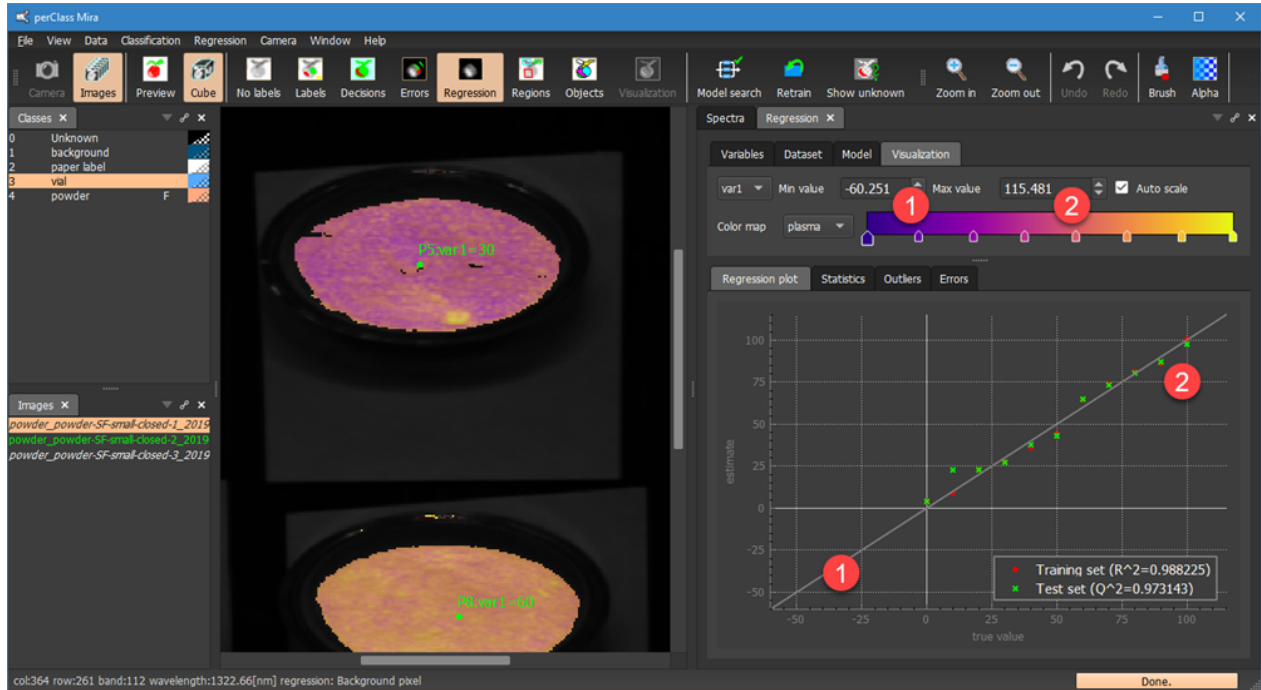
The pixel visualization of regression output allows us to understand inhomogeneities within the objects and spatial distribution of the modelled phenomena.



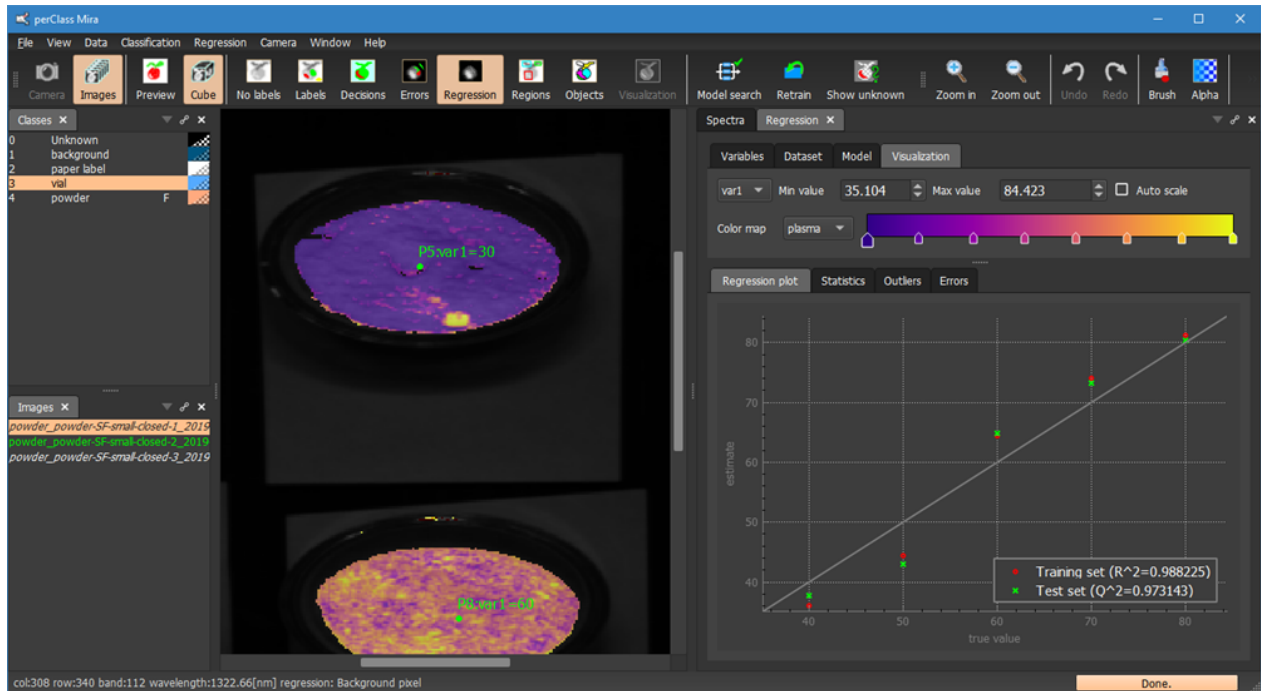
The display shows only the foreground pixels. The *Visualization* tab 2 provides number of options to fine-tune the rendering of the estimated output.

In the following example, we stretch min <sup>1</sup> and max <sup>2</sup> values of the rendering either via edit boxes or by a mouse-wheel in the left-lower or right-upper corner of the regression plot, respectively:

### Before stretch:



### After stretch:



Note, that the floating point regression output does not change, we only affect its rendering via the specified colormap and visualization settings.

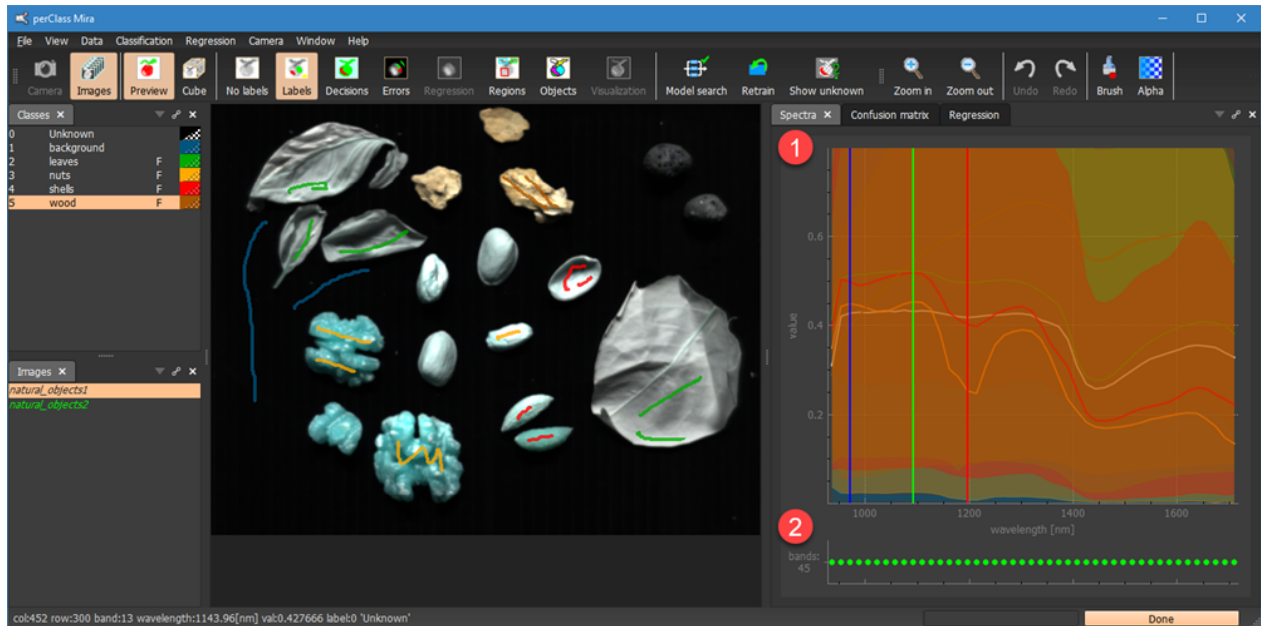
For more information on colormap control, see the [description in the visualization section](#).



## Spectral plot

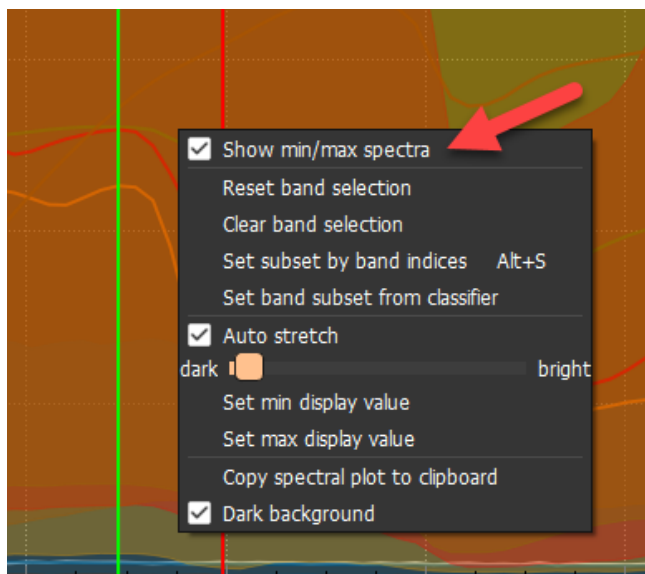
Spectral plot <sup>1</sup> displays information on labeled data across all spectral bands. The horizontal axis shows wavelength in nm. The vertical axis the pixel value. For the sake of image interpretation it is recommended to use data converted into reflectance. In perClass Mira, specific project types correct raw data into reflectance using white and dark reference scans. Working in reflectance makes models more robust regarding illumination changes.

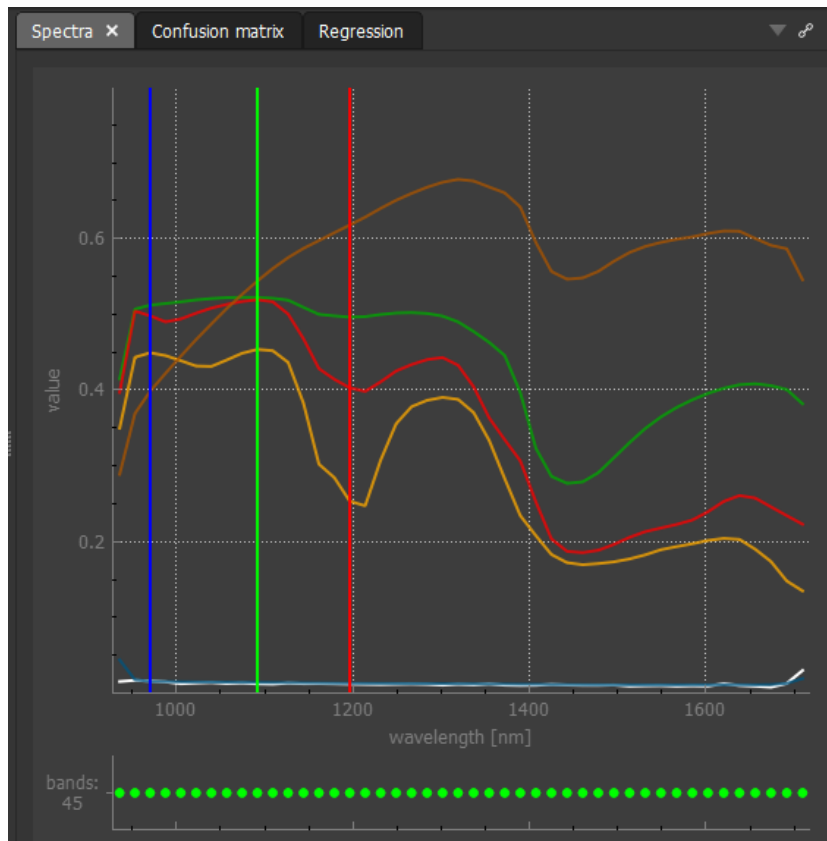
The band widget <sup>2</sup> under spectral plot shows individual spectral bands available in all images currently loaded in project. perClass Mira requires that all images use the same spectral bands/wavelength definition.



## Class-specific display

For each class with labeled samples, spectral plot provides a mean spectrum. By default also the min and max values per band are shown. The intention is to display extra information on variability of spectral signal per class. The min/max spectral range per band may be disabled in the context menu:





### Display range and scaling

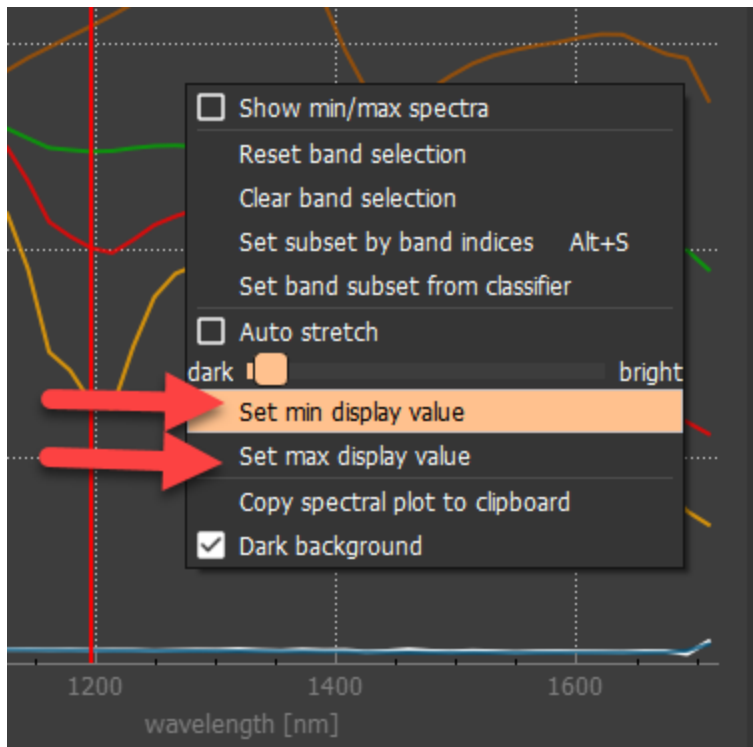
The vertical axis spectral values are by default set based on the first image loaded so that the most data is visible.

By default, **manual image visualization mode** is active where the spectral value range, defined by the spectral plot, is applied to all images. This means that switching between images, we see the same range and the gray value (for *Band* display) or RGB value (for pseudo-color preview) represent the same value in the data.

We may interactively adjust the bounds of the display value. This can be done either via the mouse wheel when the mouse pointer is close to top/bottom of the spectral plot or by a click and drag operation.

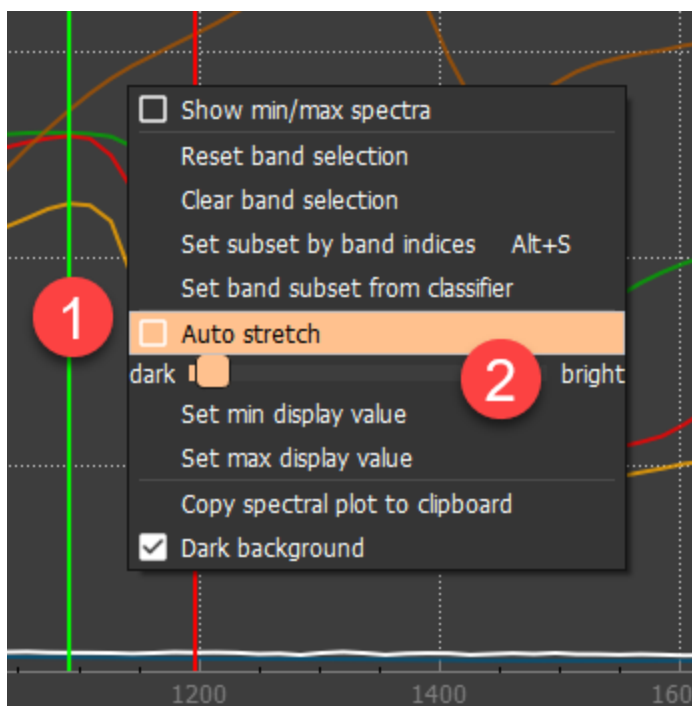
Alternatively it is possible to precisely define the numerical display values from the context menu:





Alternatively, we may enable the **automatic stretch** of display range. This mode stretches min/max of the spectral plot and hence image display specifically for each image. This is useful to always "see" meaningful content in each image irrespective of its overall brightness or darkness. However, it is important to keep in mind that at the same gray value or color in two images represents different raw spectral or reflectance values.

The *Auto stretch* mode is enabled by the checkbox <sup>1</sup> in the context menu. The display stretch is based on image content keeping certain percentile of image values in the view. This may be controlled using the slider <sup>2</sup> in the context menu.

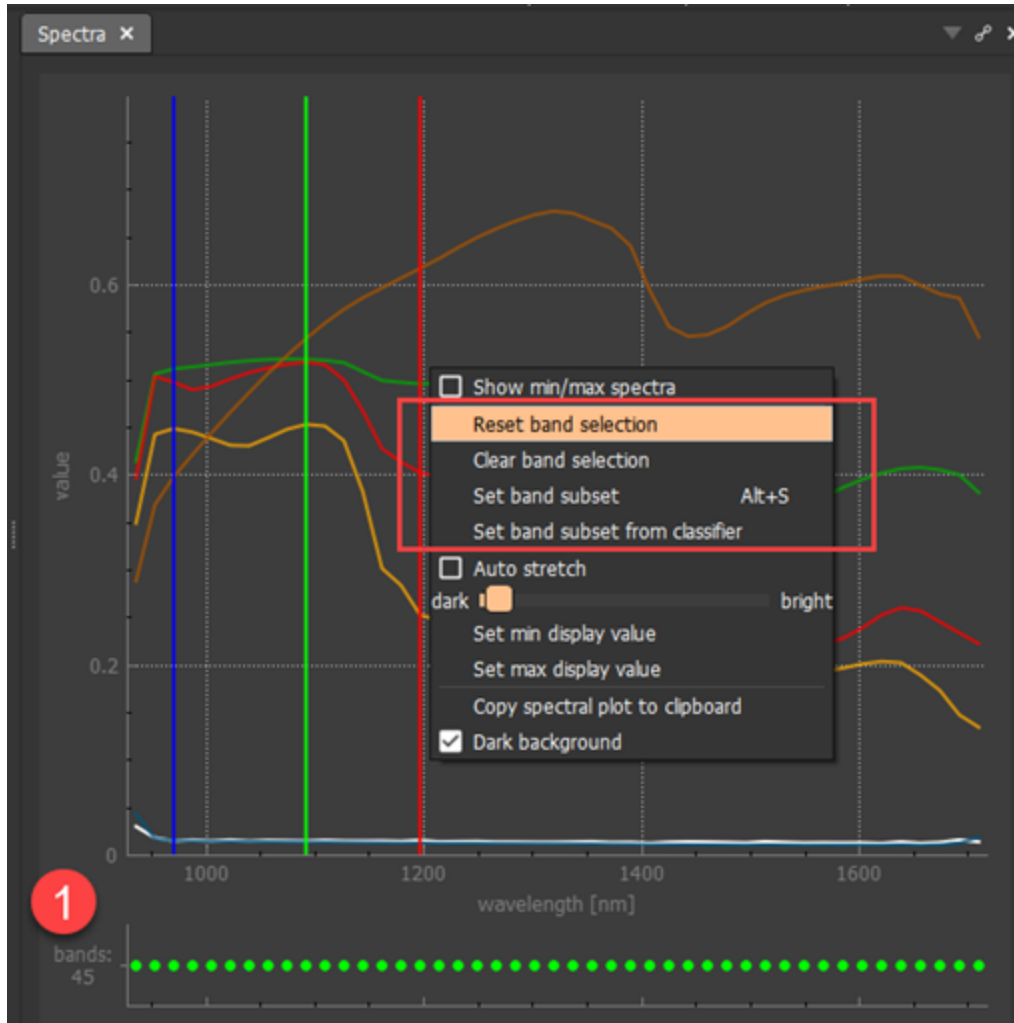


Adjusting display manually disables the *Auto stretch* function.

## Band selection

Under the spectral plot, you may find the band widget <sup>1</sup> providing user-defined band selection. For each spectral band in the data, we can see a corresponding round point. By default, all available bands are selected (green) and, when a classifier or regressor are trained, used for building the model.

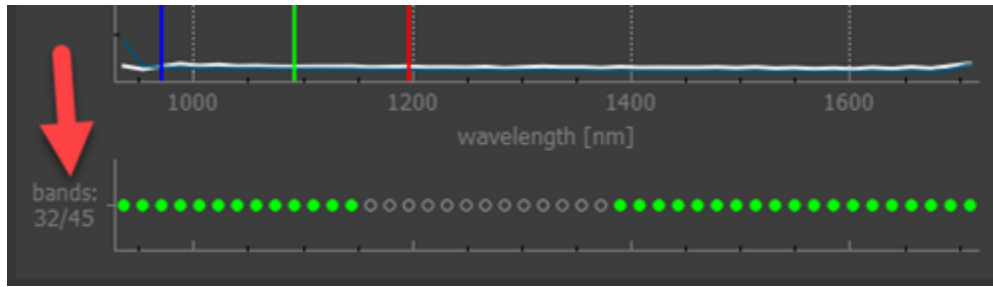
Spectral plot provide several commands allowing us to effectively define and work with band subsets. Some of these are highlighted by the red rectangle in the context menu screenshot:



### Manual band definition

Bands may be selected / deselected by clicking the individual round points in band widget. In order to select/deselect larger number of bands, we may use "painting" i.e. click and drag. The status of the first clicked band defines the action - either select or deselection of all visited bands.

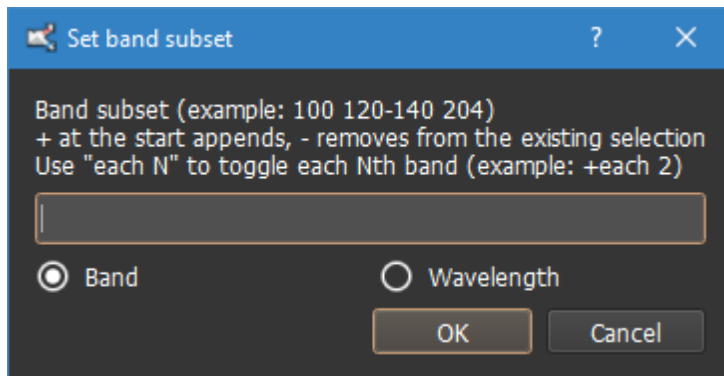
Note, that the number of selected bands and the total number of bands are provided to the left of the band widget:



### Band selection by commands

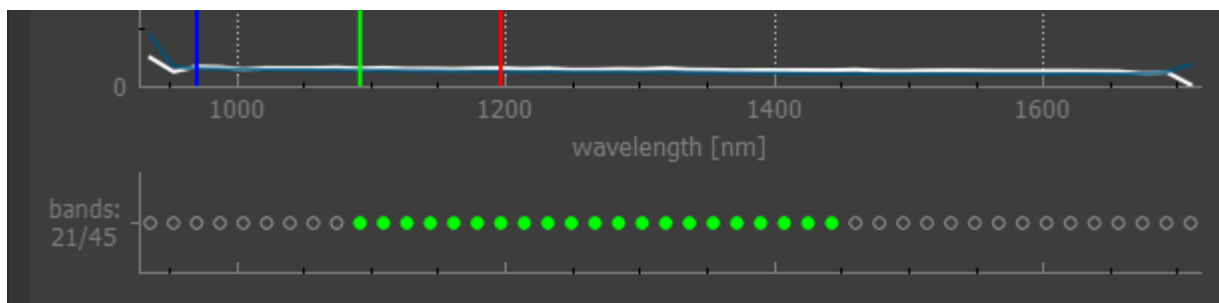
We may select and deselect all bands using the *Reset band selection* and the *Clear band selection* commands in the context menu.

In order to have fine control on specific band selection, we may use the *Set band subset* command from the context menu. Once selected, a dialog box appears that allows us to precisely define what bands should be added or removed:

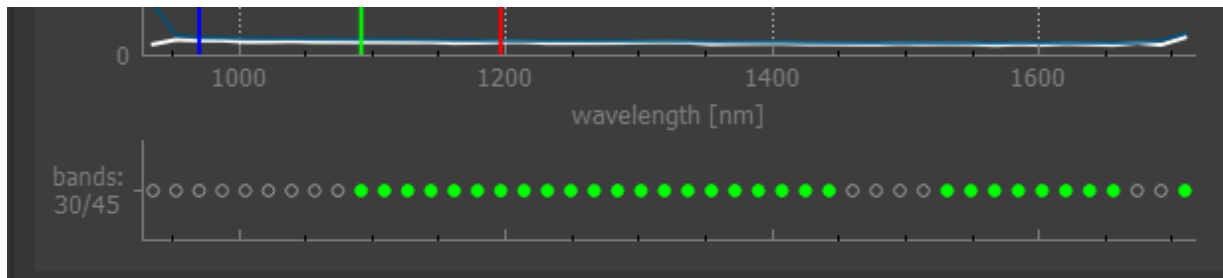


By default we may specify *band* indices. Alternatively, by selecting *Wavelengths* the selection happens on wavelength values in nanometers.

If we provide a range of bands, for example "10-30", we enable all bands starting with the 10th and ending with the 30th band:

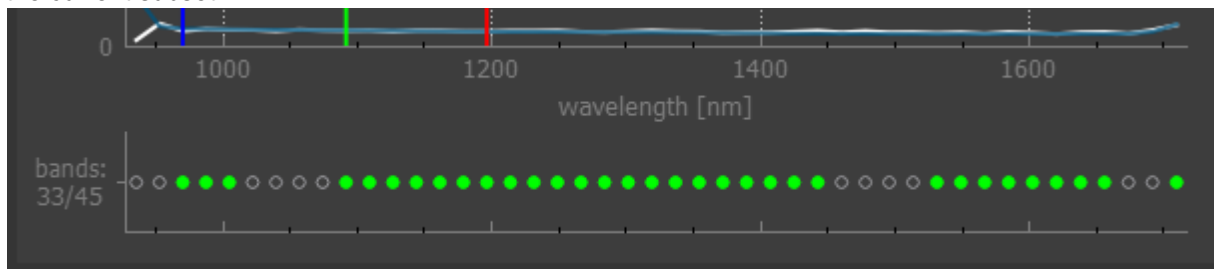


We may provide multiple regions at once separated by spaces. For example "10-30 35-42 45" will lead to these three regions:

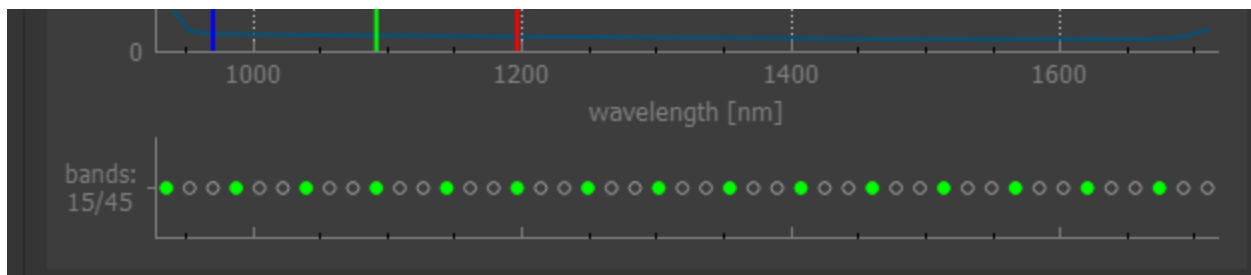


The default operation is to only enable the specified bands removing all previous state. Sometimes, we may wish to **add** or **remove** bands from the current selection. This is possible by prepending our specification with + (for adding) or - (for removing):

For example, selecting the *Set bands* command again and filling in "+3-5" will **append** three more bands to the current subset:



Sometimes, we may wish to select bands with a regular step. This is possible with "each X" syntax. For example, using "each 3" we get:

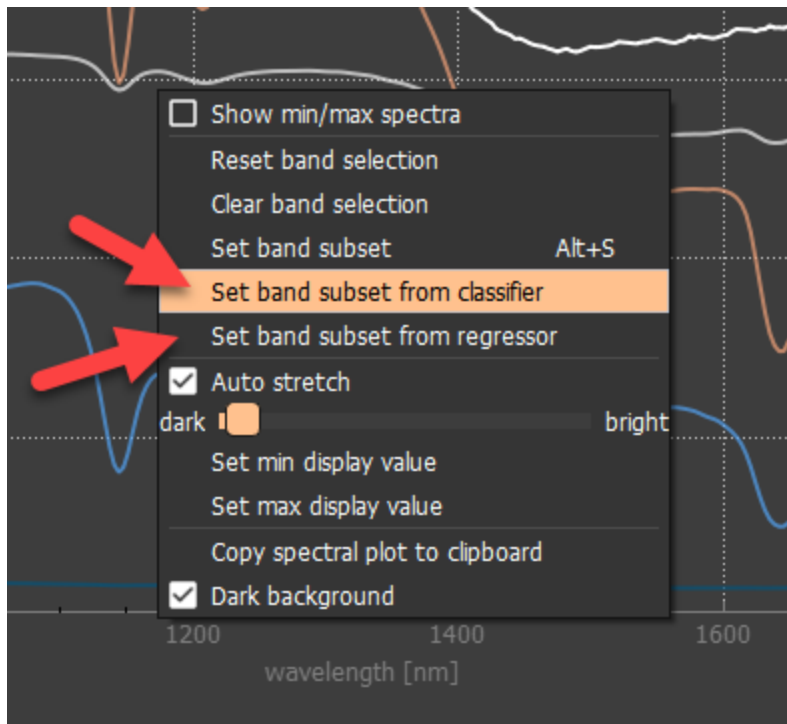


## Band subsets used by models

When training classification or regression model, the currently selected set of bands is used.

Subsequent changes in the band subset widget do not impact bands of the model, unless we retrain it.

We may, however, anytime return to the band subset used for the current classifier or regressor using the respective commands in the spectral plot context menu:



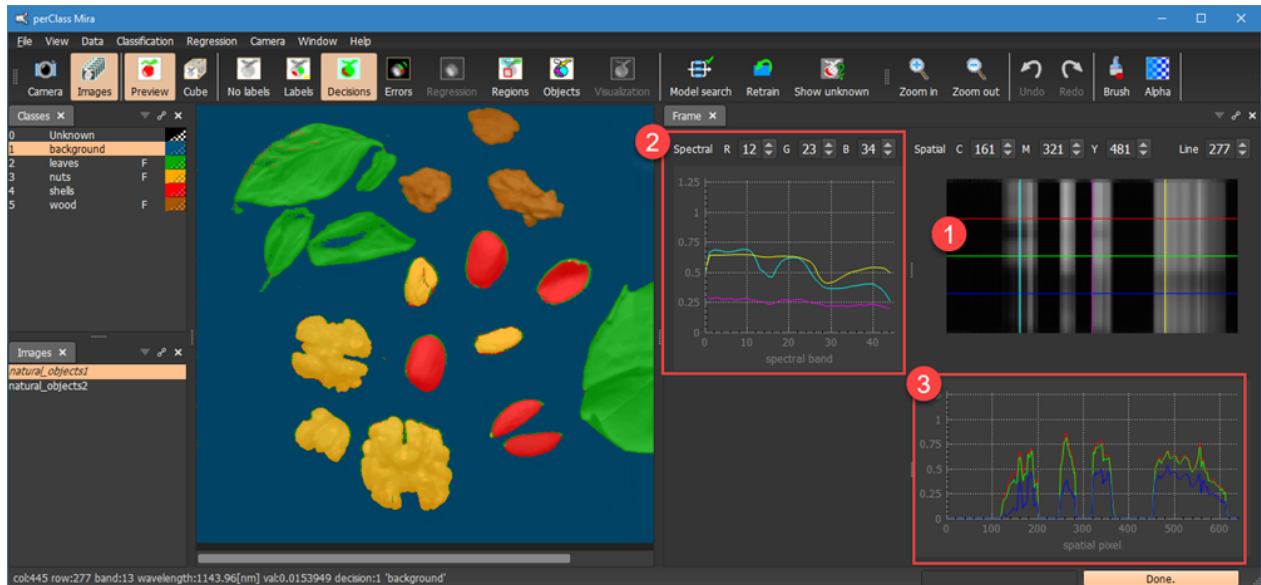
## Frame widget

Frame widget serves for detailed visualization of spectral cubes in both spectral and spatial direction. Its primary use is in live acquisition mode for line-scan applications. In this setup, it provides a logical view of the data transferred from a camera i.e. spectral frames. Each frame provides spectral responses for a line of spatial pixels. Frame widget is useful in acquisition to understand camera focus, visually judge scan borders, dead pixels or other artefacts.

Frame widget is enabled also for introspection of already loaded current spectral cube. Note the important conceptual difference: While in live acquisition mode, the frame widget shows raw spectral frames being acquired, in the off-line mode it visualizes content of the current cube that is typically already corrected into reflectance.

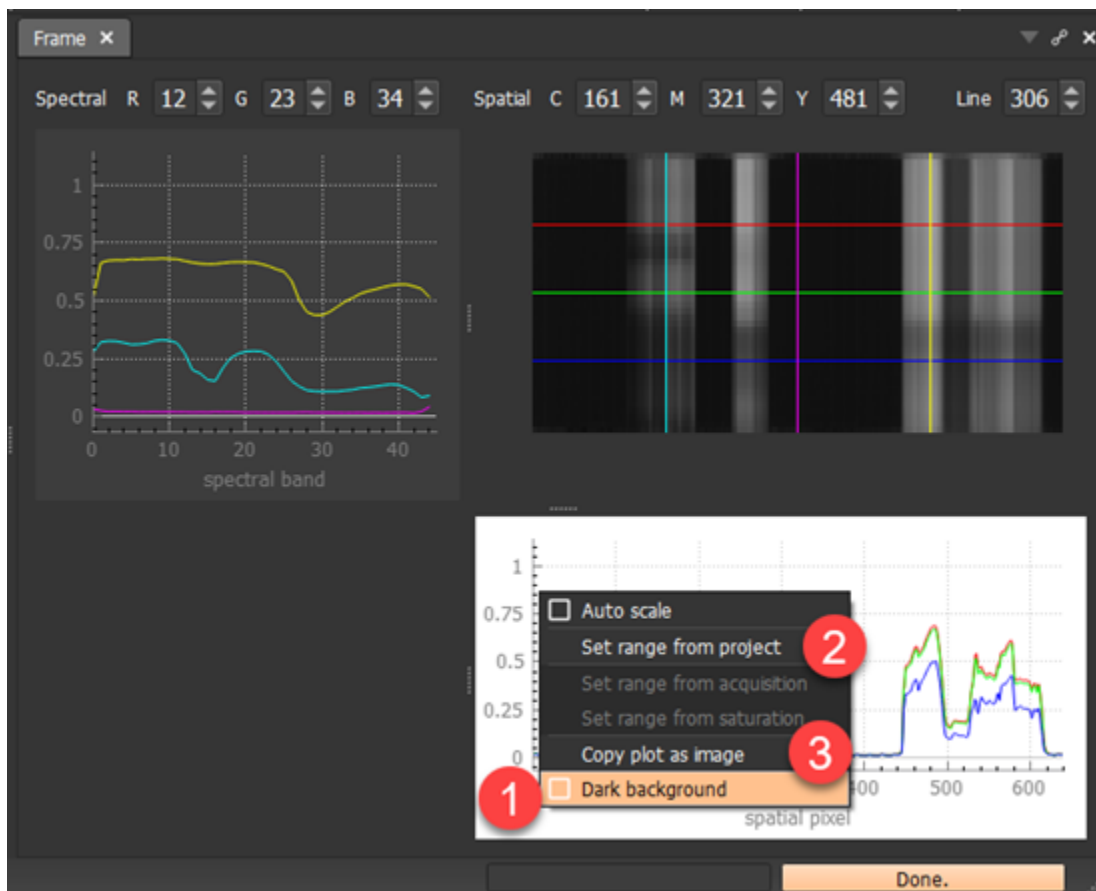
In the following screenshot, the frame widget shows one spectral frame in a loaded image. The area <sup>1</sup> shows the frame content. Horizontally, the spatial pixels are provided. Vertically, the spectral information is

displayed. The colored lines in the frame widget are then further visualized in the spectral plot <sup>2</sup> and spatial plot. The corresponding band and pixel indices are located on the top of the frame panel.



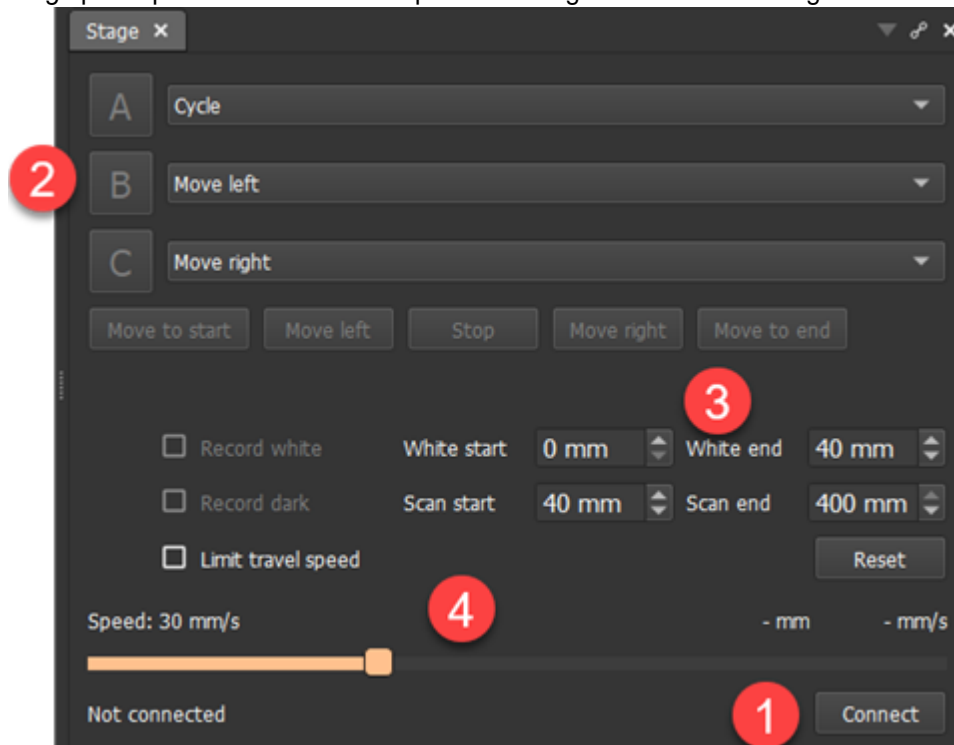
Context menu on the plots exposes number of options:

- 1 Switching between default dark and white background. White background may be beneficial when using the spatial profile plot to judge camera on a structured pattern
- 2 Several options for plot scaling. The *Auto scale* provides automatic stretch based on the data. *Set range from...* options allow the user to set axis span from project, acquisition (frame) or saturation.
- 3 The plots can be copied to clipboard as an image



## Stage

Stage panel provides full control of perClass Stage linear lab scanning device.



The stage needs to be first connected using the *Connect* button <sup>1</sup>.

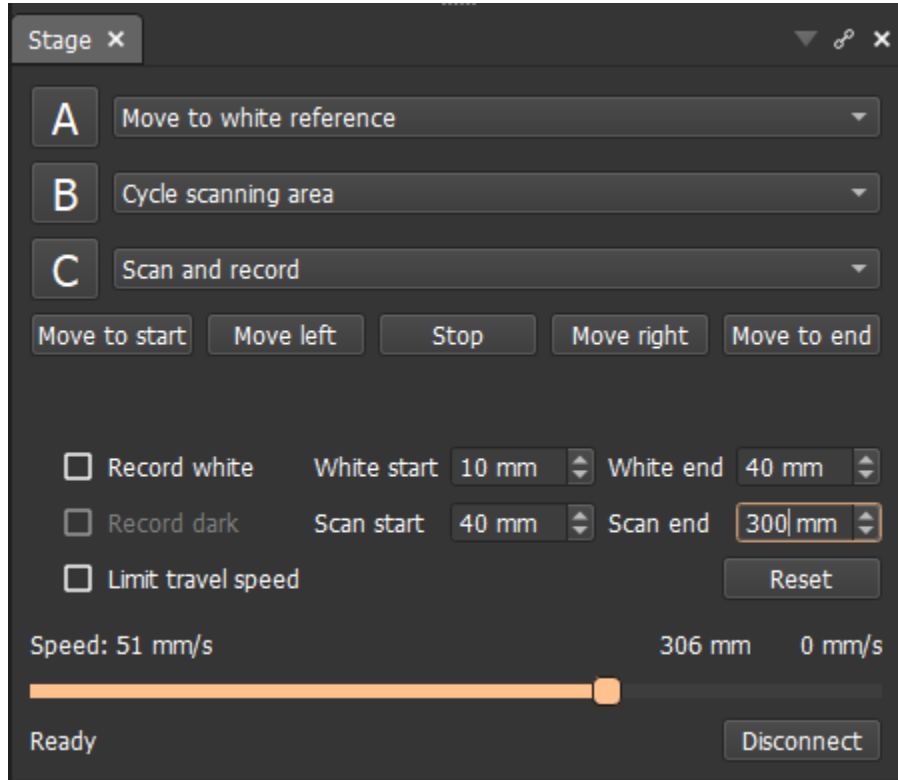


The section <sup>2</sup> contains three programmable buttons A,B and C available both on the *Stage* panel and as physical buttons on the stage.



The section <sup>3</sup> provides position limits for the reference (White) and the scan area. Current position and speed of the stage table can be read above the *Connect/Disconnect* button (after connection).

Finally, the section <sup>4</sup> provides speed control slider. It can be used anytime, also during the movement.



### Stage commands

The following stage commands are available:

- **Cycle** - cycle the entire table length (0-400mm)
- **Cycle scanning area** - cycle the scan area defined in the edit boxes. This is useful if your samples do not fill the entire length of the table
- **Move left / Move right / Stop** - movement commands. Note that they are also always present under the programmable buttons
- **Move to start / Move to end**
- **Move to center** - This command is useful to "park" the table in the central position before dismantling the stage
- **Move to white reference**
- **Scan and record**

Commands set by the user are restored in future sessions.

## Camera

Camera panel provides control of acquisition from a connected device. In order to use a camera, it is necessary to start the project with the respective acquisition plugin. Each such project is of "perClass" type. This is a major change from pre-4.2 perClass Mira releases where vendor specific projects provided acquisition control.

Camera panel displays speed information and plot with a separate line for the camera itself in cyan, "classifier" meaning full processing pipeline in red and the total processing time in green.



## Camera controls

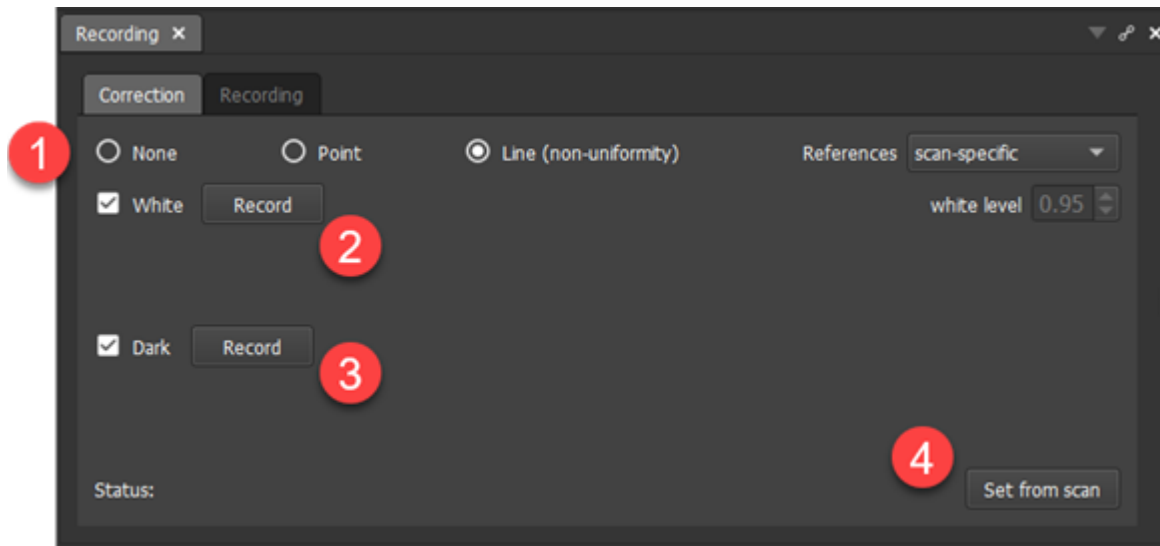
Camera controls are:

- **1** Exposure (integration time) in milliseconds. Exposure-control of the camera is assumed where exposure settings influences possible frame rate settings.
- **2** Frame rate in frames per second. If frame rate can be controlled by the user, it is provided together with the maximum possible frame rate. For some camera types, user cannot change the frame rate. Then, high default frame rate value is provided indicating that the device runs always at the maximum achievable frame rate for given exposure time.
- **3** Band control allows the user to select specific sensor band used to display data layer in the data visualization
- **4** Maximum raw display value control sets the limit for raw data visualization. It is a zero-based value. By default it is set to the maximum raw value - 1. When displaying image content for a single band, this setting makes it possible to visualize saturations by the red/yellow pattern.



## Recording panel

*Recording panel* provides control for recording references and data. By default, perClass Mira imposes data correction work-flow where dark and white references need to be acquired before data is recorded. The motivation is to make sure that modeling is performed not on raw but on reflectance-corrected data. This makes models more robust to illumination changes.



The radio button group <sup>1</sup> allows selection of correction work-flow. The default "*line (non-uniformity)*" means that full frame dark and white references are used correcting individually every pixel in the image.

The *White* <sup>2</sup> and *Dark* <sup>3</sup> references can be acquired from sensor by pressing the respective buttons. Alternatively, it is possible to set the referneces from already recorded scan, selected in the *Images* panel, using the button *Set from scan* <sup>4</sup>.

### Acquired references

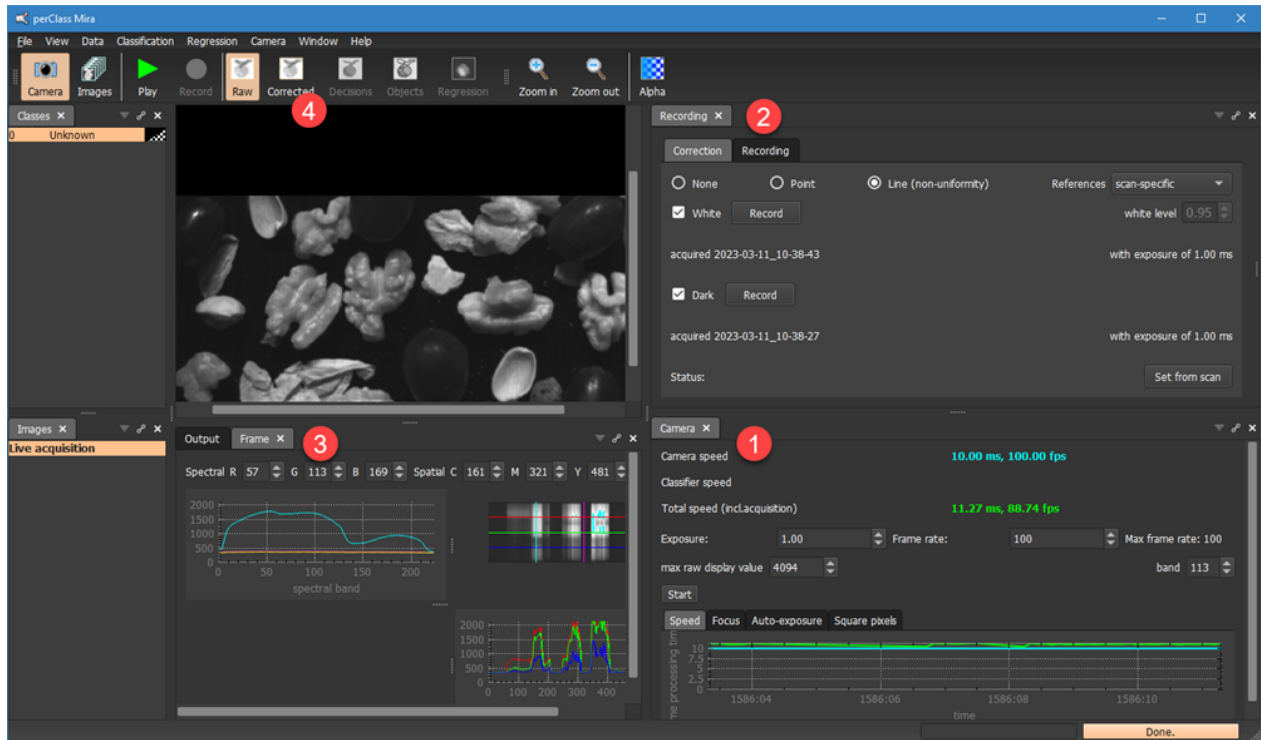
Once the references are acquired, their time-stamps <sup>1</sup> are and exposure times used <sup>2</sup> are listed. Note, that references should be acquired using the same exposure as the data. If you change exposure time, retake the references before acquiring data.

The *Recording* tab <sup>3</sup> becomes enabled once both references are available. For the correction work-flow without references (*None* radio button), the recording is always available.



## Recommended screen setup

Following screenshot illustrates the recommended screen organization for data recording.

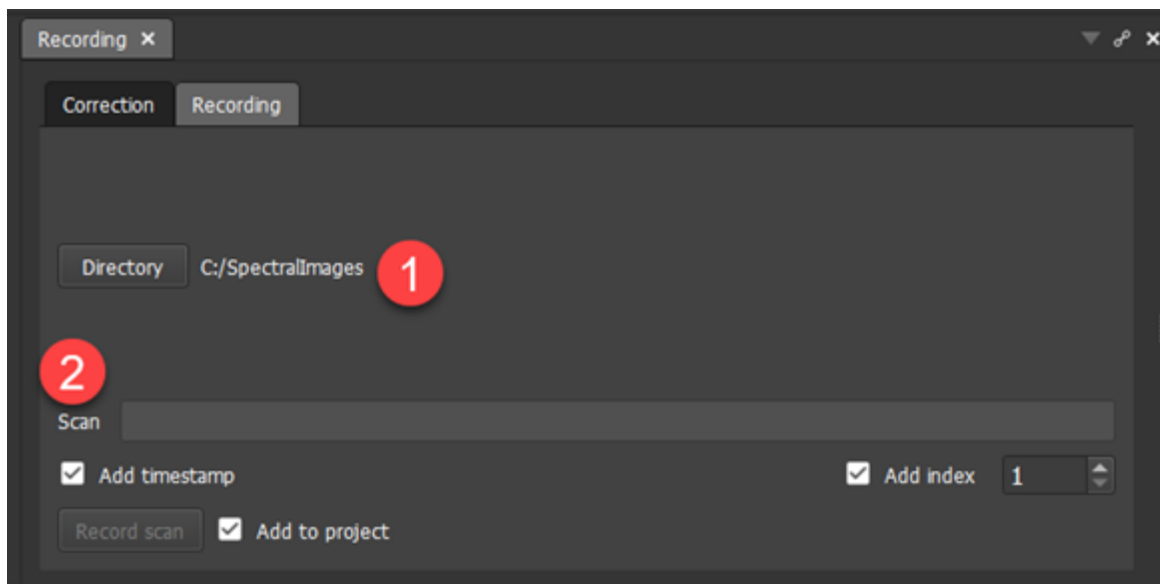


Both, the *Camera* (1) and *Recording* (2) panels are available above each other. For line-scan recordings, we recommend using also the *Frame* widget (3) under the data stream.

Note, that as references are recorded in this example, the main toolbar in the *Camera* mode shows not only *Raw* data button, but also *Corrected* data command (4).

## Setting scan name

Once references are defined in the *Correction* tab of the *Recording* panel, the *Recording* tab is enabled. It controls directory and filenames for scans to be stored.



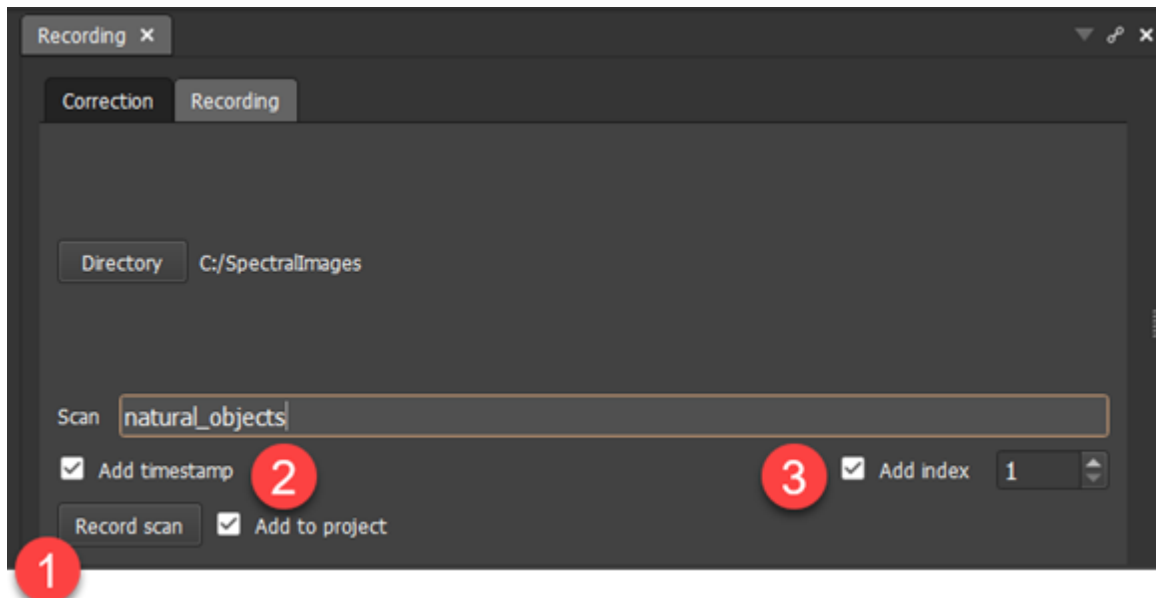
The directory button <sup>1</sup> specifies, where the recorded scans will be stored. This directory is identical to the project "Top-level data directory", that can be set from the *File* menu or from the context menu in *Images* list. The directory may be changed any time. Only the newly-recorded scans will be affected and placed in the new directory.

**TIP:** Note, that perClass Mira **never** deletes or moves any scans. It is user responsibility to perform any destructive actions on the data, if needed.

The scan field <sup>2</sup> defines the scan name to be recorded.

**Important:** Note that the scan name must be defined, in order to record a scan. When working with perClass Stage, issuing *Scan and Record* command will not work unless the scan name is defined.

Once, the scan name is defined, the *Record scan* button <sup>1</sup> is enabled. By pressing it, both camera acquisition and recording will be started. The user needs to terminate recording and acquisition using the same button <sup>1</sup>, now having the *Stop recording* label.



By default, time stamp <sup>2</sup> and index <sup>3</sup> are appended to the user-defined scan name. The index auto-increments after each recording.

**TIP:** When recording scans with training examples, it is recommended to use only examples of one class and use its name as a scan name.

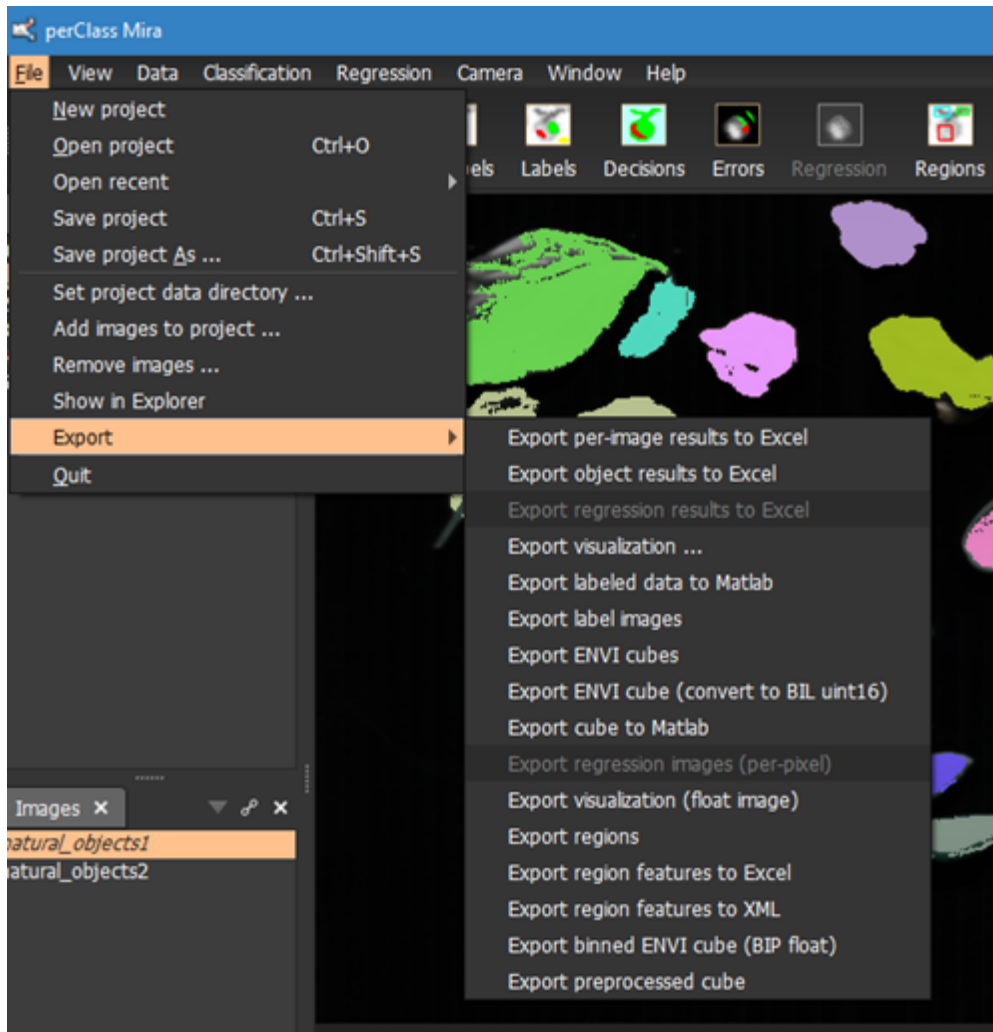
## Exporting

perClass Mira provides number of ways how to export data. Individual export commands are located in *File / Export* menu.

The following high-level export options are present:

- [Per-image results](#) - for further analysis
- [Per-object results](#) - for further analysis
- [Extracted features](#) - for further analysis
- [Visualizations](#) - for display
- [Visualizations as float images](#) - for further analysis / external model training

- [Cubes](#)
- [Labeled data](#)
- [Regions](#) - connected to [Region importing](#)

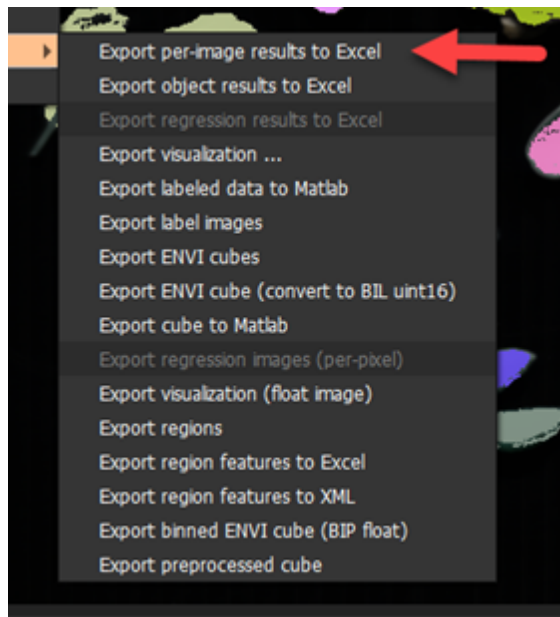


## Exporting per-image results

**The use-case for this export option is analyzing presence of certain important classes in many images.** For example, when detecting whether plants are infected by a disease, we segment a plant out of background and flag plant parts and infection as foreground classes. This export option allows us to quickly see whether the fraction of infection among all foreground pixels (representing a plant) is above acceptable limits.

In order to export results per-image, select desired images in *Images* list and use the *Export per-image results to Excel* command in *File / Export* menu



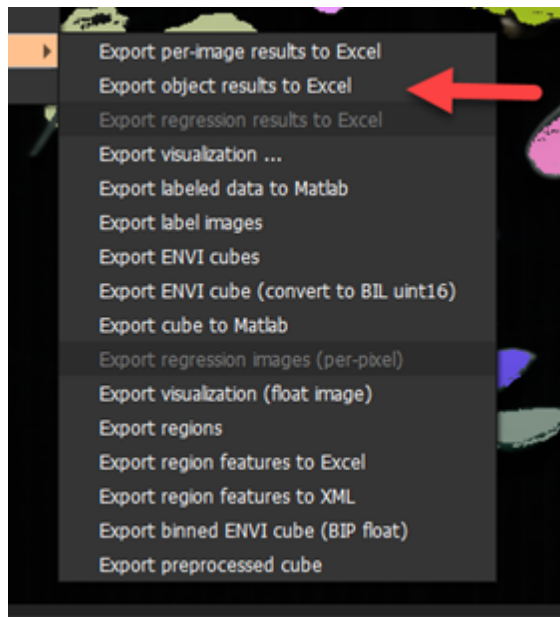


The resulting Excel file contains, for each image <sup>1</sup> the pixel count in each class <sup>2</sup>. Note, that foreground/background object flags are also provided. Additionally, for all foreground classes, their fractions within foreground are present <sup>3</sup>.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
1																	
2		version:	perClass Mira 4.2 (9-mar-2023)														
3																	
4																	
5			class names	background	leaves	nuts	shells	wood			background	foreground		leaves	nuts	shells	wood
6			class is foreground	FALSE	TRUE	TRUE	TRUE	TRUE	<sup>2</sup>		pixels	pixels	<sup>3</sup>				
7																	
8																	
9																	
10	index	image name	<sup>1</sup>														
11	1	natural_objects1		232337	43350	16614	7479	6517			247960	73960		0.586128	0.224635	0.101122	0.088115
12	2	natural_objects2		244622	37069	15262	6093	6139			259277	64563		0.574152	0.236389	0.094373	0.095085
13																	
14																	
15																	

## Exporting per-object results

This export option provides detailed information on detected objects including their position and per-object classifier decision. The use-case is building object classification solutions, for example, in sorting and grading applications.

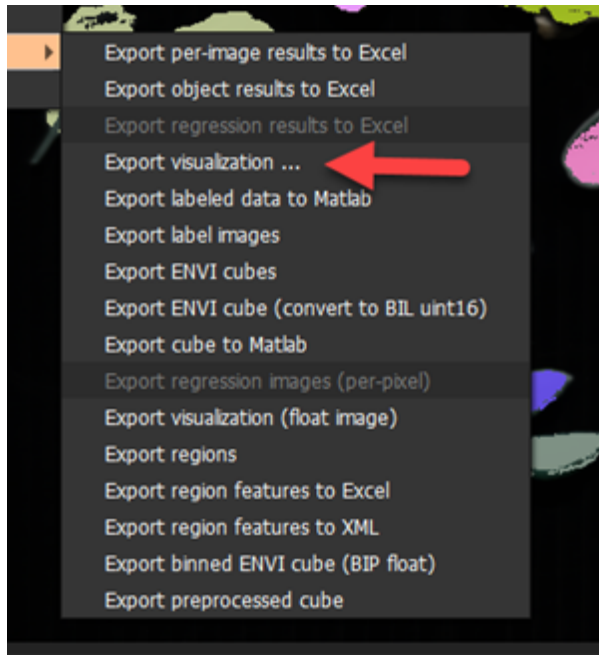


Exported results provide scan names <sup>1</sup>, training/test status in the project <sup>2</sup>, object size and bounding box <sup>3</sup>, per-object decision by perClass Mira object classifier <sup>4</sup> and break-down of pixel counts in all foreground classes <sup>5</sup>. The last can be used to define and validate custom object classification rules.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1															
2		version:	perClass Mira 4.2 (9-mar-2023)												
3															
4															
5															
6	index	image name	status	object index	size	col	row	width	height	object decision	foreground classes				
7	1	1 natural_objects1	training	1	1906	313	3	59	43	4	1:leaves	2:nuts	3:shells	4:wood	
8	1	1 natural_objects1	training	2	11575	41	21	190	105	1	9233	0	1	0	
9	1	1 natural_objects1	training	3	560	511	48	48	28	5	51	0	0	0	
10	1	1 natural_objects1	training	4	2327	231	72	64	53	4	0	0	0	1950	
11	1	1 natural_objects1	training	5	3566	333	73	91	67	4	1	0	0	2905	
12	1	1 natural_objects1	training	6	608	566	106	35	28	5	19	0	0	0	
13	1	1 natural_objects1	training	7	8060	72	118	181	97	1	6994	0	0	0	
14	1	1 natural_objects1	training	8	2611	309	144	50	68	3	99	0	1906	0	
15	1	1 natural_objects1	training	9	1795	238	175	42	58	2	81	1035	1	0	
16	1	1 natural_objects1	training	10	2499	388	177	68	53	3	55	0	1614	0	
17	1	1 natural_objects1	training	11	27559	441	200	181	201	1	26626	0	0	0	
18	1	1 natural obiects1	trainine	12	6169	109	242	95	89	2	19	4935	0	0	

## Exporting visualizations

**Exporting visualization generates color PNG images with the exact visualization content for all selected images.** The use-case is to batch process large number of data and create visual representation of a particular solution.



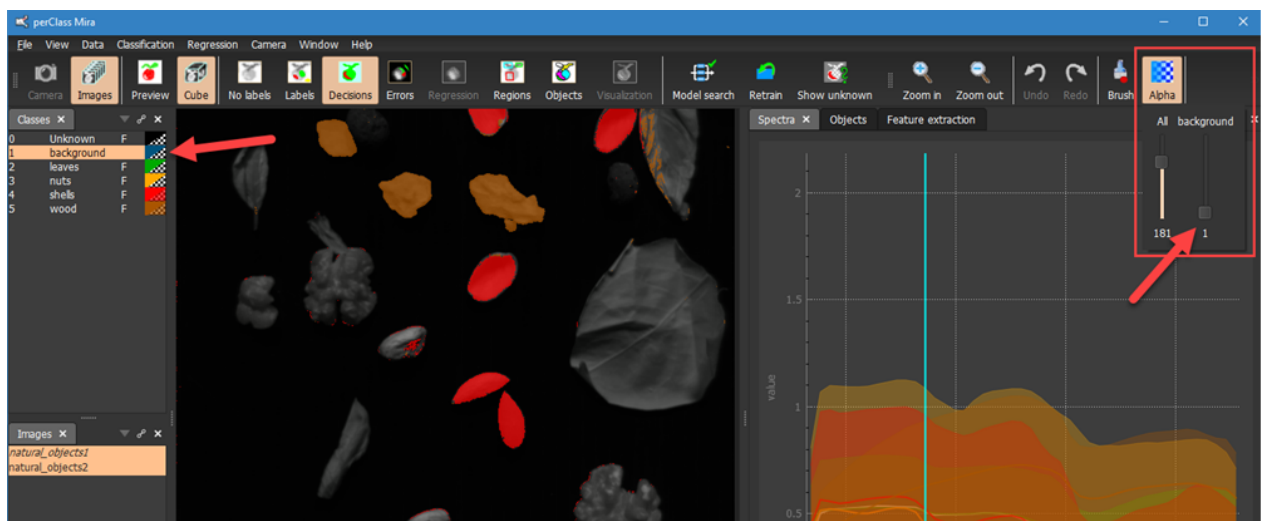
Select one or more images in *Images* list and use *File / Export / Export visualization...* command. A dialog box appears where the destination directory can be selected or a new one created. In addition, perClass Mira requests an optional suffix appended to image names. This is useful to distinguish multiple visualizations on the same set of scans or groups of scans (training, test, specific variety of a product etc.)

**TIP:** When working with many images, you may export visualizations and then get a quick view on many images using Windows Explorer view thumbnails feature.

## Visualization tips

### Highlighting only specific classes of interest

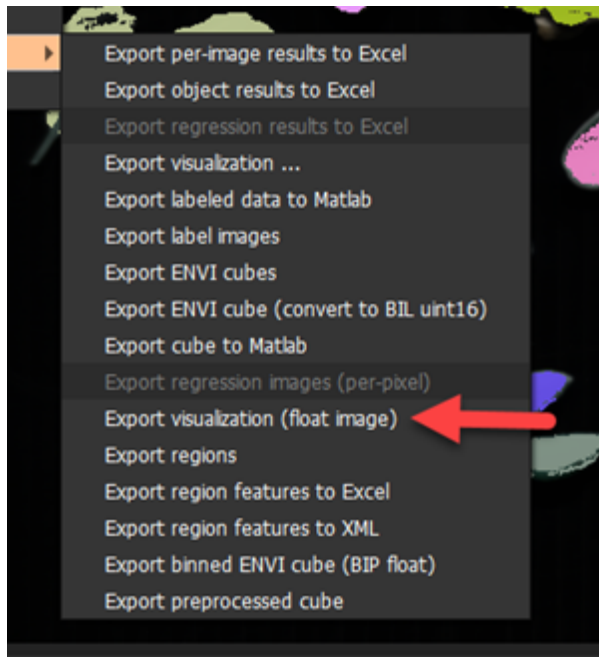
For quick visual identification of specific detections, it is convenient to control alpha layer per class and make unimportant classes fully transparent. This can be achieved by selecting specific class, using Alpha toolbar button class and adjusting the transparency only for this class. For stronger visual contrast, you may also make the background darker by adjusting the top of the *Spectra* plot.



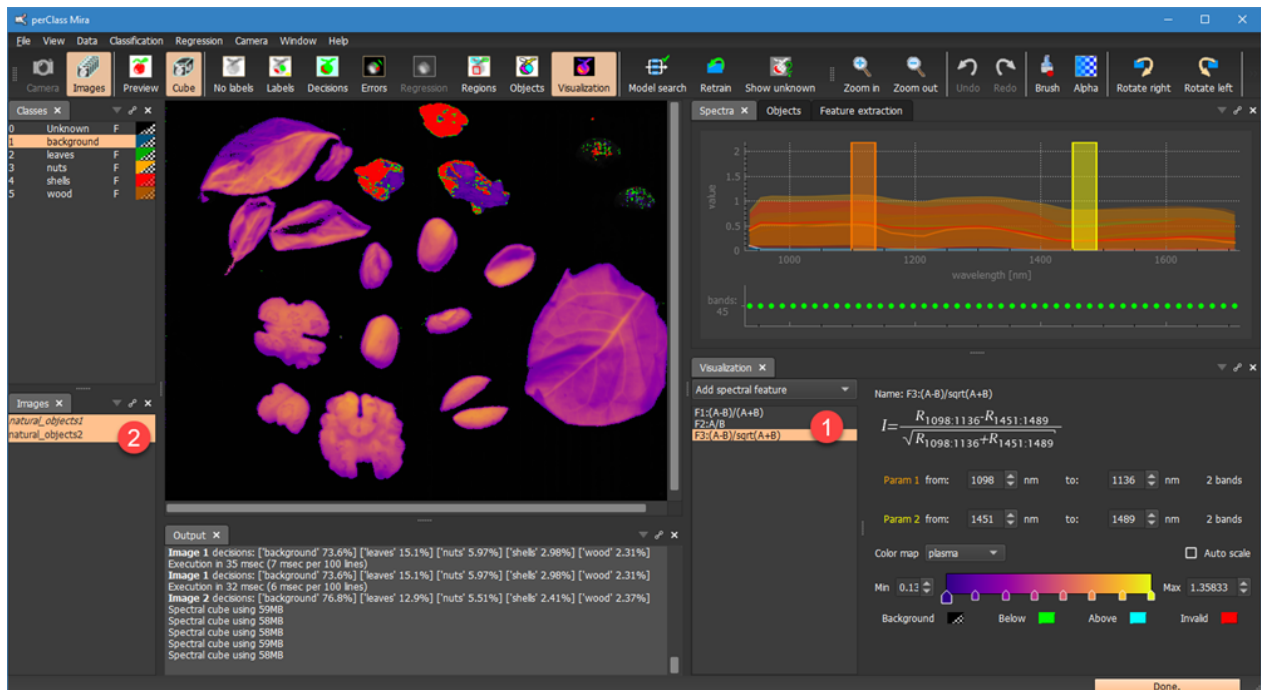
## Exporting visualizations as float images

By export visualizations as float images, we get data (Matlab .mat files) extracted from our spectral images. The use-case is to define one or more custom feature indices, export the floating point data

together with precise pixel labeling and perform further analysis or training of external models in Matlab, Python or other machine learning environment.



In the following example, we defined three [spectral indices](#) <sup>1</sup>. We select the scans we wish to process <sup>2</sup> and use *File / Export / Export visualization (float image)*. We may then select or create destination directory. For each selected image a .mat Matlab binary file is created with floating point spectral index content and separate pixel labels.



Example on Matlab side:

```
>> ls
```

```
. ..
```

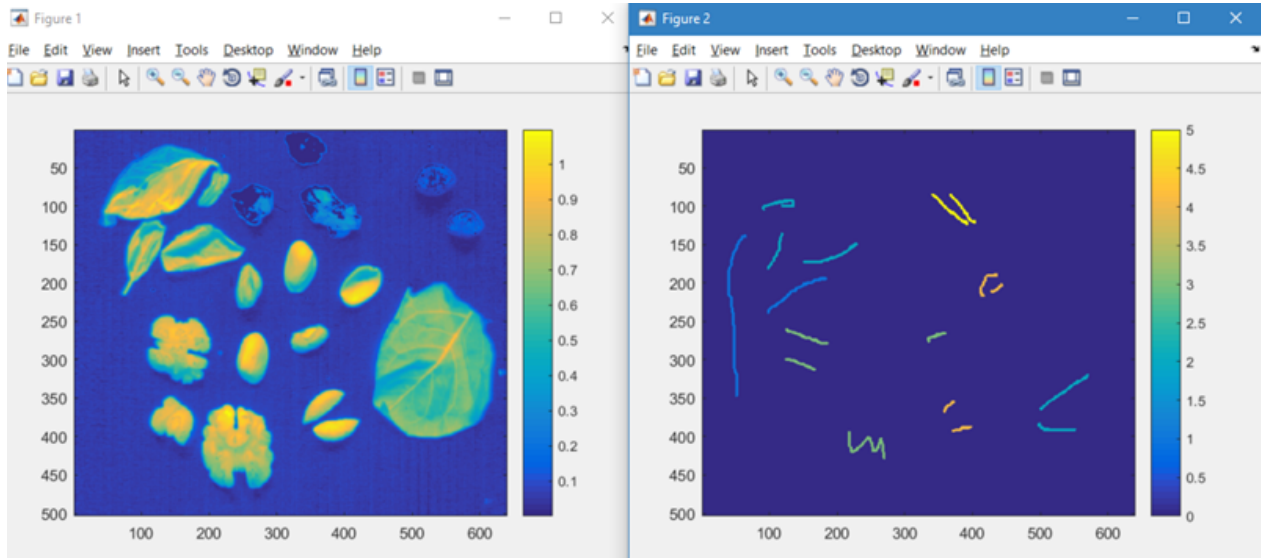
```
natural_objects1.mat natural_objects2.mat
```

```
>> load natural_objects1.mat
```

```
>> whos
      Name      Size      Bytes  Class  Attributes

      cube      640x503x3    3863040  single
      lab        640x503     321920  uint8
```

```
>> figure; imagesc(cube(:,:,3)')
>> figure; imagesc(lab')
```

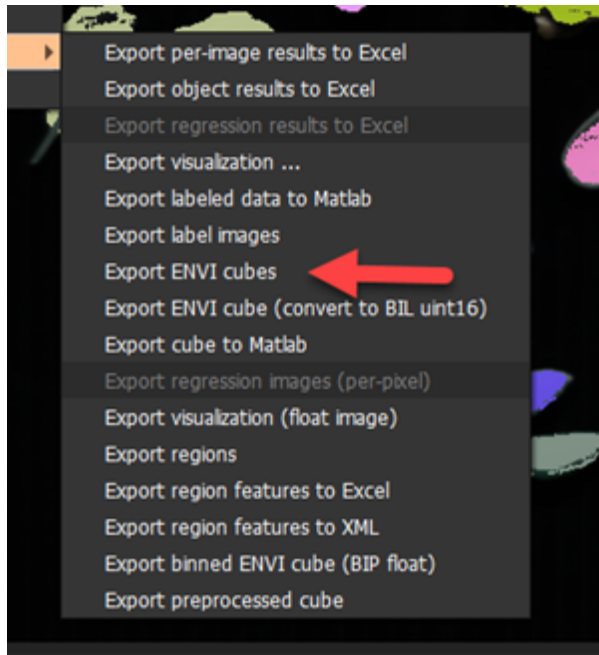


#### Comments:

- for each exported image a .mat file is present in the destination directory
- each of the files contains a cube and lab variables, respectively
- the cube variable contains a band (3rd dimension) for each of the spectral indices
- note, that we transpose the image content using ' operator to visualize images in the same way as in perClass Mira
- the lab variable contains per-pixel labels defined in perClass Mira. Class indices may directly to the class list in perClass Mira. Zero is the "unknown" - such labels are not present.

### Exporting cubes

**Exporting ENVI cubes** provides a convenient way to get simple ENVI cube representation of images in perClass Mira workspace. The use-case is further analysis of the images already converted to reflectance.



Upon selecting the export command, we may specify or create a destination directory and specify optional scan name suffix. This is useful to provide extra information on exported group of images. For example, to distinguish test scans or specific product variety.

For each scan an ENVI cube is created saving two files, namely the ENVI data cube with .bin extension and the text .hdr header file.

Notes:

- Images are exported in exact same representation as in perClass Mira workspace. This means that, for project types converting scans on load to reflectance, the exported images are stored as reflectance (typically BIP layout, float data type)
- For images, that are cropped in perClass Mira workspace, only the crop area is exported, not the full original image. Therefore, we may use this export type to conveniently focus only on relevant parts of the scans

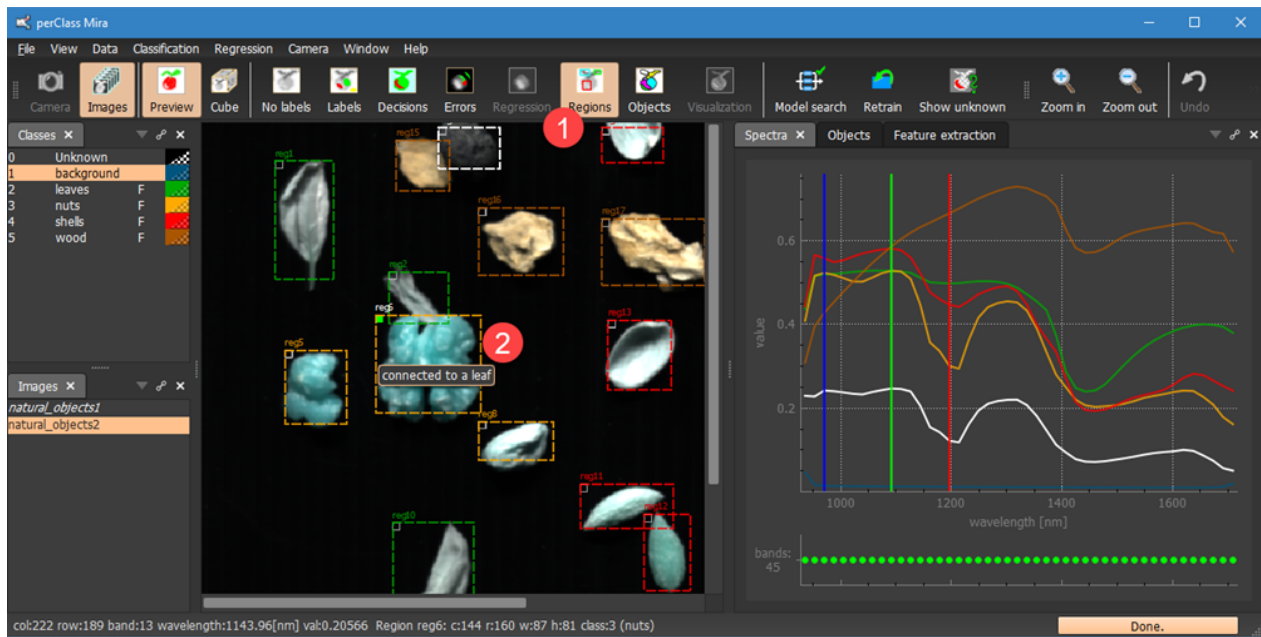
## Exporting regions

**Exporting regions** enables us to store information on *Region* definition outside of perClass Mira project. Possible use-cases are:

- performing analysis of object classification results in Excel or other software
- precise definition of regions outside perClass Mira. This is connected to additional command Import of regions into perClass Mira from Excel file

Example exporting regions defined on a scan to Excel: In the *Regions* mode <sup>1</sup>, we can see regions defined. Note, that one of the regions <sup>2</sup> has a text note attached.





After using the *Export regions* command, we obtain the following Excel file:

For each exported scan <sup>1</sup>, each region is given including its name <sup>2</sup>, bounding box <sup>3</sup>, class <sup>4</sup> and optionally also notes <sup>5</sup>.

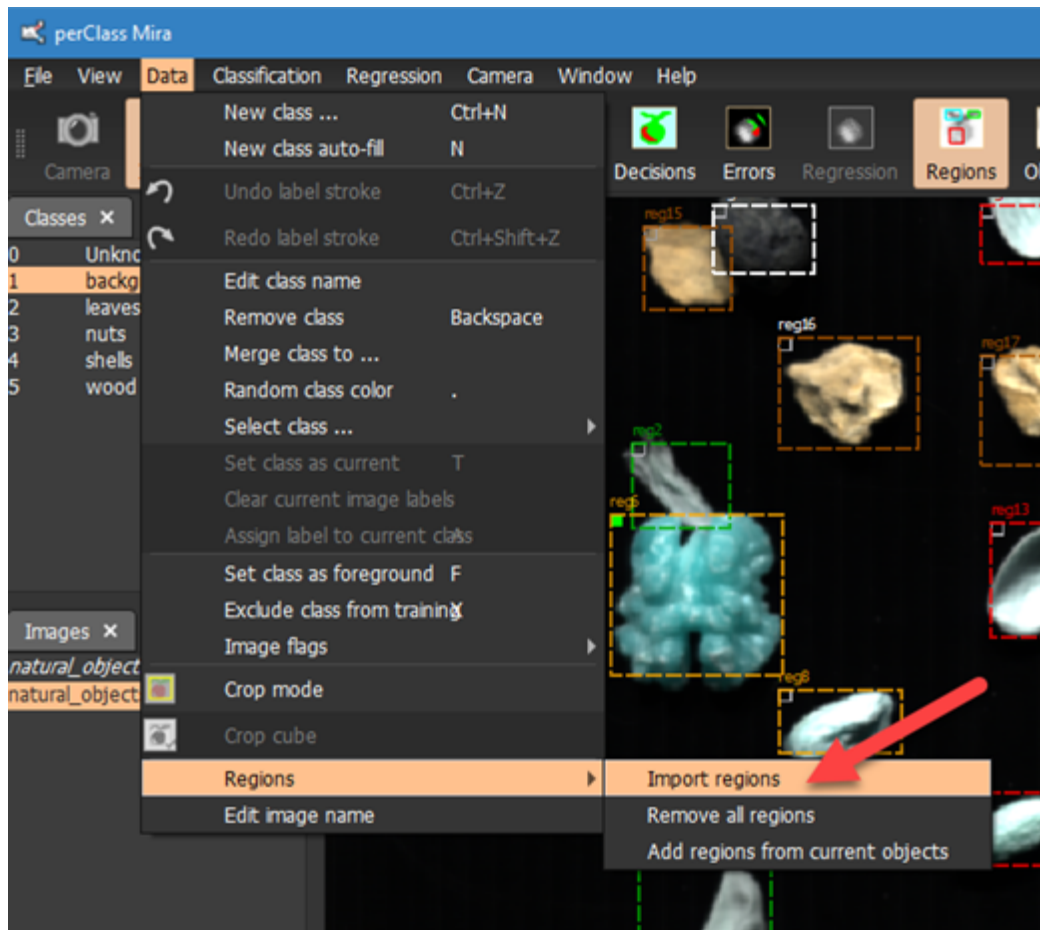
	A	B	C	D	E	F	G	H	I
1									
2	version	perClass Mira 4.2 (9-mar-2023)							
3	format	regions							
4									
5		object							
6	image name	name	column	row	width	height	class	notes	
7	natural_objects2	reg1	62	33	48	98	leaves		
8	natural_objects2	reg2	156	125	49	43	leaves		
9	natural_objects2	reg3	448	156	180	193	leaves		
10	natural_objects2	reg4	532	3	79	118	leaves		
11	natural_objects2	reg5	70	190	51	61	nuts		
12	natural_objects2	reg6	145	161	87	81	nuts	connected to a leaf	
13	natural_objects2	reg7	449	409	108	94	nuts		
14	natural_objects2	reg8	230	249	62	32	nuts		
15	natural_objects2	reg9	261	458	57	31	nuts		
16	natural_objects2	reg10	159	333	67	114	leaves		
17	natural_objects2	reg11	314	301	77	37	shells		
18	natural_objects2	reg12	367	326	38	63	shells		

## Importing regions

The **use-case for region importing** is precise definition, for example, when creating regular region grids in plant phenotyping or using external labeling and annotation sources.

In order to import region definition from an Excel file, use *Data / Regions / Import regions* menu command:



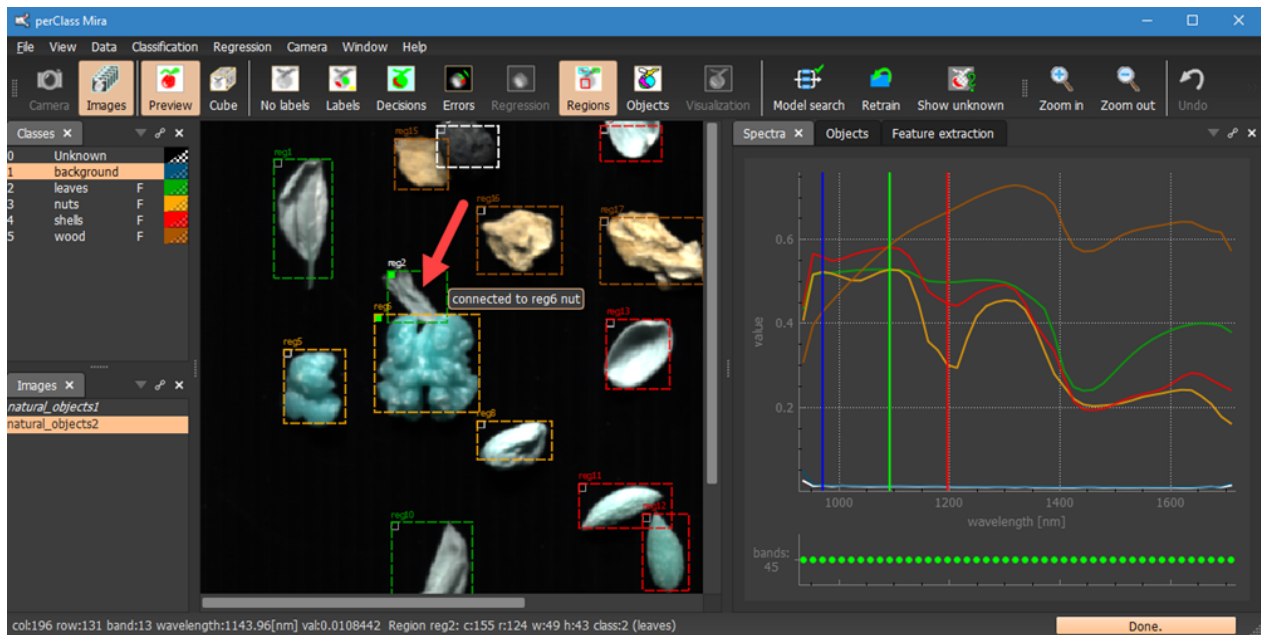


Extending our [export regions example](#), we now include an extra note to reg2 region in Excel and save the file.

	A	B	C	D	E	F	G	H	I	J
1										
2	version	perClass Mira 4.2 (9-mar-2023)								
3	format	regions								
4										
5		object								
6	image name	name	column	row	width	height	class	notes		
7	natural_objects2	reg1	62	33	48	98	leaves			
8	natural_objects2	reg2	156	125	49	43	leaves	connected to reg6 nut		
9	natural_objects2	reg3	448	156	180	193	leaves			
10	natural_objects2	reg4	532	3	79	118	leaves			
11	natural_objects2	reg5	70	190	51	61	nuts			
12	natural_objects2	reg6	145	161	87	81	nuts	connected to a leaf		
13	natural_objects2	reg7	449	409	108	94	nuts			
14	natural_objects2	reg8	230	249	62	32	nuts			
15	natural_objects2	reg9	261	458	57	31	nuts			
16	natural_objects2	reg10	159	333	67	114	leaves			
17	natural_objects2	reg11	314	301	77	37	shell			

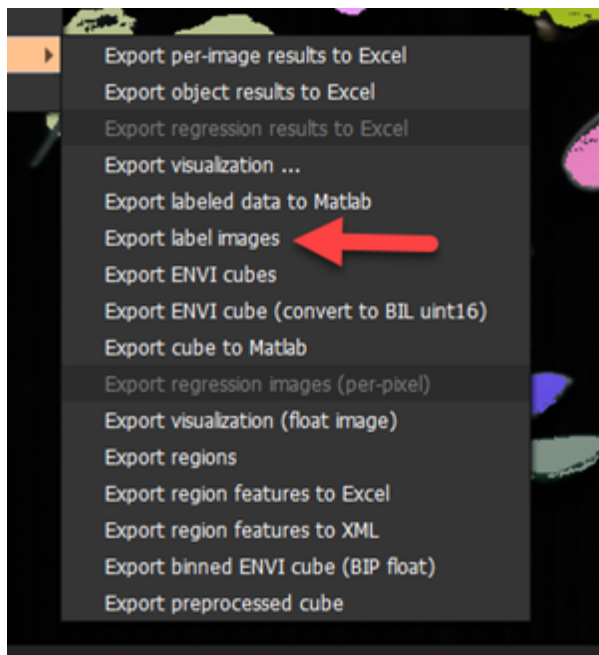
- In order to import regions, we need to first remove existing ones using *Data / Regions / Remove all regions* command (when specific scan or scans to be affected are selected in the *Images* list). Note, that this follows general perClass Mira design principle of not destructing any information behind user's back.
- After removing the regions, we can use the *Data / Regions / Import regions* command, point to the updated Excel file

The new region definition now contains text note also for the region reg2. In this same way, we may create entirely new regions or change positions and class assignments of existing ones.

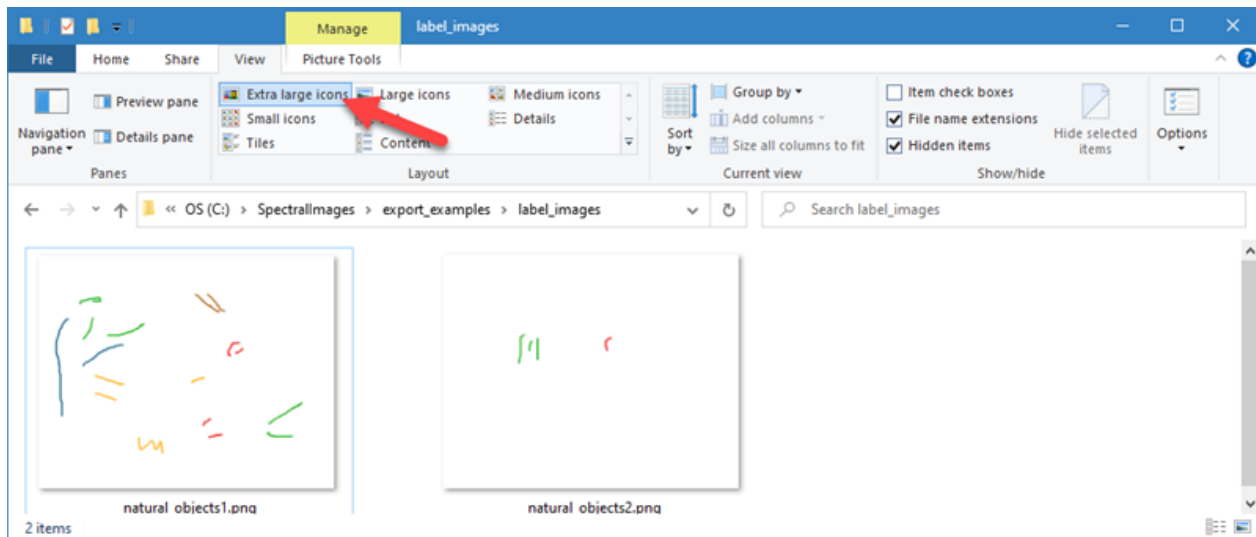


## Exporting label images

**Exporting label images** provides label masks as PNG files including class names meta-data. This is useful when using perClass Mira as a precise annotation tool for external machine learning training work-flow.



A destination directory can be selected or created. Optional scan name suffix can also be specified. This is useful to distinguish different groups of scans, for example, due to their training/test status or product variant. We may view the exported images conveniently in the Windows Explorer

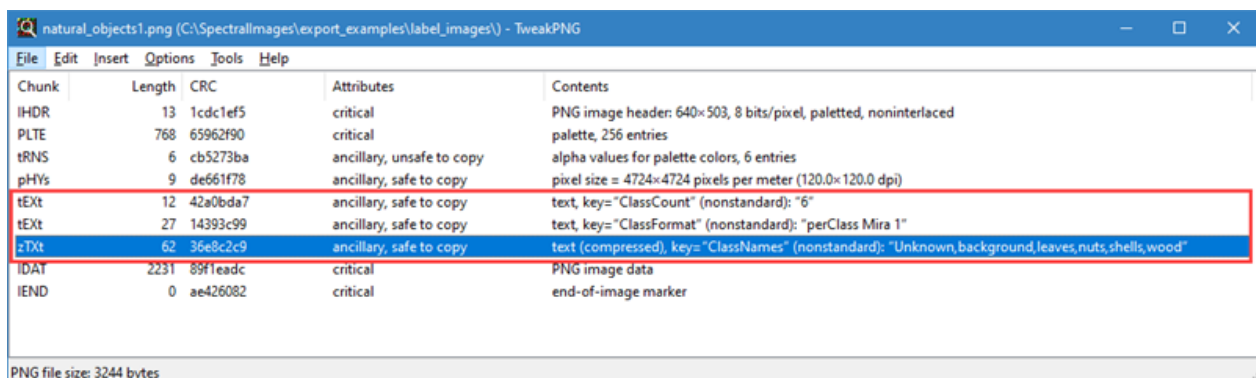


### ***Class name meta-data***

The exported images contain class name meta-data.

### ***Accessing meta-data using TweakPNG***

Free TweakPNG utility can be used to view PNG file meta-data: <http://entropymine.com/jason/tweakpng/>



### ***Accessing meta-data in Matlab***

Using `imfinfo` command in Matlab, we can access the meta-data information, stored by perClass Mira. We need to extract the "OtherText" property:

```
>> s=imfinfo('natural_objects1.png');
>> s.OtherText
```

ans =

3x2 cell array

```
'ClassCount' '6'
'ClassFormat' 'perClass Mira 1'
'ClassNames' 'Unknown,background,leaves,nuts,shells,wood'
```

## **Model testing**

Machine learning models are trained on annotated data. In perClass Mira, the concept of testing is strictly referring to evaluation of model performance on unseen examples. We would like to stress, that data used for testing should never be comprising the same of very similar physical objects as the ones used for model

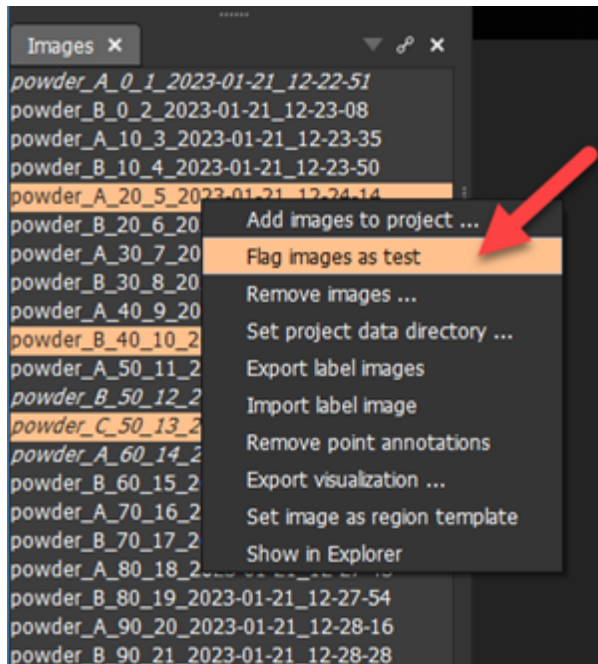
training.

In perClass Mira, images can be flagged for testing. This means, that any subsequent model retraining will not use these images for any of the steps.

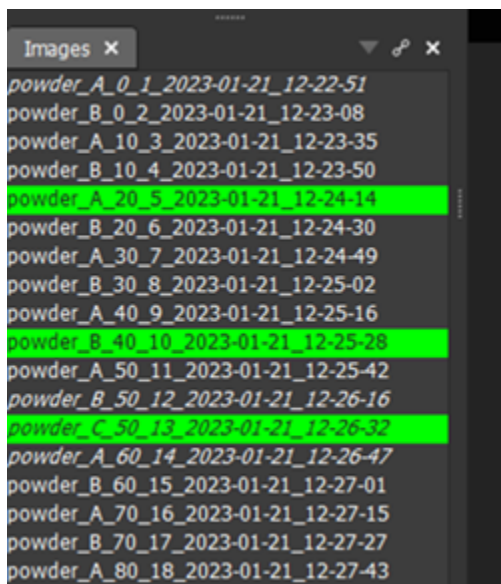
**NOTE** that flagging an image only does not change already existing models. The user needs to explicitly retrain a model or perform new model search in order for the new image flags to take effect.

## Flagging images for testing

Images can be flagged for testing using context menu in *Images* list and the *Flag images as test* command:



Selected images will be assigned a test flag which is reflected in their green color:

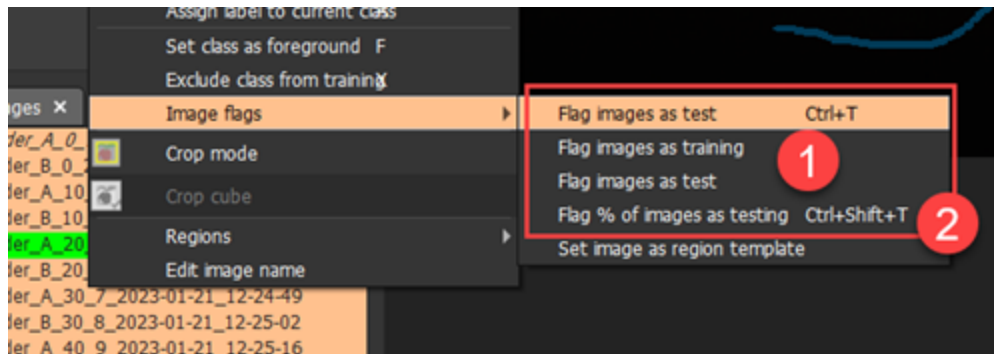


**TIP:** Ctrl+T keyboard shortcut performs the same action.

Note, that the state of the selected images changes by applying this command. Therefore, images that were already part of the test set will become part of the training set (will lose the green color emphasis).

## Additional image flagging commands

Data menu contains several additional commands for image flagging in *Image flags* sub-menu:



- 1 Selected images may be all flag for training or for testing
- 2 Percentage of of selected images can be randomly assigned a test flag. This is useful to perform manual cross-validation on a large number of scans. We select all scans (Ctrl+A) and flag random fraction as test. We rebuild a model and record test set performance. Note, that in order to retrain a model, we need to have labeled data. Therefore, most of your scans should be meaningfully annotated in order to use this manual cross-validation approach.

## Cross-validation

perClass Mira provides a convenient *Cross-validation* tool significantly simplifying statistical validation of models.

### What is cross-validation?

Cross-validation is a procedure of repeated retraining of a model and re-testing on different fractions of the data. **The goal is to assess variability of model performance on a given problem.** Repeated testing provides us with performance estimates accompanied with their respective standard deviations. Cross-validation allows us to compare two machine learning models at certain significance level. In other words, it allows us to conclude whether one model performs significantly better than other.

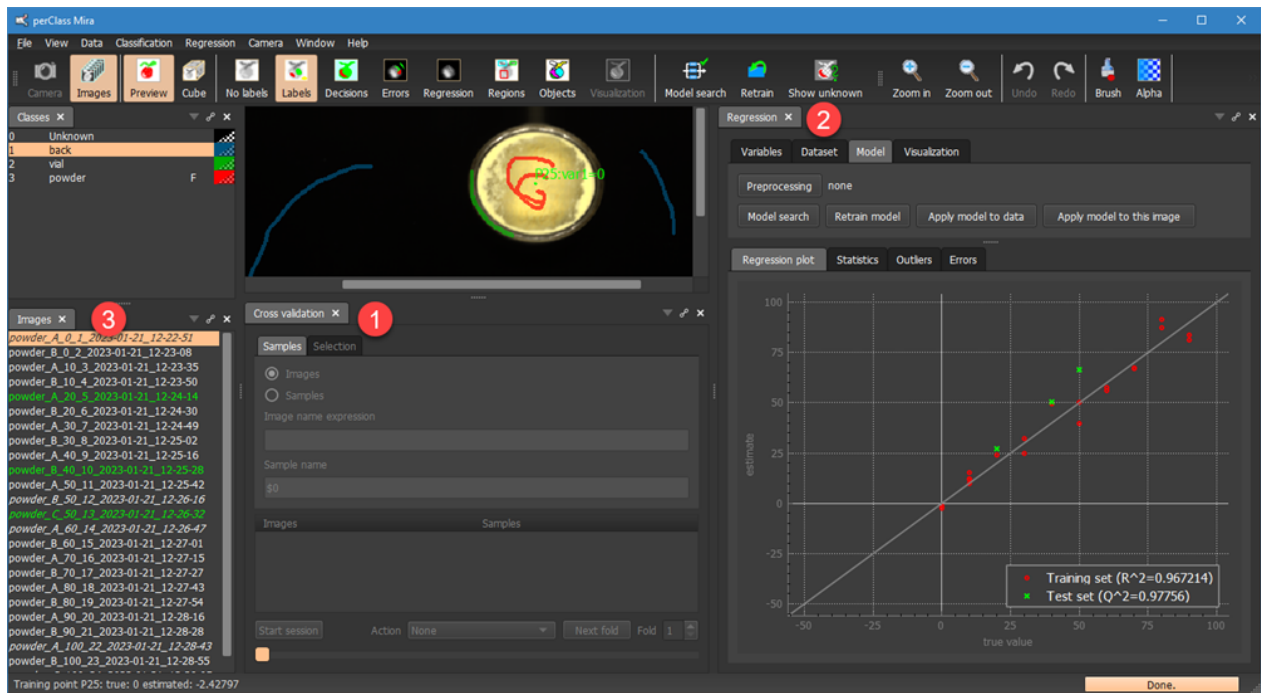
### Cross-validation in perClass Mira

The cross-validation tool in perClass Mira can support classification, regression, or custom external analysis work-flows. We will explain *Cross-validation* tool based on a regression example. We are estimating mixing proportion of two powders using regression modeling. We have placed the *Cross-*

*validation* panel 1 in the center under the image view. On the right side, we have a *Regression* panel

2 with results of a model, trained on a set of scans 3. Note, that three images were [manually flagged as testing](#). Therefore, we observe three green points in the *Regression plot*.





For this example, it is useful to clarify the scanning task and the structure of file names. We have a set of plastic vials with mixed powder. Each vial containing one specific mixing proportion. We scan each vial

alone multiple times. Each time, we record which mixing proportion the vial contains by an integer (1)

below. The letter (2) denotes which *replica* scan of the same vial we have acquired. Replicas A, B, and C mean three repeated scans of the same powder container,

(2) (1)

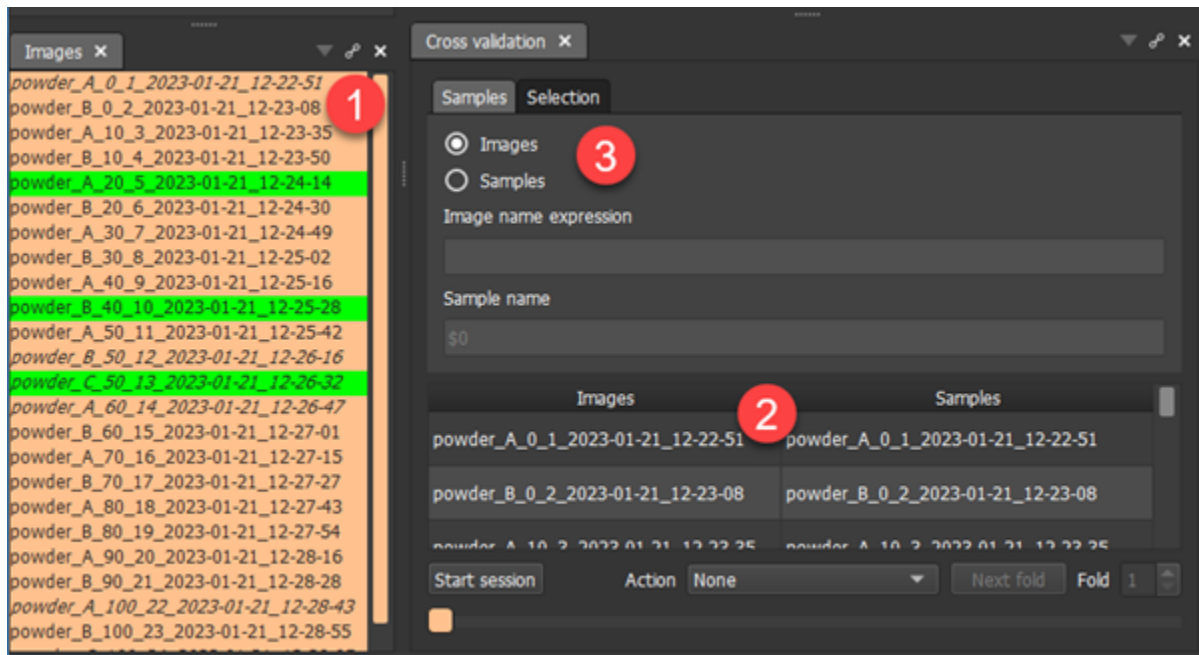
```

powder_B_30_8_2023-01-21_12-25-02
powder_A_40_9_2023-01-21_12-25-16
powder_B_40_10_2023-01-21_12-25-28
powder_A_50_11_2023-01-21_12-25-42
powder_B_50_12_2023-01-21_12-26-16
powder_C_50_13_2023-01-21_12-26-32
powder_A_60_14_2023-01-21_12-26-47
powder_B_60_15_2023-01-21_12-27-01
powder_A_70_16_2023-01-21_12-27-15
powder_B_70_17_2023-01-21_12-27-27
powder_A_80_18_2023-01-21_12-27-43
  
```

In order to use the cross-validation, we need to make a selection of images. In most cases, we wish to cross-validate on all images in the project. Therefore may select all using the Ctrl+A keyboard shortcut.

When an image selection (1) is created, the *Cross-validation* panel will fill the selected images in a table

(2) and its controls (3) will become enabled.



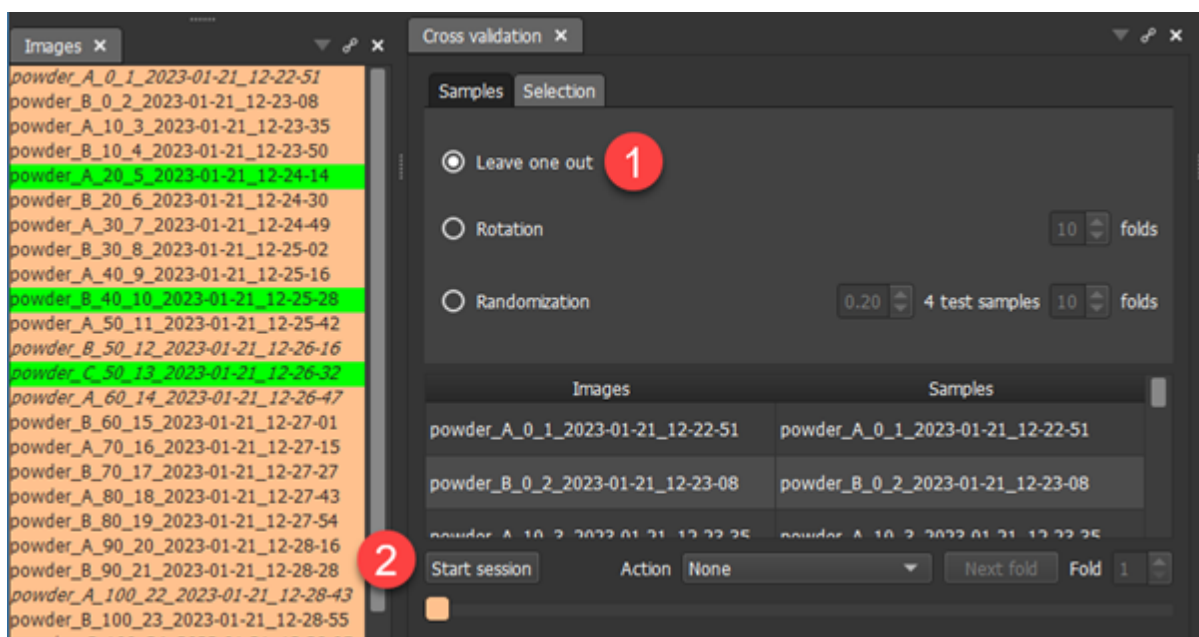
We will first explain default cross-validation over images.

### Cross-validation over images

This section explains cross-validation over images (as selected in the *Samples* tab of the *Cross-validation* panel).

In the *Selection* tab, we have three choices for common cross-validation strategies:

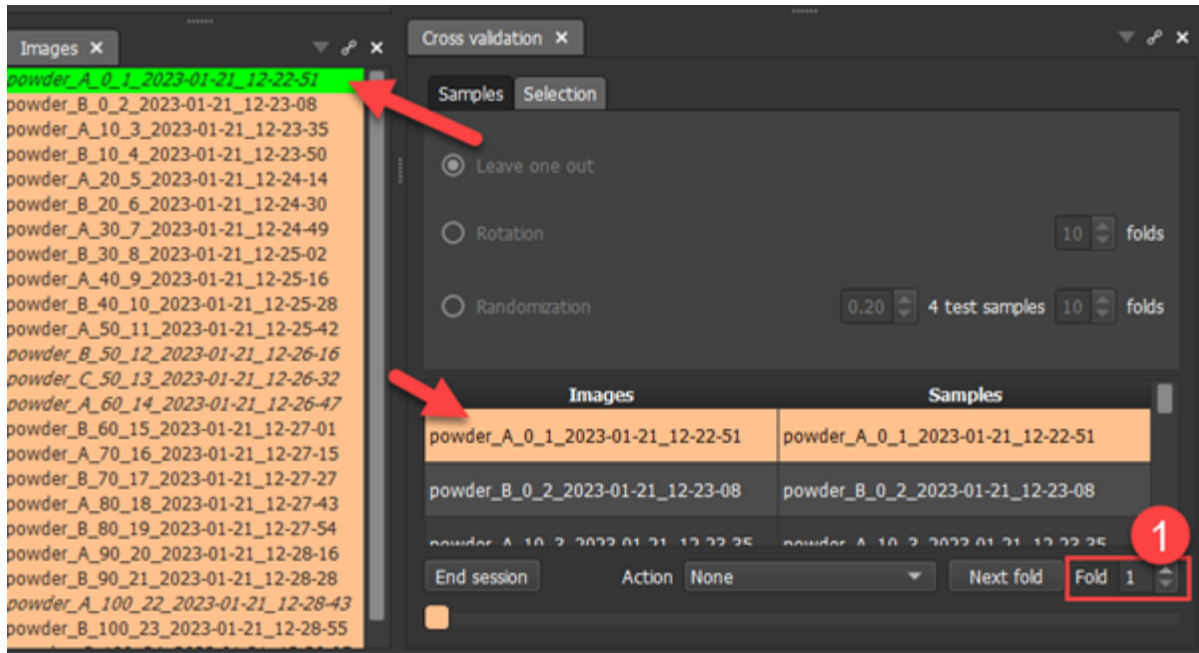
- **Leave-one out** - in this setup a single item defined in *Samples* (an image in this section) is left for testing and all others used for training.
- **Rotation** - Here a random splitting of images is performed first, followed by definition of smaller image groups called folds. In each fold, one group is used for testing and all remaining for training. Note, that images in each fold are tested only once in the rotation scheme
- **Randomization** - In this setup a random subset of a user-defined percentage is used for testing and all remaining samples for training. This process is repeated fold-times. The major difference from Rotation is that items in the test (images) may be used for testing multiple times.



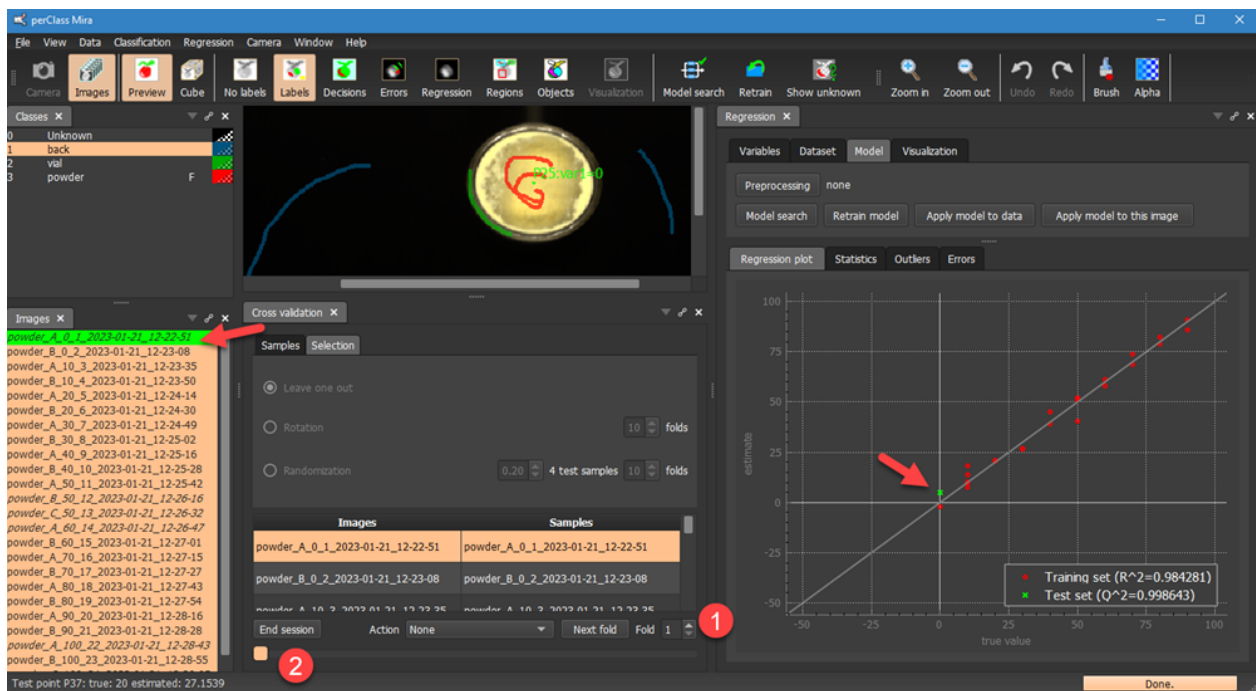


To start the cross-validation session, we click on *Start session* button <sup>2</sup>.

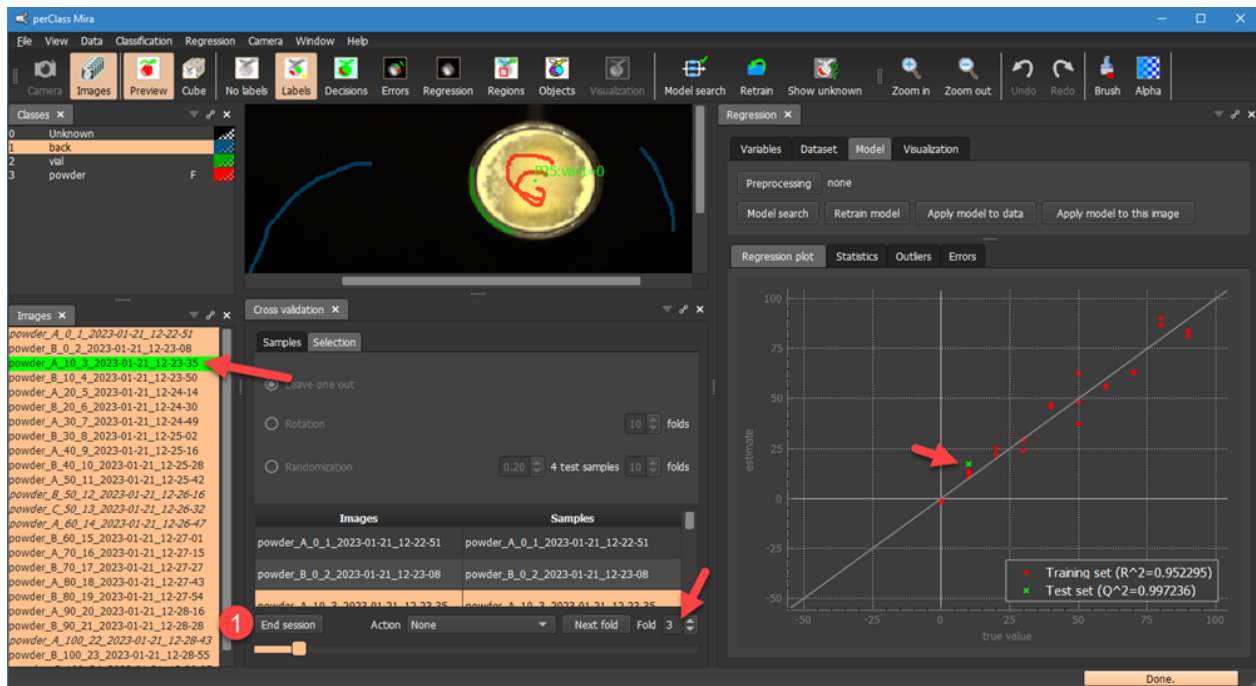
The first fold of a leave-one-image-out scheme will look like as follows:



We may now perform whatever model building action we like with standard perClass Mira tools. For example, run a model search in the *Regression* tool. We will observe a single test object:



To move to another leave-one-out fold, we may use either the fold spinbox <sup>1</sup> or the slider <sup>2</sup>. We are free to jump to any fold we like. For example, jumping to the 3rd fold and re-running model search, we will observe the following:



This is to explain the concept of cross-validation. Of course, leaving out a single image is not too complex in a normal work-flow and thus not very exciting. However, the Rotation and Randomization schemes performed using the cross-validation over images become a great help. Where the Cross-validation tool really shines is [the cross-validation considering replicas](#).

**TIP:** After each manual retraining, you may copy the test set performance out from the *Statistics* tab

### Closing cross-validation session

The cross-validation session needs to be ended by pressing *End session* button <sup>1</sup> above. Alternatively, cancelling image selection also disables the session. When selecting multiple images again, we must explicitly start the cross-validation session.

### Cross-validation over replicas

#### What is a replica?

**Replica is a repeated measurement of the same physical sample.** In order to estimate true generalization performance of our models, we should keep replicas of a specific physical object either in training or in test set, but never split between both. The reason is, that having very similar examples in both training set and the test set makes performance of our models positively biased (over-optimistic). Our models have seen very similar data in training and thus correct results on such data in the test set do not necessarily translate into good generalization capabilities. By generalization we mean robust performance on entirely unseen examples.

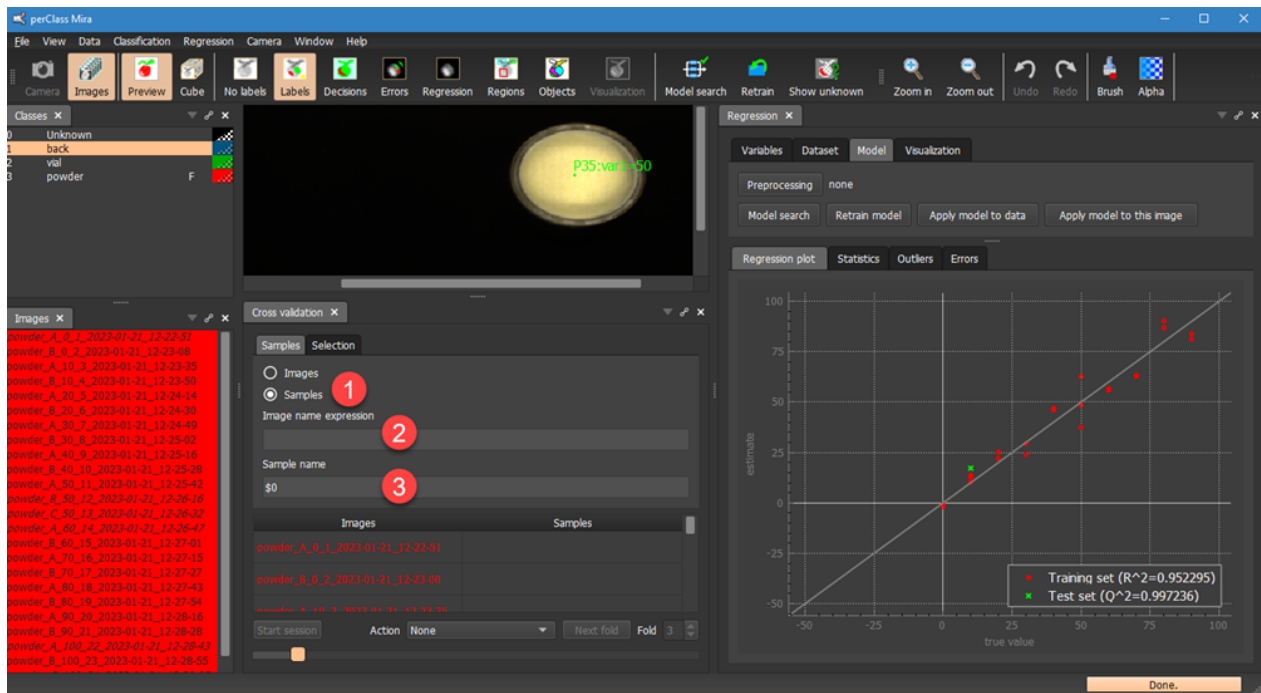
#### Cross-validation over replicas in perClass Mira

If the replica status is preserved in the scan filenames, we can easily instruct the *Cross-validation* tool to perform data splitting over replicas not over images. For example, in our powder data set, the A/B/C... letter indicates a replica of the same physical vial (container).

We keep the leave-one-out method selected in the *Selection* tab, return to *Samples* tab and change the

selection from *Image* to *Samples* <sup>1</sup>. This means, that we may define what constitutes are sample for cross-validation. The default is invalid which leads to all selected images flagged as red. We can now

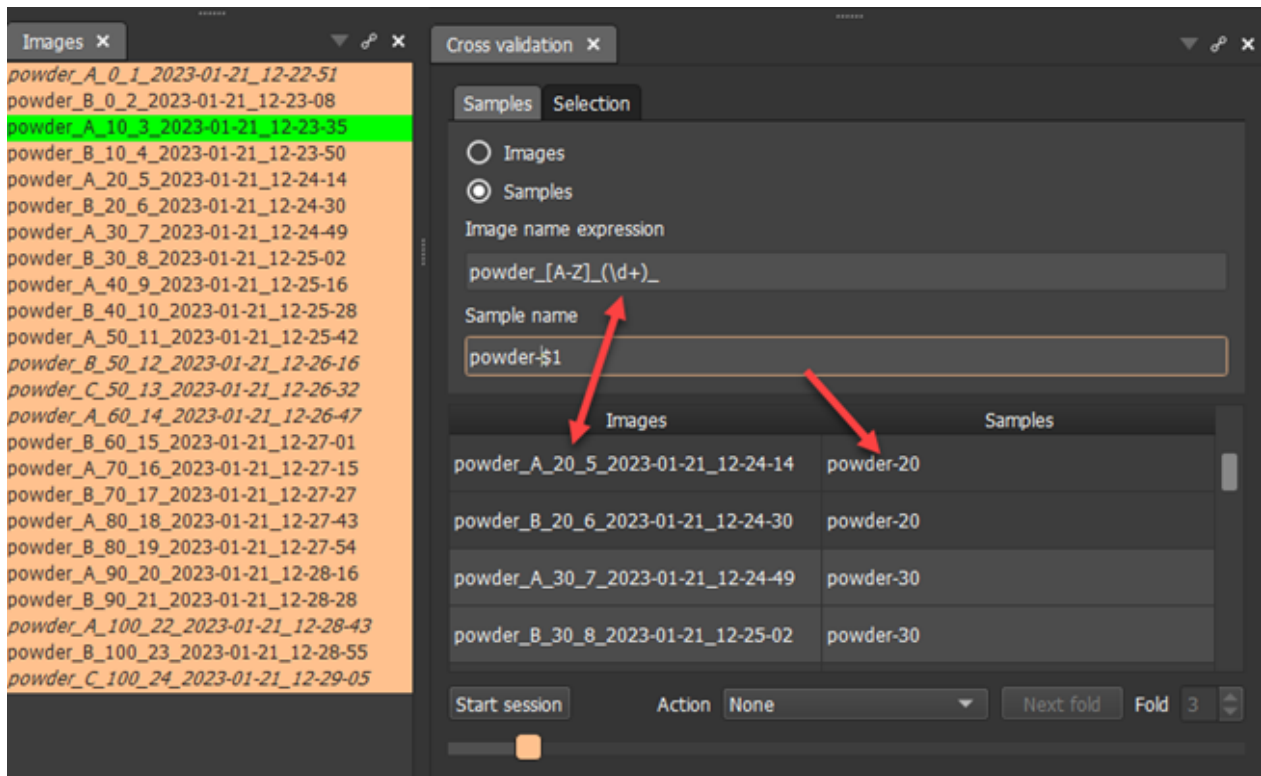
define a regular expression in <sup>2</sup> that parses image names and the sample definition in <sup>3</sup> that is used for the cross-validation.



Example solution in our case is to detect the mixing proportion from the file name and construct a new file name that only lists the mixing proportion, nothing else. The reason is that we want to make sure that one vial (i.e. one mixing proportion) ends up either in training or in testing set but not split in both.

Technically, we put in <sup>2</sup> the regular expression that matches each file name allowing for the replica definition using a single capital letter from A-Z range. After the underscore, we capture one or more diits until the unther underscore. The capture (part of a string that will be extracted and available to us for reference) is enclosed in round brackets. The \d refers to a digit, the + sign after means that the digit repeats one or more times. This is a standard regular expression syntax that is very handy when dealing with structured patterns in strings.

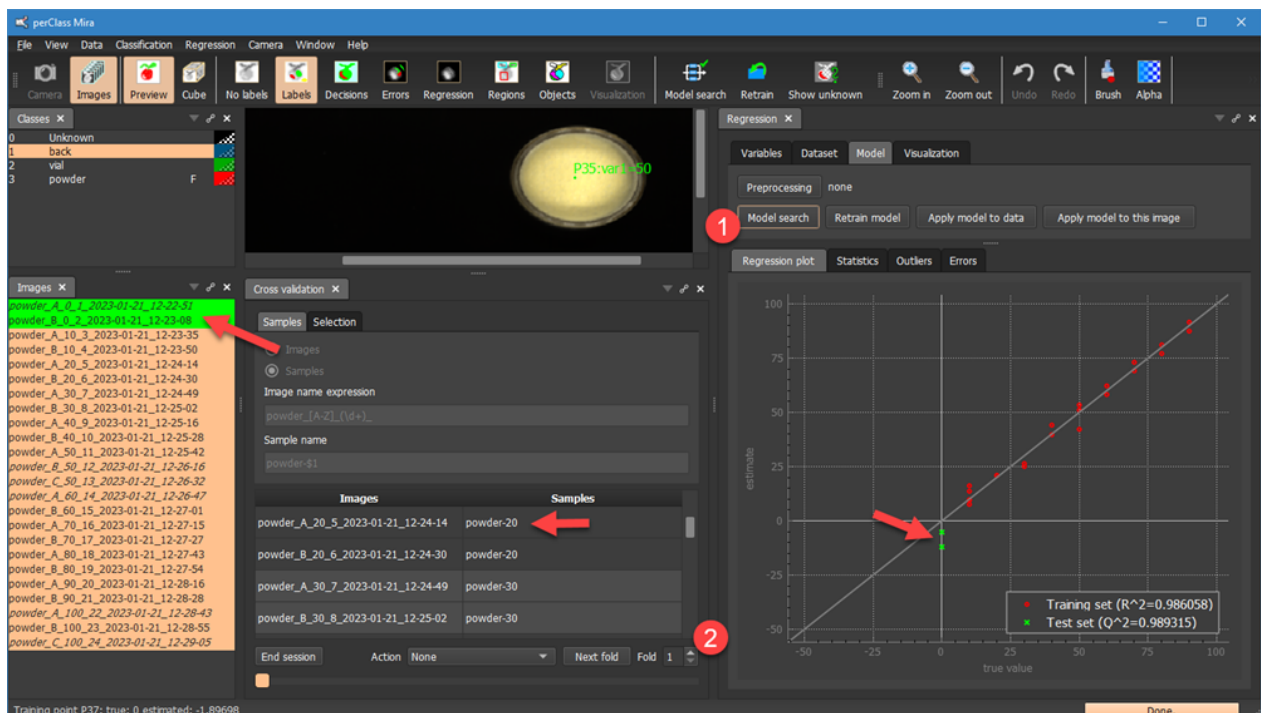
**TIP:** For a reference information on regular expressions, see [https://en.wikipedia.org/wiki/Regular\\_expression](https://en.wikipedia.org/wiki/Regular_expression)



We also fill the output pattern in the *Sample name* field <sup>3</sup> above. The important point is that we may refer here to any captures using \$1, \$2 etc. syntax denoting the 1st, 2nd or later capture (text matched within the round parentheses of the regular expression).

The table shows how original image filename translate into our new definition.

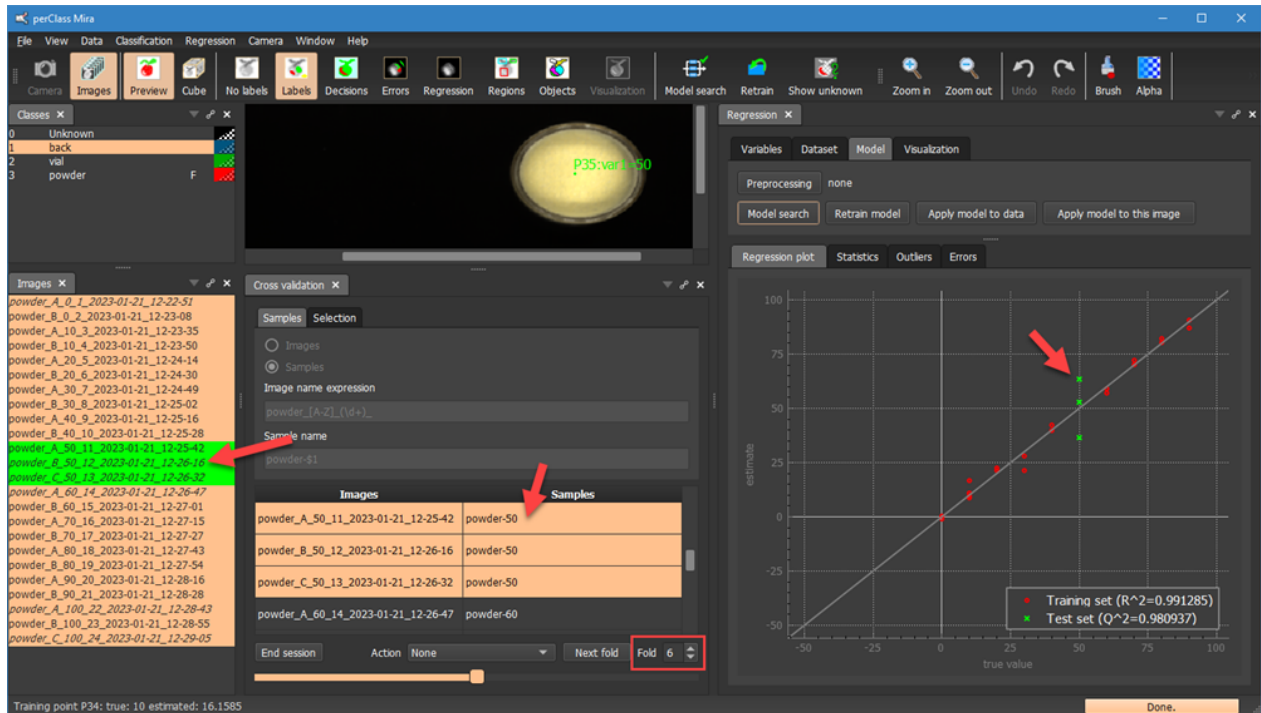
By clicking *Start session* we can initiate new cross-validation session where we will perform leave-one-vial-out. We also pressed *Model search* in *Regression* panel in order to directly see test examples in the *Regression plot*. Note, that the first fold now covers **all replicas of the vial with mixing proportion 0**. In our case these are two images.





By selecting a different fold using the spinbox <sup>2</sup> and re-running the *Model search*, we will exclude all replicas of another vial from training:

and re-running the *Model search*, we will exclude all replicas of another vial from training:



Note, that in the fold 6, we have three replicas of the vial with mixing proportion 60. All three are now in a test set.

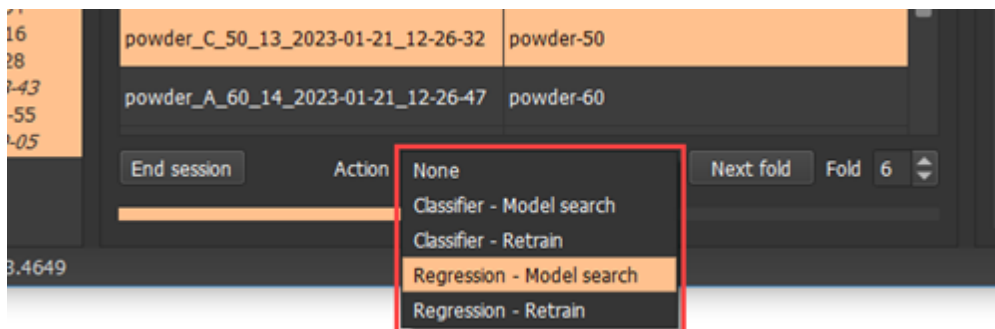
For each fold model, we may copy the regression performance from *Statistics* tab to clipboard, paste into Excel sheet. In this way, we gradually build a table of per-fold results where we will be able to assess statistical variability of each measure.

In this section, we have seen, how to fairly assess performance of our regression model on unseen vials in powder project.

### Default action

perClass Mira *Cross-validation* tool allows us to define default action executed when moving to a new fold. By default, nothing happens and user can perform any desired analysis manually. That is what we did in the earlier examples in this section.

The *Action* combo box in the *Cross-validation* panel allows us to change this default behaviour. We may retrain classifier or regressor or rerun their model search.



Now, after a new fold is selected, the regression model search is automatically re-run. This simplifies

further the common work-flow:

- Setup cross-validation scheme and definition of samples, if needed
- Set the default action
- Start cross-validation session
  - go to a fold
  - model is rebuilt automatically
  - copy results of interest
- Repeat until all folds are performed

## Reference

---

This chapter provides reference information such as

- [Software release notes](#)
- [Application server](#)
- [perClass Stage](#)
- [perClass Camera API](#)
- [perClass Mira Runtime API](#)

## Integration

This section describes how to integrate solutions, built with perClass Mira, into custom applications running camera acquisition and data processing.

The scheme below describes the modules involved. While perClass software components are rendered in blue, the spectral sensor in red and customer-specific parts in yellow.

The left side depicts the Design stage where perClass Mira Dev user-interface (1) to connect to a spectral camera (2). From the user interface, we can record scans (A) used for training and testing the models. Once the solution is built and properly validated, it may be exported into "Mira PipeLine" MPL file (B).

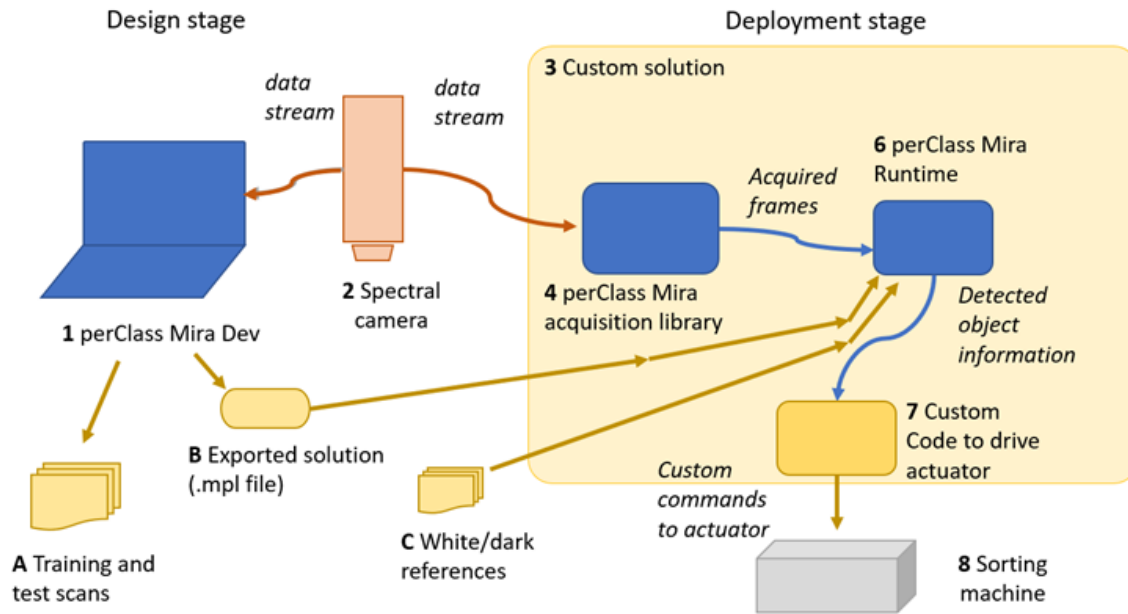
Then, we may proceed to the Deployment stage on the right side. There, we wish to run an industrial sorting machine (8) with the camera (2) in a tight control loop. This loop is executed on a PC in a custom solution (3). This is an application that reads data from the sensor (2) using [perClass Mira Camera API](#) (4) and processes this data with [perClass Mira Runtime API](#) (6).

The runtime (6) loads the exported MPL solution (B). This solution typically contains a classifier able to identify objects. Objects are detected by the Runtime (6) and their details such as size, centroid position and class are passed to a custom code (7). This module translates the coordinates from pixels (across the belt) and frames (along the belt) to machine-specific coordinates and drives the actuators.

Commented example of acquisition from Camera API [is available here](#).



## perClass deployment setup on a PC



4-mar-2022

### Example of acquisition from Camera API

This example shows how to acquire data from a sensor using perClass Mira Camera API. The sensor used is Headwall MV.C VNIR. The example is generic and should work unchanged with other line-scan cameras when linked to the respective perClass acquisition plugin.

The example shows how to:

- inquire on the version of acquisition library (line 13)
- initialize the acquisition plugin (line 20)
- scan for available devices and return their names (line 25-31)
- open a device (line 33)
- test if a device is a line-scan or snapshot (line 36)
- setup wavelength resampling to specific output wavelengths irrespective of a device (line 43-49)
- initialize acquisition (line 51)
- query frame geometry, data type and layout (60-72)
- set exposure and frame rate (lines 74-76)
- acquire 100 frames, store data in memory (line 94-100)

In order to compile the example with command line Microsoft Visual C/C++ compiler use:

```
> cl ex05.c -I "C:\Program Files\perClass Mira\lib" "C:\Program Files\perClass Mira\lib\miraacq_ximea_1.7.1.lib"
```

Note, that we point to the lib sub-dir of perClass Mira installation for includes and directly link with the acquisition plugin for your camera (Ximea plugin for MV.C VNIR).

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <stdint.h>
4. #include "miraacq.h"
5.
6. int main(int argc, const char* argv[])
7. {
8.     int res=MIRA_OK;
9.     uint16_t* pBuf=NULL;
```



```

10. FILE* fid=NULL;
11.
12. int api,rev,step;
13. const char* str=miraacq_GetAPIVersion(&api,&rev,&step);
14. printf("Example of acquiring data using perClass Mira Acquisition Plug
15.
16. const char* str2=miraacq_GetVersion();
17. printf("Version: '%s'\n",str2);
18.
19. makernel* pma=miraacq_Init(".");
20. printf("Init: %s", miraacq_GetErrorMsg(pma));
21. if( pma==NULL ) {
22.     goto Error;
23. }
24.
25. MIRAACQ_CHECK( miraacq_ScanDevices(pma) );
26.
27. const int devCount=miraacq_GetDeviceCount(pma);
28. printf("\n%d devices:\n",devCount);
29. for(int i=0;i<devCount;i++) {
30.     printf("%d : %s\n",i, miraacq_GetDeviceName(pma,i));
31. }
32. int deviceInd=0;
33. MIRAACQ_CHECK( miraacq_OpenDevice(pma,deviceInd) );
34. printf("Device opened: %d '%s'\n",deviceInd,miraacq_GetDeviceName(pma
35.
36. int isSnapshot=miraacq_DeviceIsSnapshot(pma);
37. if( isSnapshot ) {
38.     printf("Line-scan device required by this example\n");
39.     MIRAACQ_CHECK( miraacq_CloseDevice(pma,deviceInd) );
40.     goto Error;
41. }
42.
43. printf("Setting resampling from 400 to 1000, step 2\n");
44. int bands=(1000-400)/2;
45. MIRAACQ_CHECK( miraacq_SetResamplingWavelengthCount(pma,bands) );
46. for(int i=0;i<bands;i++) {
47.     MIRAACQ_CHECK( miraacq_SetResamplingWavelength(pma,i,400+(2*i)) );
48. }
49. MIRAACQ_CHECK( miraacq_SetResampling(pma,1) );
50.
51. printf("Initializing the acquisition...");
52. res=miraacq_InitializeAcquisition(pma);
53. printf("done res=%d\n",res);
54. fflush(0);
55. if( res!=MIRA_OK ) {
56.     MIRAACQ_CHECK( miraacq_CloseDevice(pma,deviceInd) );
57.     goto Error;
58. }
59.
60. printf("Geometry: width=%d bands=%d lines=%d dataType=%d dataLayout=%d\n",
61. miraacq_GetFrameWidth(pma),
62. miraacq_GetFrameBands(pma),
63. miraacq_GetFrameHeight(pma),

```

```

64. miraacq_GetFrameDataType(pma),
65. miraacq_GetFrameDataLayout(pma),
66. miraacq_GetFrameSize(pma) );
67.
68. if( miraacq_GetFrameDataType(pma)!=ACQ_DATATYPE_UINT16 ) {
69.     printf("UINT16 data type expected by this example\n");
70.     MIRAACQ_CHECK(miraacq_CloseDevice(pma,deviceInd));
71.     goto Error;
72. }
73.
74. MIRAACQ_CHECK( miraacq_SetExposure(pma,4.0) );
75.
76. MIRAACQ_CHECK( miraacq_SetFrameRate(pma,100.0) );
77.
78. const int frames=100;
79. int frameSize=miraacq_GetFrameSize(pma);
80. pBuf=malloc( frames*frameSize );
81. if( pBuf==NULL ) {
82.     MIRAACQ_CHECK(miraacq_CloseDevice(pma,deviceInd));
83.     goto Error;
84. }
85.
86. MIRAACQ_CHECK( miraacq_StartAcquisition(pma) );
87.
88. uint16_t* ptr=pBuf;
89. size_t frameID=0;
90.
91. // offset to the next frame in uint16 units (a frame has width x bands
92. int nextFrameOffset= miraacq_GetFrameWidth(pma)*miraacq_GetFrameBands
93.
94. printf("Acquiring %d frames:\n",frames);
95. for(int i=0;i<frames;i++) {
96.
97.     MIRAACQ_CHECK( miraacq_GetFrame(pma,ptr,&frameID,1000) );
98.
99.     ptr+=nextFrameOffset;
100. }
101.
102. printf("Stopping acquisition\n");
103. MIRAACQ_CHECK( miraacq_StopAcquisition(pma) );
104.
105. printf("Writing data to file:\n");
106. FILE* pFile=fopen("out.bin","wb");
107. if( pFile==NULL ) {
108.     printf("Cannot open file for writing\n");
109.     MIRAACQ_CHECK(miraacq_CloseDevice(pma,deviceInd));
110.     goto Error;
111. }
112.
113. size_t countWritten=fwrite(pBuf,(size_t)frameSize,frames,pFile);
114. printf("%zu bytes written to file\n",countWritten*frameSize);
115.
116. fclose(pFile);
117.

```

```

118.Error:
119.    if( res!=MIRA_OK ) {
120.        printf("Error %d: %s",miraacq_GetErrorCode(pma),miraacq_GetErrorM
121.    }
122.
123.    miraacq_Release(pma);
124.
125.    if( pBuf!=NULL) free(pBuf);
126.
127.    return 0;
128.}
129.

```

## Release notes

### 4.2.9 4-oct-2023

- Acquisition improvements
  - Headwall MV.C VNIR camera supporting frame buffering. If computer cannot keep up with live acquisition, the buffered frames will be now saved at the end of scanning session. Frame buffering can be disabled by ini option
  - Headwall MV.C NIR camera new shutter mode supported
  - New VimbaX plugin to support Resonon NIR systems (replacing the deprecated Resonon SDK)
  - Type of data stream included in recorded header files ("Raw calibrated" or "Resampled")
- perClass Mira Stage
  - Speeding up stage response
  - Fixing the intermittent problem where scans could be extended in length
  - Adding stage cycle number
- perClass Mira improvements
  - Regression import dialog interprets Excel address references case-insensitive

### 4.2.8 23-aug-2023

- perClass Mira Stage
  - Added support for perClass Mira Stage v2.0
  - Added TCP/IP commands to [control the stage](#) via Application Server interface
- Region improvements
  - Added command to add regions from a template
    - Works also in a batch mode for all selected image
  - Removing regions from multiple images does not remove template regions, user is asked to explicitly confirm
- Regression improvements
  - Added RER performance statistics in regression
  - Fixing the display of output with multiple objects
- Acquisition plugins
  - fix in filereader plugin - preloading multiple cubes properly handles memory size limits
  - Headwall HyperspecIII plugin adds support for wavelength information from the sensor
  - Headwall MV.X allows using arbitrary IP address for the websocket communication (identical to the eBUS data connection)
  - Pleora eBUS plugin adding eBUS 6.3 version which enables full support Windows 11 (eBUS 5.1 and 6.1 are also supported)
  - Resonon acquisition adds a setting for a number of acquired reference frames
  - Unispectral acquisition reports situations where the camera does not return a frame
- Benchmarking improvements
  - Added automatic optimization of the best band subset (ROI) and classification model for a

- given project
- Benchmark visualization of the benchmarked ROIs for speed, for error and for both criteria
- Best feature set found can be set to the project spectral plot

### 4.2.7 26-jun-2023

- Fixed issue with dropped frames on Headwall MV.C VNIR
- Benchmark frame count limit increased to 100k frames

### 4.2.6 14-jun-2023

- New command to copy class mean (min/max) spectra to clipboard as text in Spectral plot
  - Copy wavelengths and band selection to clipboard information as text
- The export into Matlab command is batched applying the action to selected images
- The Add regions from objects command is batched applying the action to selected images
- New Set source command in Camera menu allows to switch the project to a different acquisition target (from the favorites)
- New dropped frame counter in the Camera panel
- Panels in the Windows menu are now in alphabetic order
- Fixed crash when changing acquisition source
- Cubert acquisition improvements
  - Distance can be changed via mira.ini file
  - Fixed auto-exposure

### 4.2.4 20-apr-2023

- added support for Avaldata camera using TransFlyer SDK
- added export of regions as cubes
- Help menu now opens on-line documentation in a browser
- Application server: added object detection channel for filereader plugin
- fix for crash in visualization when using image rotation
- fix in sync panel to download only selected directories
- fix in buffer queueing in Pleora eBUS acquisition
- Headwall MV.X license and runtime installation and switching from sync panel
- Headwall MV.C cameras
  - adding reference frame count in header files, default 100
  - fixed ROI offset and wavelength flip

## 4.2 13-mar-2023

- new [Camera and Images modes](#) allow easy transition between live acquisition and working with saved scans
- support for [perClass Stage](#)
  - includes user-defined commands for stage hardware buttons
  - support for a quick setup of a camera
    - auto-exposure leveraging available dynamic range for current illumination
    - finding optimal focus using easy user feedback
    - adjustment of scanning speed or frame rate to reach square pixels for line-scan cameras
- new [Cross-validation tool](#)
  - Easily perform leave-one-out, rotation and randomization cross-validation for classification or regression
  - Supports [cross-validation over images](#) or [over replicas](#) (multiple scans of one physical sample that need to be all either in training or in the test set)
- new tool to add [manual object separation](#) to existing object segmentation

- This allows one to pack more objects in one scan even if touching. Manual object separation does not extend to deployment.
- Object separation enables also fine control on areas used for regression analysis
- new sensor support
  - full support for [Headwall MV.C VNIR](#) and [NIR cameras](#)
  - partial support for legacy Headwall HyperspecIII cameras (currently VNIR and NIR supported, not SWIR)
  - support for snapshot filereader
  - experimental support for Agrowing sensors
- new [perClass Camera API](#)
  - provides a unified interface for embedding data acquisition into custom applications

## 4.1 22-sep-2022

- new docking system allowing better panel positioning
  - user may name and save "perspective" of all open panels over multiple screens
- data acquisition
  - new frame widget showing live raw frames with spatial and spectral profiles
    - saturation detection for selected cameras
    - frame widget is automatically active also on loaded scans (off-line)
  - new "belt" visualization of the live data stream
    - visualizaing live object classification results also for line-scans
    - possible to switch between waterfall and belt views
- new filereader plugin for line scans
  - allows setting of frame rate to measure algorithm speed
- user-defined visualization color maps
  - multiple color points and colormap reversal
  - colormaps can be stored in mira.ini file for re-use beyween projects and copy/pasted as text
- improved alpha layer handling
  - two sliders provided, one for all classes and one for the currently selected class
  - alpha layer can be adjusted during the live acquisition to highlight only the decisions of interest
- new [application server](#) functionality
  - perClass Mira acquisition can be controlled via text commands sent over TCP/IP connection
  - this allows quick construction of live demonstrators including custom actuator without low-level programming
  - separate object detection channel allowing one to react on detections
- new sensor support
  - HAIP BlackIndustry
  - Headwall MV.X using Pleora eBUS
  - Silios CMS using Silios SDK
- fixes
  - Excel export of full spectral uses .xlsx file format by default allowing up to 16k columns
  - regression auto-scale via context menu
  - add regions from current objects works correctly in cases when segmentation was not applied to the image
  - adjustments of the confusion matrix to avoid unreadable text due to close foreground/background colors
  - fixed auto-detection of ENVI cubes without extension

## 4.0 6-apr-2022

- adding comprehensive data acquisition and recording functionality
  - Supported camera types
    - Cubert - Ultris series
    - Imec - all Mosaic systems (including PhotonFocus and Ximea cameras)

- Inno-spec - RedEye 1.7 NIR and Speccer moving stage
- Resonon - both VNIR systems and NIR Pika systems
- Specim - FX series (via SpecSensor SDK)
- Unispectral - Monarch
- redesigned new project dialog - the user can select to either to
  - load existing scans recorded in camera vendor-specific software
  - or to do live data acquisition from supported camera
- live acquisitions support raw (uncorrected) data from spectral cameras and user-defined reflectance correction work-flows
  - point correction (based on user-localized white reference in the scene)
  - non-uniformity correction to account for inhomogeneous illumination
  - user-defined white level to support gray references
  - setting references from existing scans
- data in the live acquisition is saved in the new perClass Mira data format (ENVI-based, .pcf extension)
  - In this way perClass Mira supports multiple correction work-flows for any supported camera type
  - scan-specific and directory-specific correction references
- live data processing includes object segmentation and classification for snapshots
- new Cubert project type supporting .cu3 files for all cameras
- specific features for Unispectral
  - Supporting band selection in the camera to speed up acquisition
  - Fix supporting 'default bands' field with only a single entry
- for VNIR systems, R,G and B lines in the preview mode are set to meaningful defaults. User can change preferred defaults in mira.ini
- Export of spectral cube to Matlab now includes also the wavelength vector
- Regression improvements
  - Significantly faster operation, avoiding processing of images on project load

### 3.1.2 2-dec-2021

- copy current image view to clipboard as image using Ctrl+C
- adding display autoStretch option with a slider control in the spectral plot context menu
- regression improvements
  - adding dark/light background option to regression plots
  - adding copy to clipboard to regression plots
  - adding copy as text (direct copy to Excel) for regression performance values
- object confusion matrix considers only true regions from classes flagged as foreground
- enabling regression output in acquisition mode
- fix of regression issue that could lead to non-reproducible model when bands in the end of the range were selected
- enhancing support for spectral cubes larger than 4GB
- fixing crop on Silios images
- fixing the issue where object segmentation sometimes flipped to object IDs even if object labels were set
- fixing the issue with auto scaling of regression plot in situations with a lot of outliers

### 3.1.1 25-oct-2021

- fixing the bug in display auto stretch where the stretch was on by not enabled
- fixing the problem when training classifiers on large cubes (>4GB)
- fixing the crash in object confusion matrix

### 3.1 3-sep-2021

- new project types
  - Inno-spec project including reflectance correction on scan load (correction can be specified per-image and per-directory)
  - Resonon project type supporting reflectance correction on scan load
- new installers
  - adding support for CUDA11.2
  - separate full installer including NVIDIA CUDA support
  - separate smaller installer for CPU + OpenCL backends convenient also for virtual machines
- significant speedup of classification at runtime
  - holds for both CPU and GPU backends including also older projects
- new acquisition functionality
  - acquisition plugins allow use of different vendor SDKs
  - adding support for Resonon Pika cameras
  - SpecSensor plugins for 2019 and 2020 SDKs
  - Pleora eBUS support for eBUS 5.1 and 6.1 adds support for GenICam-compliant sensors such as Inno-Spec RedEye2
- improvements in regression
  - outlier score plot and error plots help to clean training/test set of outliers
  - performance measures panel with user-defined acceptance criteria
- GUI improvements
  - object-level confusion matrix with interactive visualization of ground-truth and detections allows full introspection of object-level decisions
  - added support for object shape features (Feret diameter, Hu moments, circularity)
  - added support for multiple directory selection for projects where each scan is a directory
  - images with labels show image names in italics
  - new auto-stretch of image brightness with slider-based adjustment in spectral plot menu

### 3.0 22-mar-2021

- improvements in regression
  - support for multiple regression variables
  - significant speed-up when updating regression data sets
  - separate commands for model search, retraining model and applying model both to data and on a new scan
  - import regression meta-data from Excel also at region level (via named regions, see below)
  - easy inspection of outliers: jump to a scan containing specific object/annotation point
  - runtime API for per-object and per-pixel regression output for each variable
    - support for background pixel masking
- introducing user-defined regions
  - regions have unique names within each image and are assigned to a specific class
  - regions can define object ground-truth labels
  - by matching regions to object found it is possible to estimate of confusion matrix at object level and assess sorting performance
  - Excel export and import of region definitions
  - user-defined text annotation such as expert remarks can be added
- introducing feature extraction
  - extract and export user-defined features from objects or user-defined regions
    - mean spectra
    - spectral index mean or histogram per object
    - fraction of decisions per object
    - regression output per object
    - object count



- information can be extracted from computed objects or from user-defined regions
    - for regions, presence/absence of data is reported (e.g. no plant in a germination well)
  - export to Excel and XML formats
- introducing batch feature extraction accessible from scripts without GUI via `perClass_Mira_Batch.exe`
  - export to XML format
  - define a template image specifying regions for extraction (e.g. grid of germination wells)
  - validating scans via a user-defined model rejecting data unseen in training
- improvements to image flagging
  - set selected images for testing or training
  - set a percentage of selected images as test (to perform user-defined cross-validation studies)
- batch crop applied to selected images
- spectral index definitions are saved in the `.mira` project file
- improves when processing large number of scans
  - ability to cancel long running operations (like result exports or regression meta-data imports)
- commands to switch between band subset used for a classifier and for a regressor
- possible to define band subset manually by band indices (e.g. 20:40 will enable bands 20 to 40)
  - adding and removing bands to/from existing band subset (useful to disable certain ranges)
  - possible to set or toggle each Nth band
- new project type for Silios CMS cameras

## 2.4 28-sep-2020

- added reflectance correction for Headwall project type (correction by `whiteReference` and `darkReference` ENVI cubes in the same directory)
  - allows loading of externally corrected cubes in the same project
  - enables multiple scans per directory sharing the same correction
  - default cube extension is `.bin`, arbitrary extensions are supported
  - to apply correction at runtime, pass directory containing `whiteReference` and `darkReference` scans to `mira_LoadCorrection` (example:  
`mira_LoadCorrection(pmr, "path_to_dir_with_correction_files", NULL) )`
- added general ENVI project type supporting arbitrary cube file extension
- added Corning project type
  - added perClass Mira Runtime support for native BIP data stream corrected with dark reference inside the camera
- improved selection of multiple images (click and drag supported, no image reload in multiple selection)
- improved drag&drop of directories (adding all files within each dropped dir)
- added support for NVIDIA CUDA11 (Ampere)
- when using floating licenses, specific licensing product can be requested based on `floatingLicenseProduct` setting in `mira.ini` (`mira` for perClass Mira Dev and `mira.gui` for perClass Mira)
- when importing regression annotation from Excel, existing points are removed to avoid duplicates
- fixed a problem when adding regression annotation to all objects in each scan
- fixed problem when label painting with large brushes
- fixed memory leak in loading large number of specim FX scans
- fix for dropped frames at the start of live acquisition session
- at runtime, all projects (including line-scans) must explicitly enable object segmentation with `mira_SetSegmentation(pmr, 1)`

## 2.3 26-jun-2020

- support for foreign object detection with truly unknown objects
  - label materials you know. Enable *Show unknown* to highlight all materials unseen in training.
  - user-adjustable sensitivity on per-class basis provides extra control (slider via the right-click in

- the class-list)
    - objects unseen in training can be segmented out (flag *Unknown* decision as foreground)
    - the new foreign object optimizer is on by default, can be disabled in Classification menu.
- color wells display transparency (change alpha for a specific class in the color dialog or by via alpha toolbar button by holding Ctrl)
- crop improvements
  - crop rectangle line thickness auto-adjusted for very large cubes
  - adjust crop rectangle by dragging lines
- segmentation improvements
  - support for up to 20 foreground classes including access to their content information
  - per-object results can be batch-exported to Excel including per-class content in each object
  - fix for a crash due to changing object size in live acquisition mode
- confusion matrix improvements
  - added light mode (to allow copy/paste directly to documents)
  - added option to copy as text for direct copy/paste to Excel
- fixed live acquisition issue when Specim calibration file (.scp) was not found
- fix for min/max visualization setting in presence of NaNs and infinite values
- support for case insensitive fields ENVI in header files (for Python integration)
- runtime improvements
  - support for region of interest (ROI) for snapshots. Applying classifier only to specific ROI.
  - support for object segmentation for snapshot use-cases (Imec project type, float data type, BIP layout)

## 2.2 29-apr-2020

- new [Visualization mode](#) showing computed indices using different common equations
  - define using individual wavelengths or wavelength ranges
  - auto-scaling and manual scaling
  - indication of below, above and invalid values
  - define wavelength ranges interactively in spectral plot
  - render using different colormaps
- improved [regression](#)
  - visualize [per-pixel regression output](#) (e.g. distribution of moisture)
  - [import point annotations from Excel](#) (matching scan names exactly or with regular expressions)
  - move and edit point annotations
  - use only specific subset of spectral bands
  - show cross-validated regression error (RMSECV) which has the same units as the regressed value
  - when hovering over the results in the regression plot, display specific annotation points with their true and estimated values
  - visual indication that some point annotations are not linked to objects (e.g. point not on foreground class)
  - [export regression results in Excel](#) together with per-object size, bounding boxes, true and estimated regression outputs
- perClass Mira Runtime improvements
  - added model export for perClass Mira Runtime (new "Mira Pipeline" .mpl format using base64 encoding)
  - added API to query expected data type, data layout and geometry of data from spectral camera
  - added support for all object segmentation configurations created in the GUI including per-object content retrieval and object classification by rules
  - added snapshot processing mode (mira\_ProcessCube). Currently only pixel decisions are provided, not yet the object segmentation or content.
- added support for OceanInsight Spectrocam and Pixelcam data formats
- added support for ENVI cubes with uint32 data type and little-endian float

- added classifier preprocessing (smoothing, 1st and 2nd derivative)
- export and import labels as PNG images
- export per-image results to Excel allowing quick summary of fraction of decisions within foreground (e.g. disease within plant leaves)
- update of live acquisition using Specim SpecSensor SDK
  - Applying regression both per-object and per-pixel in live acquisition
  - Calibration pack information stored in settings, reused for further sessions
- fixes in object pannel: When retraining the classifier, object classification rules are preserved
- adding default class color map
- repeatable object label colors (can be change using random seed dialog)
- added per-class transparency (alpha setting in the color dialog and using the toolbar transparency slider - hold Ctrl to change only the current class alpha)

## 2.1 18-feb-2020

- Specim FX project type allows scan directories with different name than raw cube in capture sub-folder
- Unicode support in image file names for ENVI-based formats
- providing informative error messages when image cannot be loaded
- adding Cubert Tiff project type with native support for Cubert Ultris camera
- adding Headwall project type
- license file can be drag & dropped from Explorer to the license dialog
- RGB bands are set based on ENVI header file
- mira.log file is now written to AppData/Roaming, not to the installation directory (now by default in Program Files (x86))
- fix of calibration pack loading in SpecSensor
- labels can be exported into .png files
- ENVI import supports int16 data type
- when the number of sampe is too low, the output window shows a red message that can provide details on click
- when alpha is too low (high label transparency), the toolbar alpha button blinks to remind the user that labes may be badly visible

## 2.0 18-oct-2019

- new Cubert ENVI project enabling data from Cubert Ultris and upsampled UH185 images
- perClass Mira Runtime binaries adding dongle support

## 2.0 10-oct-2019

- adding support for double-precision ENVI data cubes
- supporting model deployment for execution on live data from Cubert Ultris light-field hyperspectral camera
- enabling Cubert plugin export for ENVI-based projects.
- fixes in live acquisition using Specim FX cameras when device loading fails or opening FileReader gets cancelled
- fixing a crash due to very large training set
- fixing a bug in error visualization mode where switching to images without labels did not show proper image

## 2.0 20-sep-2019

- Fix: Installation directories with non-ASCII characters are now supported
- Live acquisition executables for Specim cameras included (perClass\_Mira\_live.exe and perClass\_Mira\_gpu\_live.exe)

- Senop project: Images are automatically processed with per-band gain

## 2.0 6-sep-2019

- Estimate [object quality using regression](#) (examples: sugar content estimation per tomato)
  - annotate quality per object
  - automatic model selection reporting performance ( $R^2$  and  $Q^2$  statistics)
  - user-defined pre-processing (smoothing and derivatives)
  - apply regression to new images (show a bounding box + regression output per object)
  - allow localized information extraction by a radius around annotation points
- Images can be [flagged for testing only](#) (not used for building the model)
  - **Test confusion matrix** provides a detailed view of the performance on test images
- [Error visualization mode](#) brings insight in model performance.
  - visualize where the current model fails
  - this helps to identify incorrect labels or (together with test image flagging) whether the data is well represented in the training set
  - **Image confusion matrix** shows only labeled examples on the current image
  - [interactive error visualization](#) by moving mouse over the image confusion matrix
- [Object segmentation mode](#) with multiple options
  - one object / one class mode for object detection (e.g. detect plastic pieces in a food product stream for automatic removal)
  - one object / multiple classes for object classification (e.g. detect potato pieces, classify entire piece as defective if it contains more than 5% of greening or rot inside)
  - visualizing object labels or object decisions
  - object decisions by majority vote or rules (size of or fraction of a specific class)
- Usability improvements
  - **assign label stroke to the current class**. This allows one to exclude a specific label stroke from training and see the impact on model performance (define an additional class and exclude it, assign strokes to it and retrain)
  - the data validation mechanism excluding invalid spectra is now off by default. It can be enabled using context menu in the spectral plot.
  - all modes (labels, decisions, errors, objects) accessible by direct keystrokes
  - confusion matrix size can be decreased/increased (useful for large number of classes)
  - auto-check for software updates + direct link to download latest version from the GUI (Help / Check for updates)
- *experimental* [Live data acquisition](#) from **Specim FX cameras** using Specsens SDK (needs to be installed separately)
  - apply a classifier and object segmentation to a live data stream
  - live visualization of processing speed and drop frame indication to assess production performance
  - user-control of exposure and camera frame-rate
  - supports practical situations where production light conditions are different from the training situation
    - the white and dark references used for live data processing can be specified without model retraining
  - automatic handling of spectral and spatial binning based on specific scan meta-data
  - support for **outdoor operation**: Define white reference by specifying an image region where a reference tile was placed
  - **recording data** from a live acquisition in the standard LUMO format

## 1.4 22-may-2019

- **perClass Mira Runtime** is now included in the distribution
  - high throughput (1.5ms/frame on NVIDIA GPU in an example foreign object detection project, Specim FX17, 640 spatial pixels, 224 bands, 6 materials)

- the runtime directly reports object positions, sizes and classes
  - support for **NVIDIA Jetson** platform (both ARM CPU and NVIDIA GPU backend)
  - support for line-scan use-case on Specim projects (specific white/dark correction format)
- **Linux build** for both perClass Mira GUI and perClass Mira Runtime
  - accelerated CPU and GPU support on Linux
- new high-throughput segmentation engine
  - automatically discarding objects smaller than user-defined minimal size
  - supporting multiple foreground classes
  - high-speed line-scan segmentation with constant per-frame speed
- export visualization as PNG images (band or RGB, with labels, pixel decisions or segmented objects)\
- for Cubert projects, proper wavelength ranges are shown

## 1.3 8-feb-2019

- zoom using mouse wheel now follows cursor
- image rotation using toolbar buttons (and > < keyboard shortcuts)
- adding images using drag and drop from Windows explorer
- support for ENVI files with high-endian byte order uint16 (byte order=1)
- saved projects now preserve settings of the current band, R,G,B lines and allow direct execution of the trained model when project is loaded
- exported decision images (PNGs) contain meta-data such as class count and class names accessible by standard tools such as [tweakpng](#) or Matlab `imfinfo` command
- multiple directory selection for Specim FX and Tiff stack project types can be enabled in mira.ini file (using `useNativeDirSelection=false`). It is not enabled by default because it uses a non-native file dialog.
- new project type for Senop cameras (formerly Rikola)

## 1.2 5-dec-2018

- Added [band-selection widget](#). It is now possible to manually select the wavelengths used for building models
  - Band brushing allows quick selection or clearing of wavelength ranges
  - Exported models start from the full set of wavelengths but use only the selected subset for the model. This allows quick deployment of different models to custom applications assuming full spectrum (single binding with perClass Runtime is needed)
- Added export of labeled data to perClass Toolbox `sddata` format
- Added export of entire data cube in Matlab format as 3D matrix
- Models results are now repeatable with a new random seed dialog controlling the internal data partitioning process.
- Separate CPU-only and CPU+GPU builds are available. The CPU-only build is always available by default to avoid issue related to GPU drivers or CUDA versions installed. The CPU+GPU executable is called `perClass_Mira_gpu.exe`
- Band index and the wavelength number are now updated on the status bar when dragging the band line in spectral plot
- Added support for logging of status messages when starting up the application. This is useful to understand some issues with GPU installations and CUDA versions. Logging is off by default, can be switched on in the mira.ini file.
- Licensing improvements:
  - For activated licenses, there is now an auto-update mechanism that pulls updated license from the activation server when the application starts. The application may be used without on-line connection - it is needed only once in two weeks.
  - Adding support for floating licenses obtained over network from a license server. Floating licenses are now checked out one per session.
- Fixed wrong file name of previous project used for saving new project with File/Save command
- Fixed a crash when preview image could not be loaded

## 1.1 10-sep-2018

- [confusion matrix view](#) showing detailed error information
  - interactive performance optimization in a confusion matrix (slider in right-click context menu or a mouse wheel on confmat entries)
  - confusion matrix shows normalized errors and precisions, absolute sample counts available as well
  - quickly switch to confmat with 'c' key and to spectral plot with 's' key
  - define [performance constraints](#) via double click on a confusion matrix field (create/remove constrain)
  - constraints may be adjusted live by Ctrl+mouse wheel
  - constraints may be enabled/disabled to understand available performance options
  - move between available solutions fulfilling all constraints with [ and ] shortcuts
- [preview image from user-adjustable R,G and B bands](#) when spectral cube is loaded
  - this view improves labeling experience for many material types that look similar in a single band but their differences may be highlighted in R,G,B view
- **undo/redo** for label painting speeds up labeling
- **image crop** providing significant memory use reduction and processing speedups
  - when a project with a cropped image is loaded, the original cube is loaded and cropped
  - original cube may be loaded as a new image and multiple crops from the same cube are supported
- including perClass Runtime DLL and example of spectral cube processing in C
  - support for both **single precision** and double precision pipelines (with a new perClass 5.4 Runtime)
  - significant speedup of exported classifiers
  - legacy export option supporting older deployed runtimes <= 5.2
- a **preview rotation** command allows one to fix the rotation between preview and spectral cube (e.g. on Specim IQ projects)
- adding an option to **exclude a class from training** (right-click in class list or press 'x')
  - this allows one to quickly check the impact of specific classes on the overall solution
- option to purchase a license online and directly turn the demo into a commercial product
- dialog to request Skype/Teamviewer session on start up
- fix for a wrong class index after removing a class
- fix for clear labels of an image

## 1.0 13-jul-2018

- fix for a dock shift bug (when resizing a docked window and clicking on the image, the docked pannel resized back)
- adding band line dragging by mouse
- adding max valid line which is automatically set on image load
- when user is on preview and tries painting, a dialog is shown to load the entire cube (allows quick image changing without load)

## 1.0 29-may-2018

- first public release

### Application Server

perClass Mira provides an Application Server functionality that enables remote control of perClass Mira GUI over TCP/IP networking protocol. **The use-case for the Application Server is a quick construction of live visual demonstrators with camera and perClass Mira processing in the custom control loop.**

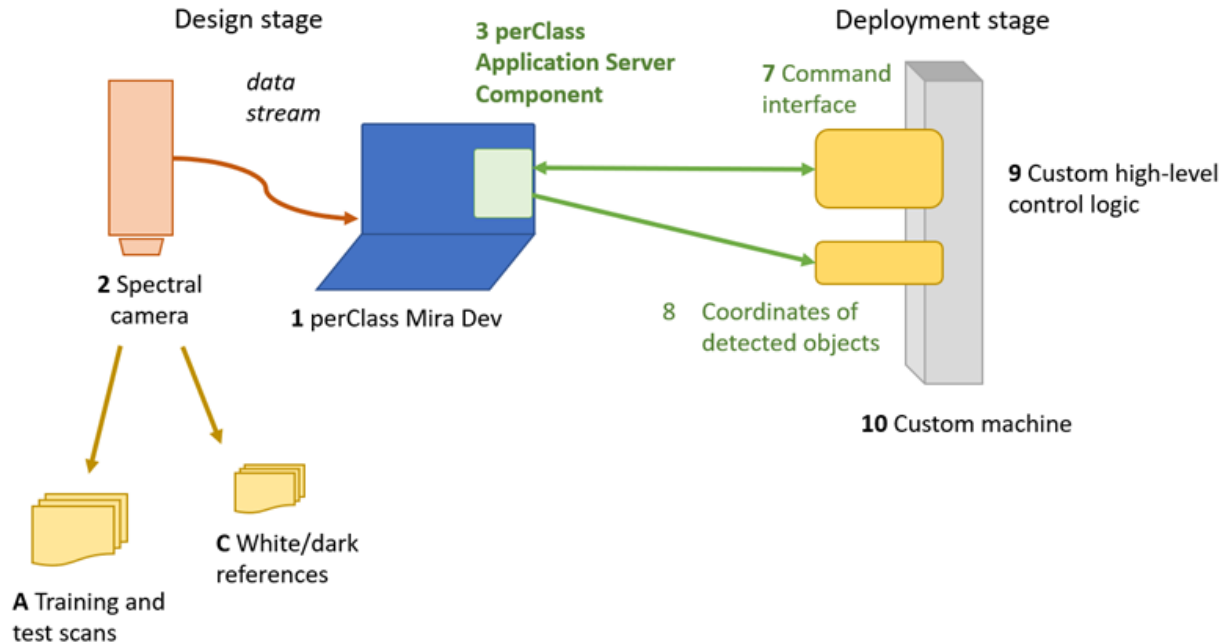
The following scheme clarifies the Application Server operation. The Application Server (3) is running under

perClass Mira Dev (1) environment. It listens to commands sent from a custom interface (7). The commands can control the attached camera (2). If the solution is able to detect objects in the live data stream, their coordinates are passed over a separate connection to a listening component (8) that can act on the detections.

The Custom Machine (10) is a demonstrator controller that may also involve an actuator such as a PLC or a robot arm. Custom application logic (9) can leverage the Application Server (3) to bring perClass Mira live processing and visualization capabilities in custom control loop using only simple text commands.



### perClass deployment setup on PC



## Enabling application server

Application server is only available in perClass Mira Dev product and requires presence of "appserver" licensing option in the license file.

### Check whether the current license offers Application Server functionality

- In a running perClass Mira instance, select *Help / Open license directory* command. A Windows Explorer window will open in the directory containing licenses and settings (C:\Users\USERNAME\AppData\Roaming\perClassBV)
- open the license file present (by default mira.lic) in a text editor
- The Application Server is available only in perClass Mira Dev licenses. Therefore, the license file needs to start with **LICENSE prsysd mira**
  - If the license file starts with **LICENSE prsysd mira.gui**, the current license is perClass Mira (GUI only) and cannot run camera acquisition or Application Server.
- the license options field should contain **appserver** string in order to use the Application Server
  - Note that to control acquisition, also acquisition support needs to be present (**acq** option)

### Enable Application Server function

By default, Application Server functionality is disabled. We can enable it in perClass Mira settings file (mira.ini).

- close any running instance of perClass Mira
- open the C:\Users\USERNAME\AppData\Roaming\perClassBV\mira.ini file in a text editor
- Edit the line with startCommandServer command so that it is enabled:
  - **startCommandServer=true**



- Save the mira.ini file
- Start perClass Mira
  - Select the connected camera for acquisition or a filereader

The perClass Mira output windows should list the note on open Application Server ports:

```

Output x
perClass Mira 4.2 (13-mar-2023) build 0814, Windows 10 (10.0), CPU+GPU
Settings loaded from 'C:/Users/pavel/AppData/Roaming/perClassBV/mira.ini'
CUDA: Platform: CUDA Runtime 11.2, Driver: 11060
License auto-update..ok
perClass Mira Development: Commercial license issued on 14-mar-2023, expiration on 28-mar-2023 (in 15 days)
Command TCP/IP server listening on 0.0.0.0 port 51234
Object detection TCP/IP server listening on 0.0.0.0 port 51300
Project type: perClass
CUDA: NVIDIA GeForce RTX 2070 Super, 8192 MB, CUDA Compute 7.5
Searching plugins at: C:/Program Files/perClass Mira/bin
Attempting to load acquisition plugin File Reader via File Reader 1.3.0...
Plugin loaded: miraacq_filereader_1.3.0.dll
perClass Mira 4.2 (20-jan-2023) build 0841, FileReader plugin 1.3.0 (1.0?)
Enabling source plugin File Reader via File Reader 1.3.0
Setting acquisition window height to 500 pixels/lines
  
```

When opening the application for the first time, Windows OS may request user consent to open the ports in a separate window.

If you cannot see the ports open, double check that you're not running any security software preventing that perClass Mira application opens TCP/IP ports.

## Communicating with the server

Application Server listens on TCP/IP port **51234**.

In order to communicate with the server, we need to send TCP/IP text commands from some external utility to perClass Mira. In our example, the utility may run on the same computer as perClass Mira. This is necessary. Typical demonstrators will run perClass Mira on a separate computer dedicated to a "second screen" and control it from already available control computer or PLC system orchestrating the entire process (belt control, actuators, lights etc.)

In the following example, we use the free **PacketSender** software you may download from:  
<https://packetsender.com/>

To prepare perClass Mira for running the Application Server session, we need to initialize an acquisition device (either a camera or a filereader).

Once we are able to start and stop acquisition from perClass Mira side, we may do the same remotely. Application Server provides a number of text commands that invoke actions.

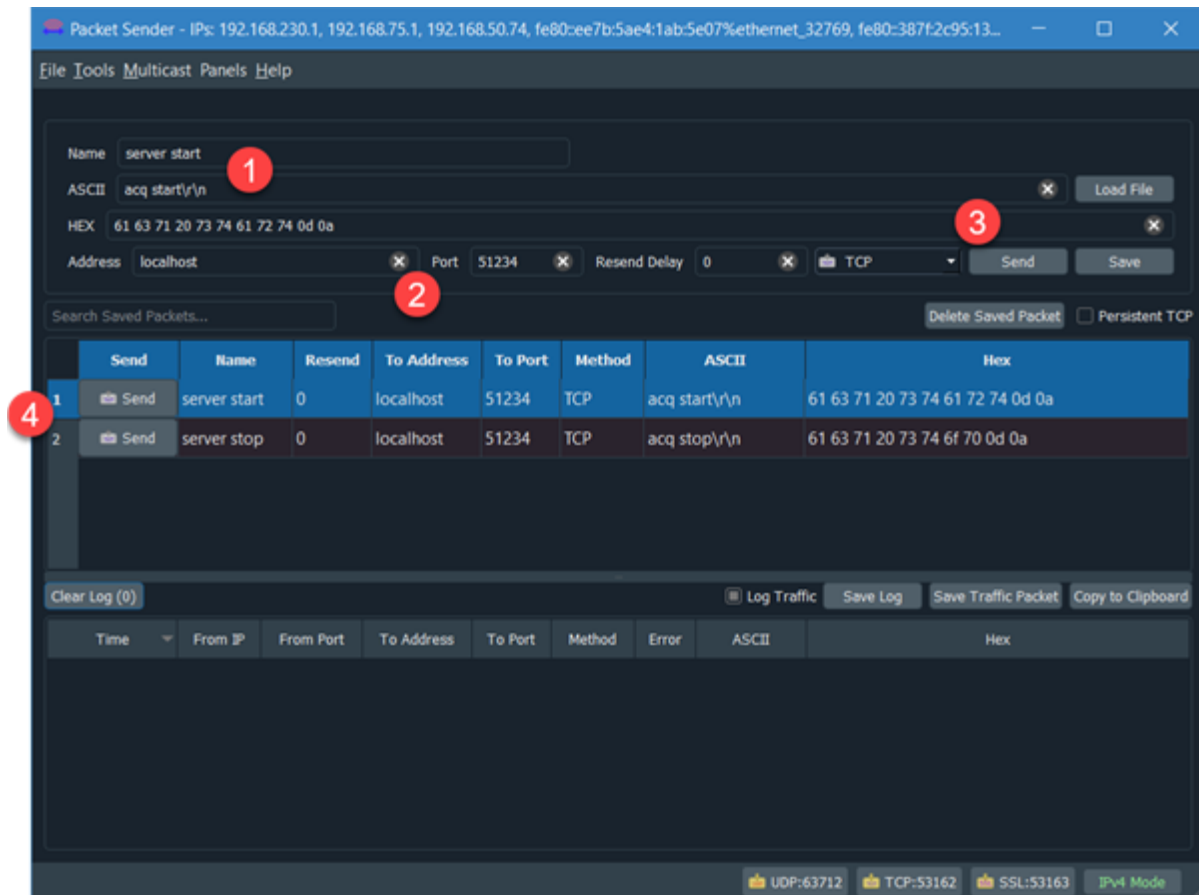
**TIP:** perClass Mira installation contains a ready-to-use data base of commands in PacketSender format in the lib directory. You may import these to a PacketSender session.

In PacketSender, we define two commands, namely "acq start" and "acq stop". For each command, its

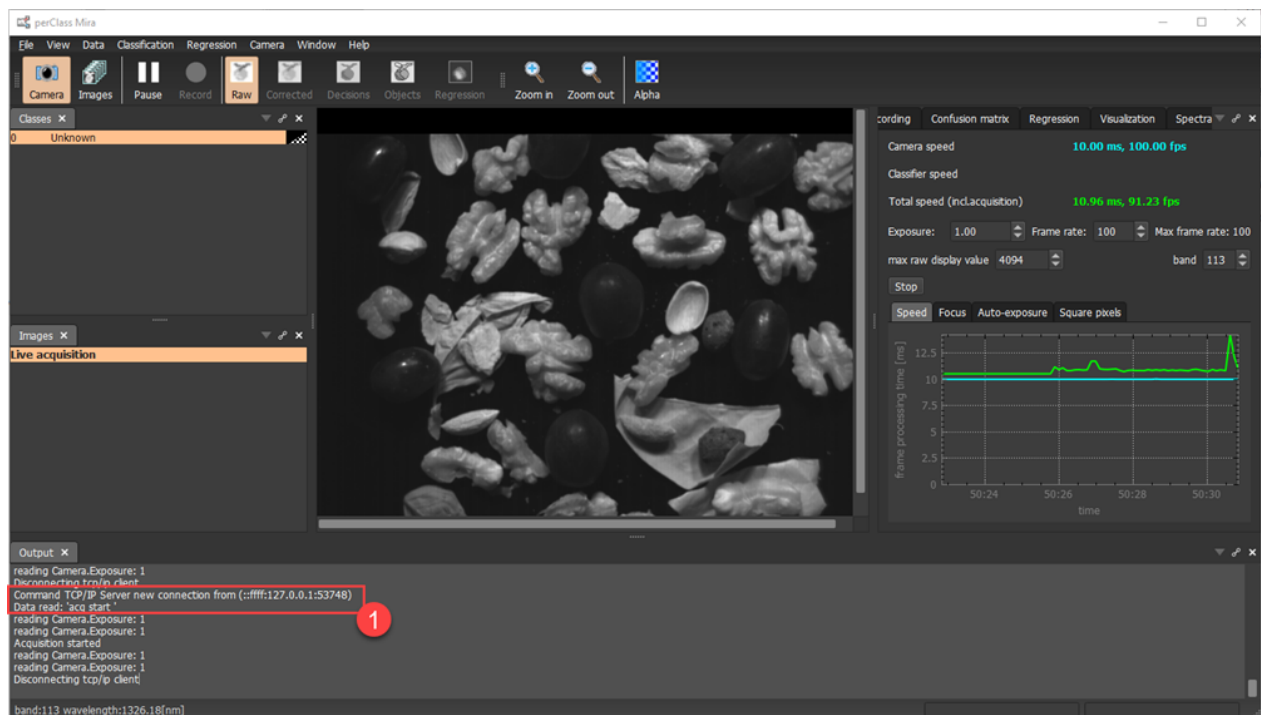
name and payload (content) are defined in the section <sup>1</sup>. Section <sup>2</sup> defines the Application Server machine address and port. In our situation, we fill "localhost" as both perClass Mira and the PacketSender run on the same machine.

We can send the command by pressing *Send* button <sup>3</sup>. The command can be saved for later use with

*Save* button <sup>4</sup>.

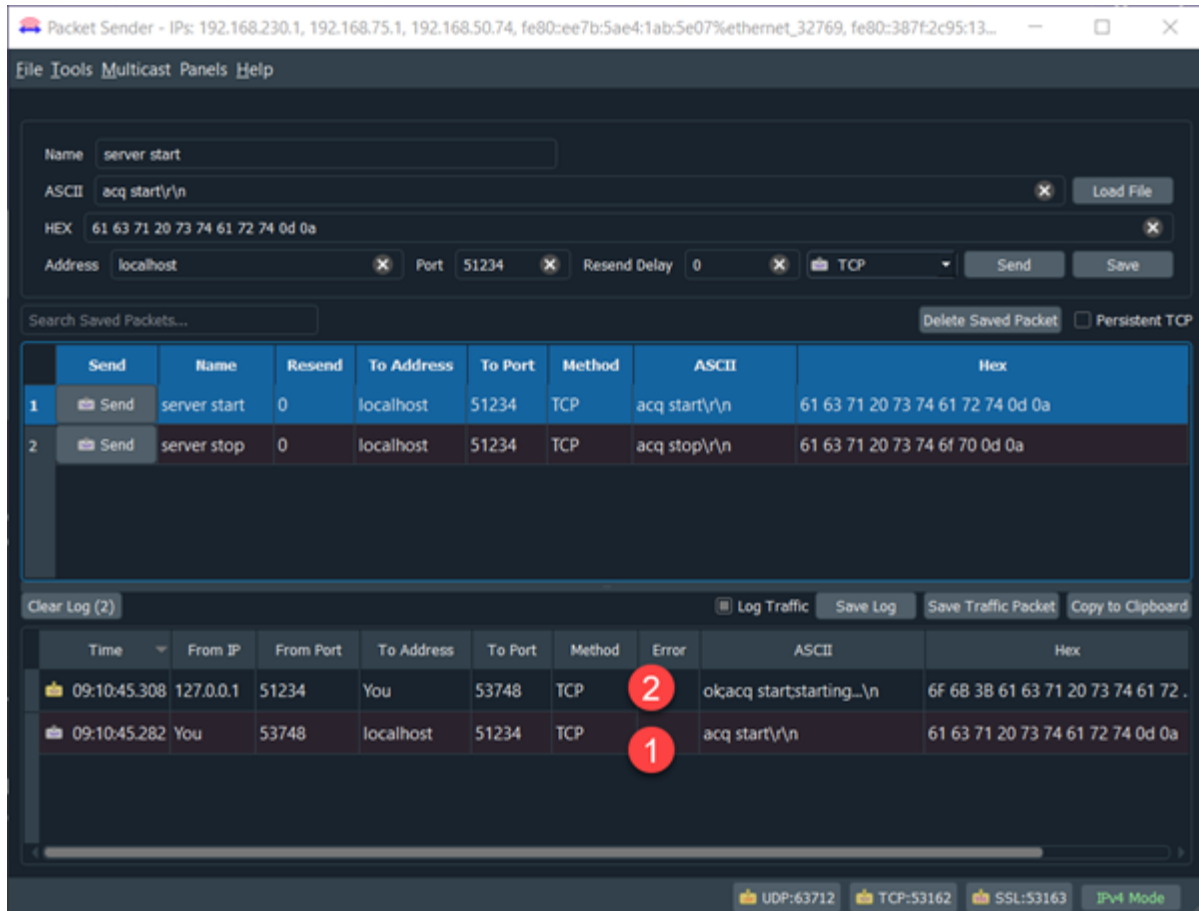


In perClass Mira window, we will see the text command listed in the *Output* window and the acquisition will start:



Below you can see, that the PacketSender window lists both the command sent and also the response

of the Application Server <sup>2</sup>. The response always starts with "ok;" or "error;" string denoting whether the command is understood, followed with the actual command string, another semicolon delimiter and an additional comment.



## Command list

Available commands of the Application Server

### Generic commands

- **acq start** - start acquisition
- **acq stop** - stop acquisition
- **acq state** - return acquisition state
  - example response: "ok;acq state;1"
    - 1 = acquisition initialized but not running
    - 2 = acquisition running
- **acq info enable** - enable object detection reporting on separate object stream
- **acq info disable** - disable object detection reporting on separate object stream
- **acq info totalsize** - return the total size of objects detected in the session
  - the total object size counter is reset when starting acquisition
- **acq darkref record** - record dark reference
  - Only full frame references are supported (non-uniformity)
  - reference recording does not automatically close the shutter. Use camera-specific shutter command before to close the shutter and reopen after the reference recording.
- **acq whiteref record** - record white reference

- Only full frame references are supported (non-uniformity)
- **view clear** - clear the display
- **view save FILENAME** - save current display into PNG file named FILENAME in current directory (top-level data directory)
- **view acq MODE** - set specific view mode
  - raw - view the raw data stream
  - corrected - view the reflectance corrected data stream (references need to be acquired)
  - decisions - show the decision layer (model needs to be defined)
  - objects - show object detections. The model and object segmentation needs to be defined in a model
  - waterfall - set the view type as waterfall (overwriting the data in a cyclical fashion)
  - belt - set the view type as a belt view (the stream moves like if watching belt from the top)
- **decisions** - return the comma-separated list of model decisions
  - Example output when running a four class classifier:  
"ok;decisions;background,leaves,shells,nuts\n"

### ***Control of perCass Mira Stage***

- **stage connect** - connect to the stage
- **stage disconnect** - disconnect from the stage
- **stage stop** - stop any stage movement
- **stage left | right | center** - move the stage right, left or center
- **stage position POS** - moves the stage to the position POS (in mm). If POS is omitted, the current position is returned
- **stage speed SP** - sets a specific speed SP (in mm/sec). If SP is omitted, the current speed is returned
- **stage scan** - working with the scanning area
  - **stage scan** - returns start and end position of the scanning area
  - **stage scan START END** - sets the start and end position of the scan area
  - **stage scan start** - moves the stage to the current START position
  - **stage scan end** - moves the stage to the current END position
- **stage cycle** - starts cycling between START and END scan area
- **stage white** - working with the area of the white reference
  - **stage white** - returns start and end position of the white reference
  - **stage white START END** - sets the start and end position of the white reference
  - **stage white start** - moves the stage to the current white reference START position
  - **stage white end** - moves the stage to the current white reference END position

### ***Camera specific commands - Headwall MV.X***

- **mvx Shutter.Close** - close MV.X shutter
- **mvx Shutter.Open** - open MV.X shutter

### ***Camera specific commands - Specim Specsensors SDK***

This specific interface mirrors command functionality of SpecSensor SDK. Please refer to SpecSensor documentation on the SDK commands for more details.

Basic commands to open and close the shutter:

- **specsensors Shutter.Open** - open the shutter
- **specsensors Shutter.Close** - close the shutter

Extended commands

- **specsensors Shutter.IsOpen** - return if the shutter is open
- **specsensors Shutter.IsConnected** - return if the shutter is connected.
- **specsensors Shutter.IsToggle** - return if the shutter is toggled (flipped)

- **specsensor Shutter.IsToggle X** - pass value to SpecSensor IsToggle
  - the value can be "enable", "on" or "true" or "disable", "off" or "false"
  - Example: "specsensor Shutter.IsToggle enable"

### Example communication using Tcl

This section shows simple communication with Application Server using Tcl command language. Tcl is a very simple scripting and command language.

For more information on Tcl, see

- Official website: <http://tcl.tk/>
- Windows binary distribution can be found here: <https://www.magicsplat.com/tcl-installer/index.html>

Other scripting environments, like Python, should be able to control Application Server in very similar fashion. The Tcl is chosen for its simplicity. We can get the communication going using few socket configuration commands.

### Setting up Application Server communication in Tcl

Creating a socket on localhost port 51234:

```
% set so [socket localhost 51234]
sock000001F0932F2CB0
```

Configuring the socket buffering to line:

```
% fconfigure $so -buffering line
```

Defining a new command called sendcmd that writes content into the socket, reads and returns the response:

```
% proc sendcmd {so cmd} {puts $so $cmd;gets $so data; return $data}
```

### Controlling Application Server from Tcl

We can now send commands to the Application Server and receive the responses:

```
% sendcmd $so {acq start}
ok;acq start;starting...
```

```
% sendcmd $so {acq stop}
ok;acq stop;stopped
```

### Receiving information on object detections

The Application Server provides the second TCP/IP channel where object detections are announced. We open the second Tcl comand shell, create an async socket and define a call back function that would be executed, when any data arrives. In the callback, we print the data sent by Application Server to the standard output. If the socket is closed, we close the processing.

We can past the following code to the Tcl command shall:

```
proc readData {serverChannel} {
    global end
    set noOfChars [gets $serverChannel data]
    if {$noOfChars!=-1} {
        puts "read: $data"
    }
}
```

```

        if {[eof $serverChannel]} {
            close $serverChannel
            set end 1
        }
    }
}
set sd [socket -async localhost 51300]
fconfigure $sd -blocking 0
fileevent $sd readable [list readData $sd]
vwait end

```

The first part defines the callback function. Then the socket is created and configured as non-blocking. The event callback is set on the socket connecting our callback function.

Finally, we enter event loop using "variable wait" vwait command. It blocks the system processing events until the variable "end" is created.

The complete application setup is shown in the following screenshot. Apart of the perClass Mira window, we

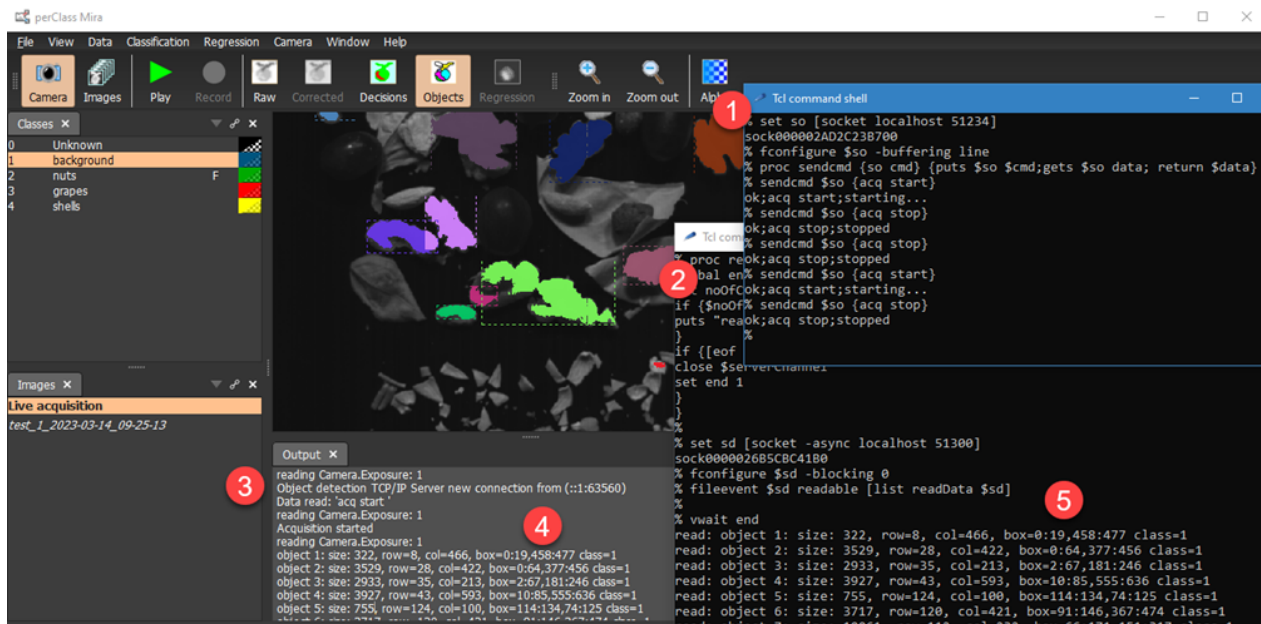
have two separate shells (one Tcl shell <sup>1</sup> to issue commands and one shell receiving object detections

<sup>2</sup>). Once we connect in <sup>2</sup> to the object detection channel on port 51300, perClass Mira *Output*

window displays the connection details <sup>3</sup>. In the command shell <sup>1</sup>, we can now start the acquisition using "acq start" command (note that also "acq info enable" command was run to enable object reporting and the display was set to show objects (not shown on the screenshot).

As soon as the objects are detected in the live stream <sup>4</sup>, the respective messages are sent over the

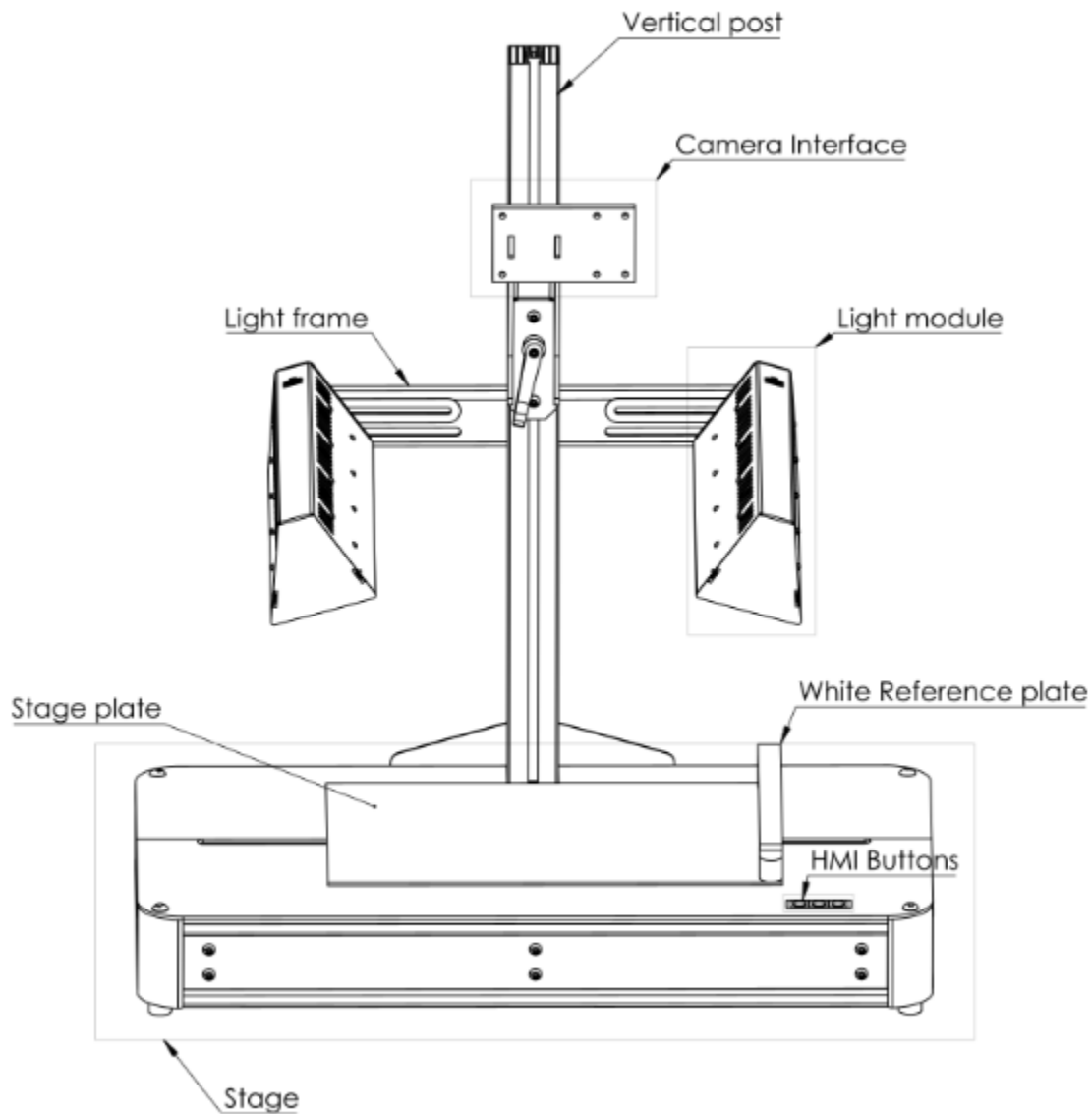
object detection socket, captured in <sup>5</sup> and printed to the standard output.



## perClass Stage

perClass Stage is a linear lab-scanning kit tightly integrated with perClass Mira. It supports a range of line-scan and snapshot spectral cameras through easy-to-exchange mounts.

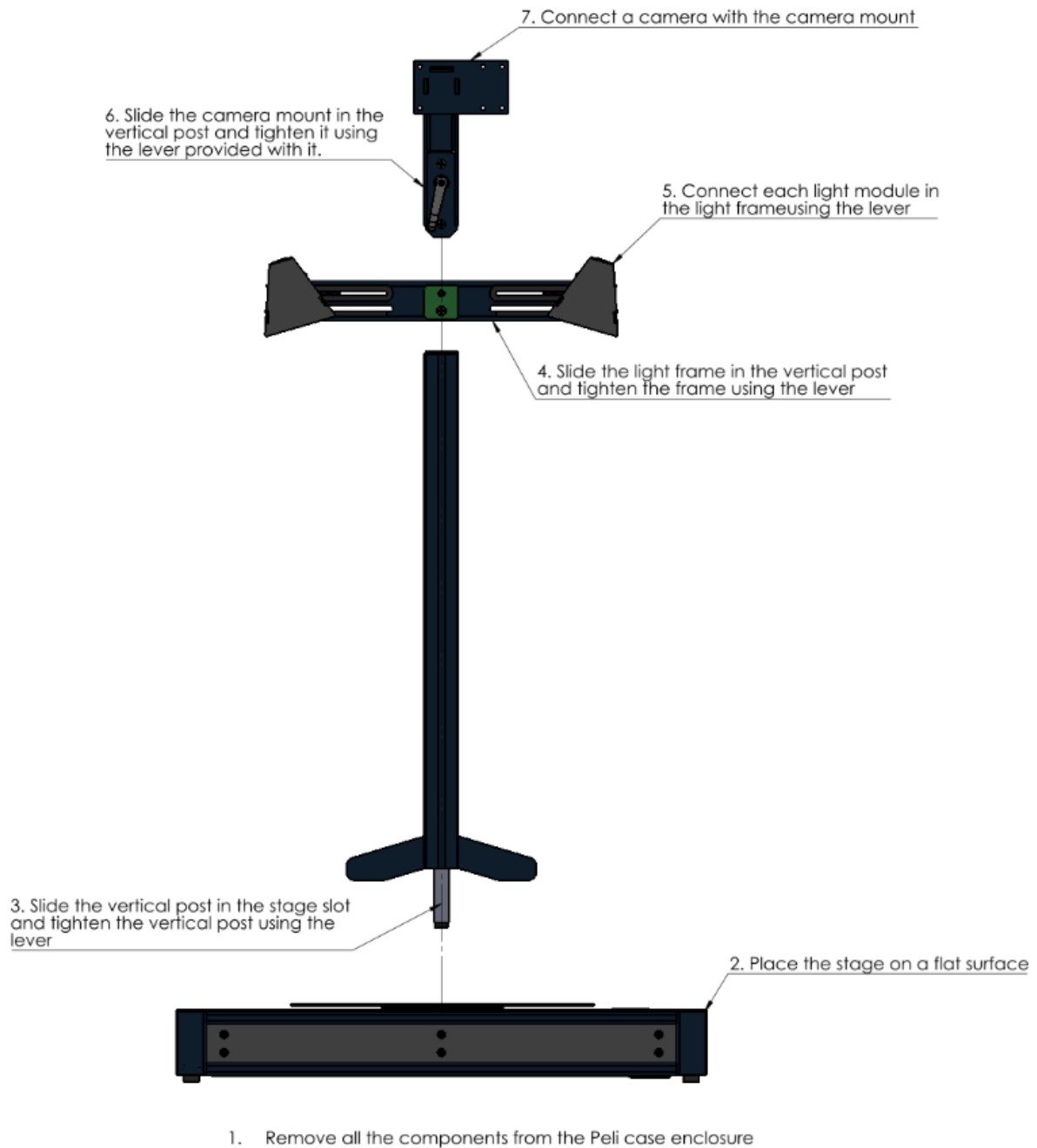
**WARNING!** Assembly should be performed by instructed personnel. Using excessive force for assembling or disassembling the stage has a risk of damage. All parts are designed such that the Linear stage can be assembled without the use of (power)tools. The interconnection is made using manual clamp levers.



### Assembling instructions

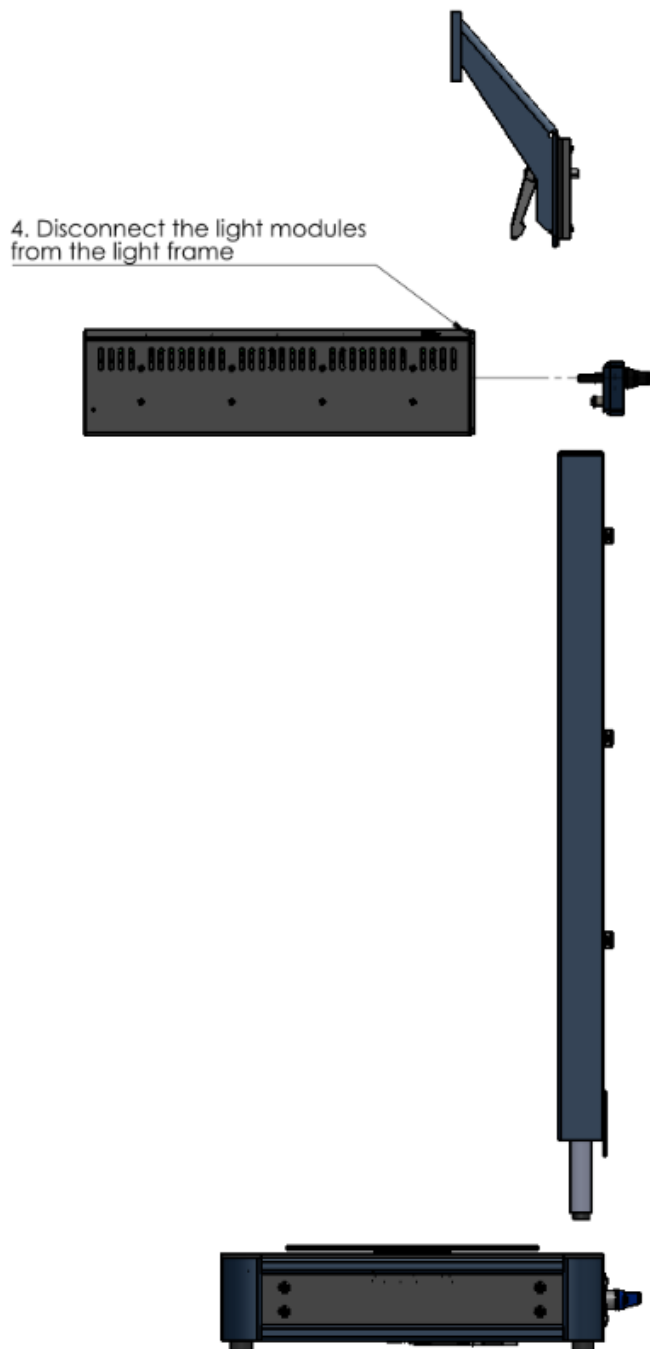
1. Remove all components from the Peli Case enclosure
2. Place the stage (base) on a flat surface
3. Slide the vertical post, with the alignment plate at the rear, in the stage slot
4. Tighten the vertical post by attaching the manual lever
5. Slide the light frame in the vertical post
6. Tighten the light frame with an M8 T-nut and manual clamping lever
7. Connect each light module using an M6 manual clamping lever from the inside of the light frame, to the light module thread
8. Slide the camera mount in the vertical post
9. Tighten the camera mount with an M8 T-nut and manual clamping lever
10. Adjust all parts if needed and secure all connections safely
11. Connect a camera to the camera mount
12. Connect the light power cables and secure the loose cables
13. Power the perClass stage





## Disassembling instructions

1. Depower the perClass Stage
2. **Wait** for the light modules to be of reasonable temperature
3. Disconnect all cables
4. Disconnect the camera mount plus camera
5. Disconnect the light modules
6. Disconnect the Light frame from the vertical post
7. Disconnect the Vertical post from the perClass Stage by unscrewing the manual lever (do not tighten the manual lever after removing the vertical post).



2. Disconnect all the cables from the light module to the base



## Supported cameras

### **Headwall**

- MV.C VNIR using MV.C holder
  - MV.C NIR using MV.C holder
  - MV.X using MV.X dedicated holder
- 
- Any camera using 1/4 standard mount

## Supported spectral cameras

### **Cubert**

- All cameras supported by CUVIS 3.2 and later SDK such as Cubert Ultris S5, X20

## HAIP

- BlackIndustry supported fully for live acquisition and processing
- BlackMobile supported to synchronize data from on-board storage and model building (via perClass project type)

## Headwall

- [MV.C VNIR](#)
- [MV.C NIR](#)
- [MV.X VNIR](#)
  - perClass Mira models can be deployed for on-board processing on the MV.X
  - MV.X cameras can be used as generic acquisition devices using Pleora eBUS (separate license needed)
- Cameras supported by the legacy HyperspecIII SDK, currently VNIR and NIR systems currently **excluding** the SWIR camera

## Imec

- For acquisition and processing: Imec Mosaic devices such as RedNIR and SWIR
- Starting from acquired scans: Imec SNAPSCAN

## Inno-spec

- RedEye2 supported using Pleora eBUS (separate license needed)

## [Resonon](#)

- Both VNIR and NIR systems using the legacy Resonon SDK

## Silios

- All Silios camera supported by Silios SDK

## Specim

- All cameras supported through Specim SpecSensor SDK (such as Specim FX10, FX17, FX50, SWIR)
- When using Pleora eBUS directly, all cameras that are GenICam compatible (such as FX10,FX17)

## Unispectral

- Unispectral MonarchII

## [Cubert](#)

In order to acquire data using Cubert cameras in perClass Mira, Cubert CUVIS SDK needs to be installed.

Installing CUVS SDK:

- Install CUVIS 3.2.0
- Make sure the directory containing “ cuvis.dll” is in the PATH environment variable: “ C:\Program Files\Cuvis\bin”
- Make sure the following directories exist: For newer plugins (may need to copy from CubertFuchsia):
  - C:\Program Files\Cuvis\factory
  - C:\Program Files\Cuvis\user

Connecting to the camera

- Make sure the property “ ULTRIS5\_GevSCPD” is set to “ 10000” in “ C:\Program Files\Cuvis\user\settings\ultris5.settings”
- Go to the ethernet adapter settings and setup a static IP address

- Set the IP to 192.168.200.5 (or anything in 192.168.200.0/24, except the device IP address)
- Set the subnet mask to 255.255.255.0
- If required set the gateway to 192.168.200.254 (or anything in 192.168.200.0/24, except the device and your IP address)
- You may set the IP and subnet using this command from Windows cmd shell, started as Administrator

```
netsh interface ip set address name="Ethernet" static 192.168.200.5
255.255.255.0
```

#### Extra hints

- Disable the power saving options for the ethernet port, for instance
  - Energy-Efficient Ethernet
  - Green Ethernet
  - Power Saving Mode
- Make sure firewall is not blocking perClass Mira on public networks
- Make sure that there are no other Cubert plugin versions in the Mira installation directory

## Pleora eBUS

perClass Mira supports acquisition from eBUS 5.1, 6.1 and 6.3. Note that only one eBUS release can be installed at a time. It is advised to use the newest release available.

- Install Pleora eBUS Runtime
  - You may download [Pleora eBUS Runtime 6.3 here](#)
- Computer restart is needed after eBUS installation in order to make kernel drivers active
- Make sure that no machine vision or camera software is providing additional, incompatible, version of Pleora eBUS.
  - Specifically, make sure that Specim Specsensor or Imec SDK are not on the path

**TIP:** You may append a string to each of the paths in system Path environment variable to make it not found. For example appending `_DISABLED`. This allows you to easily bring these dependencies back when you wish to use other camera types

## Headwall

### Headwall MV.X

Headwall MV.X integrated system may be used together with perClass Mira in two different modes:

1. As a standalone camera, using Pleora eBUS
2. Deploying solutions, built in perClass Mira, onboard of the MV.X unit using the integrated perClass Mira Runtime

#### **Standalone camera use**

- [Pleora eBUS should be installed](#)
- Connect Ethernet cable to port 2 on the MV.X unit
- Set the computer IP to 10.0.65.1 (or anything in 10.0.65.0/24 range, **except** 10.0.65.50 that is dedicated to the MV.X web interface)
  - Set the subnet mask to 255.255.255.0
  - If required set the gateway to 10.0.65.254 (or anything in 10.0.65.0/24, except 10.0.65.50 and your IP address)

#### **On-board processing using perClass Mira Runtime**

- Solutions, exported from perClass Mira can be run on-board of the MV.X system using the integrated runtime
  - Export the classification model using *Classifier / Export classifier as a pipeline / perClass Mira Runtime (single precision)* command

## Headwall MV.C NIR

### **Installation instructions**

- [Install Pleora eBUS 6.1 or 5.1](#)

### **Network interface settings**

- Go to the ethernet adapter settings and setup a static IP address
  - Set the IP to 169.254.1.1 (or anything in 169.254.0.0/16, except the device IP address)
  - Set the subnet mask to 255.255.0.0
  - If required set the gateway to 169.254.254.254 (or anything in 169.254.0.0/16, except the device your IP address)

### **Avoiding dropped frames**

- Disable the power saving options for the ethernet port
  - Energy-Efficient Ethernet
  - Green Ethernet
  - Power Saving Mode
- Make sure no firewall or security application is blocking perClass Mira application

## Headwall MV.C VNIR

### **Installation instruction for Headwall MV.C VNIR camera**

- Download and install Ximea xiAPI SDK: <https://www.ximea.com/support/wiki/apis/xiAPI>
- Put the C:\XIMEA\API\xiAPI path to System Path variable:
  - Open Windows Settings, fill "env" and select *Edit environment variables for your account*
  - Edit the *Path* environment variable
  - Make sure the C:\XIMEA\API\xiAPI directory is on the path
  - Start perClass Mira application again for the change to take effect

## **Troubleshooting**

### **Camera returns -503 error in perClass Mira**

This may be caused by improper USB connection. Please make sure the camera is connected directly to the PC, not to a USB hub. If possible, try different USB connections.

## Resonon

perClass Mira supports Resonon cameras through Resonon SDK provided by camera distributor or Resonon Inc.

In perClass Mira installation, two separate acquisition plugins are included, one for the VNIR Basler-based cameras and one for the NIR AlliedVision-based sensors.

In order to to acquire data from Resonon camera:

- Install Resonon SDK (3.8 or higher)
- Add path to C:\Program Files\ResononAPI\bin64 directory to Windows Path environmental variable

**For NIR cameras**

- the AlliedVision VimbaC SDK needs to be installed (it is part of Resonon SDK installation)
- the following path needs to be present in Windows Path environmental variable: C:\Program Files\Allied Vision\Vimba\_6.0\VimbaC\Bin\Win64

**perClass Camera API**

## Versioning

- `miraacq_GetVersion`
- `miraacq_GetAPIVersion`

## Initialization and cleanup

- `miraacq_Init`
- `miraacq_Release`

## Error handling

- `miraacq_GetErrorCode`
- `miraacq_GetErrorMsg`

## Acquisition device selection

- `miraacq_ScanDevices`
- `miraacq_GetDeviceCount`
- `miraacq_GetDeviceName`
- `miraacq_OpenDevice`
- `miraacq_CloseDevice`

## Acquisition initialization

- `miraacq_InitializeAcquisition`
- `miraacq_GetFrameSize`
- `miraacq_GetFrameWidth`
- `miraacq_GetFrameHeight`
- `miraacq_GetFrameBands`
- `miraacq_GetFrameDataType`
- `miraacq_GetFrameDataLayout`

## Wavelength and spectral resampling control (optional)

- `miraacq_CanReturnWavelengths`
- `miraacq_GetFrameWavelength`
- `miraacq_SetResamplingWavelengthCount`
- `miraacq_SetResamplingWavelength`
- `miraacq_SetResampling`

## Running acquisition and obtaining data

- `miraacq_StartAcquisition`
- `miraacq_GetFrame`
- `miraacq_StopAcquisition`

## Setting acquisition parameters

- `miraacq_SetExposure`
- `miraacq_GetExposure`
- `miraacq_SetFrameRate`
- `miraacq_GetFrameRate`

**miraacq\_Init**

Initialize the camera acquisition environment.

```
makernel* miraacq_Init(const char* path)
```



*Input:* Path to a directory with a license file

*Output:* Runtime environment pointer.

*Description:*

The `miraacq_Init` function initializes the camera acquisition environment. It returns a pointer used for any other API function that interacts with the library. We refer to this pointer a "runtime pointer" in the acquisition library reference. Please note this is referring to "acquisition runtime" which differs from "processing runtime" of perClass Mira. The processing runtime is initialized by [mira\\_Init](#) function. Its API is [described here](#).

The acquisition runtime exposes data acquisition from a camera via specific acquisition plugins. Each acquisition plugin name starts, by convention, with `miraacq_` string followed by the plugin type and version number. For example, the plugin exposing Headwall MV.C VNIR camera based on Ximea API is named `miraacq_ximea_1.6.0.dll`

The input is a path to a directory where a license file with `.lic` extension can be found. Pass `."` for the current directory.

After initialization, `miraacq_GetErrorMsg` provides welcome string listing software version or error message.

*Example:*

```
makernel* pma=miraacq_Init(".");
printf("Init: %s", miraacq_GetErrorMsg(pma));
if( pma==NULL ) return;
```

## `miraacq_GetVersion`

Return version of the acquisition runtime

```
const char* miraacq_GetVersion( )
```

*Input:* None

*Output:* Version string

*Description:*

`miraacq_GetVersion` returns version string together with the release date.

*Example:*

```
const char* str=miraacq_GetVersion();
printf("Version: '%s'\n",str);
```

*Output:*

Version: '4.2 (16-feb-2023)'

## `miraacq_GetAPIVersion`

Return API version information

```
const char* miraacq_GetAPIVersion(int* pApi,int* pStep,int* pRev);
```

*Input:*

`pApi` - pointer to integer: Major outer API for all supported plugins

`pStep` - pointer to integer: Functionality version of this specific plugin (perClass Mira functionality changes)

`pRev` - pointer to integer: Revision of the functionality (only the plugin changes)

*Output:* Full version string including perClass Mira build info, lugin info and third party SDK version (if

linked)

*Description:*

`miraacq_GetAPIVersion` returns full version string including information about the plugin and the linked third-party SDK used. Via three integer pointers, it also provides the numerical API versioning.

*Example:*

```
int api, rev, step;
const char* str=miraacq_GetAPIVersion(&api,&rev,&step);
printf("API version: %d.%d.%d (%s)\n",api,rev,step,str);
```

*Output:*

API version: 1.6.0 (perClass Mira 4.2 (16-feb-2023) build 1548, Ximea plugin 1.6.0 (Ximea API 4.24.00.03))

## `miraacq_GetRecorderType`

Return string name of the acquisition backend

```
const char* miraacq_GetRecorderType()
```

*Input:* None

*Output:* Recorder (acquisition plugin) type string

*Description:*

`miraacq_GetRecorderType` returns version string together with the release date.

*Example:*

```
const char* str=miraacq_GetRecorderType();
printf("Type: '%s'\n",str);
```

*Output:*

Type: 'Ximea'

## `miraacq_GetErrorCode`

Return error code

```
int miraacq_GetErrorMsg(makernel* pma)
```

*Input:* Runtime environment pointer

*Output:* Error code

*Description:*

`miraacq_GetErrorCode` returns error code. For a specific string description, `miraacq_GetErrorMsg`

## `miraacq_GetErrorMsg`

Returns error message.

```
const char* mira_GetErrorMsg(makernel* pma)
```

*Input:* Runtime environment pointer

*Output:* Error message string

*Description:*

`miraacq_GetErrorMsg` returns error message as string. For a specific error code, use `miraacq_GetErrorCode`

**miraacq\_ScanDevices**

Scan available acquisition devices

```
int mira_ScanDevices(makernel *pma)
```

*Input:*

- Runtime environment pointer pma

*Output:* Result: MIRA\_OK or error code

*Description:*

`miraacq_ScanDevices` searches for available acquisition devices. After calling `miraacq_ScanDevices`, one can get device count using `miraacq_GetDeviceCount` and names with `miraacq_GetDeviceName`.

*Example:*

```
MIRAACQ_CHECK( miraacq_ScanDevices(pma) );
const int devCount=miraacq_GetDeviceCount(pma);
printf("\n%d devices:\n",devCount);
for(int i=0;i<devCount;i++) {
    printf("%d : %s\n",i, miraacq_GetDeviceName(pma,i));
}
...
Error:
    if( res!=MIRA_OK ) {
        printf("Error %d: %
s",miraacq_GetErrorCode(pma),miraacq_GetErrorMsg(pma));
    }
    miraacq_Release(pma);
```

The `MIRAACQ_CHECK` macro checks the output result. If error occurs, the program flow is terminated. See `miraacq.h` definition. Note that the `MIRAACQ_CHECK` macro expects an Error label defined. The following code can display the error message and release the runtime.

**miraacq\_GetDeviceCount**

Returns the number of acquisition devices found

```
int miraacq_GetDeviceCount(makernel* pma)
```

*Input:* Runtime environment pointer

*Output:* Number of devices found

*Description:*

`miraacq_GetDeviceCount` returns the number of devices found by `miraacq_ScanDevices`.

**miraacq\_GetDeviceName**

Returns the name of a specific acquisition device

```
const char* miraacq_GetDeviceName(makernel* pma,int deviceInd)
```

*Input:*

- Runtime environment pointer
- Device index

*Output:* String name of a device

*Description:*

[miraacq\\_GetDeviceName](#) returns the name for a specific device. Before using [miraacq\\_GetDeviceName](#) or [miraacq\\_GetDeviceCount](#), the device list needs to be constructed by [miraacq\\_ScanDevices](#).

## **[miraacq\\_OpenDevice](#)**

Opens specific acquisition device

```
int miraacq_OpenDevice(makernel* pma, int devInd)
```

*Input:*

- Runtime environment pointer
- Device index

*Output:* Result code (MIRA\_OK or error)

*Description:*

[miraacq\\_OpenDevice](#) opens the specified acquisition device. The device list needs to be constructed by [miraacq\\_ScanDevices](#).

*Example:*

```
MIRAACQ_CHECK( miraacq_ScanDevices(pma) );
int deviceInd=0; // here we open the device 0
MIRAACQ_CHECK( miraacq_OpenDevice(pma,deviceInd) );
printf("Device opened: %d '%s'\n",deviceInd,miraacq_GetDeviceName(pma,deviceInd));
```

The [MIRAACQ\\_CHECK](#) macro checks the output result. See the usage example in [miraacq\\_ScanDevices](#) and [miraacq.h](#) for details.

## **[miraacq\\_CloseDevice](#)**

Closes specific acquisition device

```
int miraacq_CloseDevice(makernel* pma, int devInd)
```

*Input:*

- Runtime environment pointer
- Device index

*Output:* Result code (MIRA\_OK or error)

*Description:*

[miraacq\\_CloseDevice](#) closes the specified acquisition device. Device needs to be previously opened using [miraacq\\_OpenDevice](#)

## **[miraacq\\_DeviceIsSnapshot](#)**

Returns if a currently opened device is snapshot camera

```
int mira_DeviceIsSnapshot(makernel* pma);
```

*Input:*

- Runtime environment pointer

*Output:* 1 if the currently opened device is a snapshot or 0 if not

This function returns the type of currently opened device. If the device is a snapshot camera (providing a full spectral cube with all bands for all pixels in a single call) or not. Devices that are not snapshots are "line-scans" i.e. cameras that provide single pixel line and all spectral bands. To form a cube, a line scan camera needs to acquire multiple spectral frames. Therefore, while the width (pixel count) and band count of line-scan camera is defined by the sensor, its height is 1 (single frame at a time).

### **miraacq\_InitializeAcquisition**

Initializes the acquisition

```
int miraacq_InitializeAcquisition(makernel* pma)
```

*Input:*

- Runtime environment pointer

*Output:* Result code (MIRA\_OK or error)

*Description:*

`miraacq_InitializeAcquisition` is needed in order to define geometry and data type of data coming from the sensor.

### **miraacq\_GetFrameSize**

Get the size of the acquired image in bytes

```
int miraacq_GetFrameSize(makernel* pma);
```

*Input:*

- Runtime environment pointer

*Output:* Size in bytes width if > 0 or an error code if < 0

After the acquisition is successfully initialized by `miraacq_InitializeAcquisition` this function returns the number of bytes returned by the sensor during acquisition.

### **miraacq\_GetFrameWidth**

Get the width of the data returned by the sensor

```
int miraacq_GetFrameWidth(makernel* pma);
```

*Input:*

- Runtime environment pointer

*Output:* Image width if > 0 or an error code if < 0

After the acquisition is successfully initialized by `miraacq_InitializeAcquisition` this function returns the width (number of pixels) returned by the sensor during acquisition.

### **miraacq\_GetFrameHeight**

Get the height of the data returned by the sensor

```
int miraacq_GetFrameHeight(makernel* pma);
```

*Input:*

- Runtime environment pointer

*Output:* Image height if > 0 or an error code if < 0

After the acquisition is successfully initialized by [miraacq\\_InitializeAcquisition](#) this function returns the height in pixels returned by the sensor during acquisition. For snapshot cameras, this corresponds to the height of the cube returned in each acquisition step. For line-scans, this is fixed to 1 as only one spectral frame is returned in each acquisition.

## **[miraacq\\_GetFrameBands](#)**

Get the number of spectral bands of the data returned by the sensor

```
int miraacq_GetFrameBands(makernel* pma );
```

*Input:*

- Runtime environment pointer

*Output:* Number of spectral bands if > 0 or an error code if < 0

After the acquisition is successfully initialized by [miraacq\\_InitializeAcquisition](#) this function returns the number of spectral bands returned in each acquisition step

## **[miraacq\\_GetFrameDataType](#)**

Get the data type of the acquired image

```
int miraacq_GetFrameDataType(makernel* pma );
```

*Input:*

- Runtime environment pointer

*Output:* Integer denoting the data type

```
ACQ_DATATYPE_UNKNOWN  0
ACQ_DATATYPE_UINT16   1
ACQ_DATATYPE_FLOAT     2
ACQ_DATATYPE_UINT8     3
```

After the acquisition is successfully initialized by [miraacq\\_InitializeAcquisition](#) this function returns the data type of content returned in acquisition.

## **[miraacq\\_GetFrameDataLayout](#)**

Get the data layout of the acquired image

```
int miraacq_GetFrameDataLayout(makernel* pma );
```

*Input:*

- Runtime environment pointer

*Output:* Integer denoting the data layout in memory

```
ACQ_DATA_LAYOUT_BIP      1 /* spectrum-by-spectrum (dimensions: bands-width-
height) */
ACQ_DATA_LAYOUT_BIL      2 /* frame-by-frame (dimensions: width-bands-height)
*/
ACQ_DATA_LAYOUT_BSQ      3 /* spatial frame by frame (dimensions: width-height-
```

```
bands) */
```

After the acquisition is successfully initialized by [miraacq\\_InitializeAcquisition](#) this function returns the data layout of image returned in acquisition.

For line-scans, typical layout is BIL where all pixels (width) and all bands are returned each time. Spectral cube is created by collating multiple such spectral frames, each with unit height.

For snapshot systems that perform band scanning, each band is appended internally. This results in BSQ data layout (width x height x bands).

For some snapshot systems, the data is already reorganized into BIP layout where the spectral data of each pixel is located next to each other in memory (bands x width x height).

## [miraacq\\_CanReturnWavelengths](#)

Check if the device can return the wavelength information for each spectral band

```
int miraacq_CanReturnWavelengths(makernel* pma);
```

*Input:*

- Runtime environment pointer

*Output:* 1 if the device can return wavelength info per band and 0 otherwise

After the acquisition is successfully initialized by [miraacq\\_InitializeAcquisition](#) this function returns information if the device can provide wavelength information per spectral band.

## [miraacq\\_GetFrameWavelength](#)

Return wavelength in nanometers for a specific spectral band

```
int miraacq_GetFrameWavelength(makernel* pma, int bandInd, double* pWavelength);
```

*Input:*

- Runtime environment pointer
- bandInd - integer index of a spectral band (zero-based i.e. 0.. number of bands -1 )
- pWavelength- pointer to a double-precision value (to output the wavelength value in nanometers)

*Output:* Result: MIRA\_OK or error code

After the acquisition is successfully initialized by [miraacq\\_InitializeAcquisition](#) this function allows one to iterate over all spectral bands and retrieve the respective wavelength information.

## [miraacq\\_SetResamplingWavelengthCount](#)

Defines how many output wavelengths will be provided

```
int miraacq_SetResamplingWavelengthCount(makernel* pma, int count);
```

*Input:*

- Runtime environment pointer
- Integer: Number of output wavelengths after resampling

*Output:* Result code (MIRA\_OK or error)

This function starts the resampling step. By resampling, we refer to spectral resampling or interpolation.



The user may specify output wavelength list shared by all devices. This function initiates the procedure to define resampling by specifying how many outputs wavelengths will be provided. It should be followed by [miraacq\\_SetResamplingWavelength](#) calls to provide individual wavelength values in nanometers.

## **[miraacq\\_SetResamplingWavelength](#)**

Defines output wavelength value

```
int miraacq_SetResamplingWavelength(makernel* pma, int bandInd, double wavelength);
```

*Input:*

- Runtime environment pointer
- Integer: Index of an output band for which we're setting the wavelength value
- Double: Wavelength value in nm

*Output:* Result code (MIRA\_OK or error)

This function specifies wavelength values for desired output bands. The band indices are zero based: 0 .. count -1, where the count is defined using [miraacq\\_SetResamplingWavelengthCount](#) function. The wavelength values are provided in nanometers.

In order to use spectral resampling, the sensor needs to be calibrated, i.e. it needs to provide the mapping of spectral bands (pixels) to wavelengths.

Wavelength values, provided by this function, must also lay within the range of device wavelengths. This is because spectral interpolation not extrapolation is adopted.

## **[miraacq\\_SetResampling](#)**

Enables or disables resampling

```
int miraacq_SetResampling(makernel* pma, int enable);
```

*Input:*

- Runtime environment pointer
- Integer: 1 to enable or 0 to disable resampling

*Output:* Result code (MIRA\_OK or error)

This function enables or disables spectral resampling. The output wavelengths need to be specified using [miraacq\\_SetResamplingWavelengthCount](#) and [miraacq\\_SetResamplingWavelength](#) functions.

## **[miraacq\\_StartAcquisition](#)**

Starts the acquisition

```
int miraacq_StartAcquisition(makernel* pma)
```

*Input:*

- Runtime environment pointer

*Output:* Result code (MIRA\_OK or error)

*Description:*

[miraacq\\_StartAcquisition](#) starts the acquisition process. The acquisition needs to be successfully initialized using [miraacq\\_InitializeAcquisition](#).

## **[miraacq\\_GetFrame](#)**

Return new spectral data from sensor

```
int miraacq_GetFrame(makernel* pma, void* pBuf, size_t* pFrameID, int timeout );
```

*Input:*

- Runtime environment pointer
- Pointer to external buffer with raw spectral frame data
- Pointer to size\_t value to store frame id of the returned frame
- Integer timeout

*Output:* Result code (MIRA\_OK or error)

*Description:*

`miraacq_GetFrame` returns new data from spectral camera. The function can be only called after acquisition is initialized and started. The provided buffer pointed to by `pBuf` needs to hold at least number of bytes returned by `miraacq_GetFrameSize`. The content of `pFrameID` is set to the ID of a returned frame. On systems that support internal frameID, this value can be used for dropped frame detection.

*MV.C VNIR:*

The timeout value needs to be non-zero.

## `miraacq_StopAcquisition`

Starts the acquisition

```
int mira_StopAcquisition(mrkernel* pmr)
```

*Input:*

- Runtime environment pointer

*Output:* Result code (MIRA\_OK or [error](#))

*Description:*

`mira_StopAcquisition` stops the acquisition process. Statistics on number of processed frames, speed per frame and frame-rate is available via a subsequent `mira_GetErrorMsg` call.

*Example:*

Average alg time: 2.79676 ms/frame (357.557 fps), processed 1500 frames, first 500 skipped for warm-up.

## `miraacq_SetExposure`

Sets the exposure time

```
int miraacq_SetExposure(makernel* pma, double val);
```

*Input:*

- Runtime environment pointer
- Double: Exposure value in ms

*Output:* Result code (MIRA\_OK or error)

This function sets the exposure (integration time) in milliseconds. It can be used only after `miraacq_InitializeAcquisition`. It may be using during running acquisition (after `miraacq_StartAcquisition`).

## `miraacq_GetExposure`

Gets the exposure time

```
int miraacq_GetExposure(makernel* pma, double* pVal);
```

*Input:*

- Runtime environment pointer
- pointer to double: Output value

*Output:* Result code (MIRA\_OK or error)

This function gets the current exposure (integration time) value in milliseconds. It can be used only after [miraacq\\_InitializeAcquisition](#). It may be using during running acquisition (after [miraacq\\_StartAcquisition](#)).

## **[miraacq\\_SetFrameRate](#)**

Sets the frame rate in frames per second

```
int miraacq_SetFrameRate(makernel* pma, double val);
```

*Input:*

- Runtime environment pointer
- Double: Frame rate value in ms

*Output:* Result code (MIRA\_OK or error)

This function sets the frame raw (integration time) in frames per second. It can be used only after [miraacq\\_InitializeAcquisition](#). It may be using during running acquisition (after [miraacq\\_StartAcquisition](#)).

## **[miraacq\\_GetFrameRate](#)**

Gets the exposure time

```
int miraacq_GetFrameRate(makernel* pma, double* pVal);
```

*Input:*

- Runtime environment pointer
- pointer to double: Output value

*Output:* Result code (MIRA\_OK or error)

This function gets the current frame rate value in frames per second. It can be used only after [miraacq\\_InitializeAcquisition](#). It may be using during running acquisition (after [miraacq\\_StartAcquisition](#)).

## **[miraacq\\_Release](#)**

Release runtime internal session and clean resources.

```
void miraacq_Release(makernel* pma)
```

*Input:*

- Runtime environment pointer

*Output:* None

*Description:*

[miraacq\\_Release](#) ends the session and releases all memory allocated by the runtime.

## perClass Mira Runtime API

Initialization and cleanup

- [mira\\_Init](#)
- [mira\\_Release](#)

Error handling

- [mira\\_GetErrorCode](#)
- [mira\\_GetErrorMsg](#)

Computational device selection

- [mira\\_RefreshDeviceList](#)
- [mira\\_GetDeviceCount](#)
- [mira\\_GetDeviceName](#)
- [mira\\_SetDevice](#)

Loading model

- [mira\\_LoadModel](#)
- [mira\\_LoadCorrection](#)

Data processing - querrying input data parameters

- [mira\\_GetInputWidth](#)
- [mira\\_SetInputWidth](#)
- [mira\\_GetInputHeight](#)
- [mira\\_GetInputBands](#)
- [mira\\_GetInputDataType](#)
- [mira\\_GetInputDataLayout](#)

Data processing

- [mira\\_SetSegmentation](#)
- [mira\\_StartAcquisition](#)
- [mira\\_ProcessFrame](#)
- [mira\\_ProcessCube](#)
- [mira\\_StopAcquisition](#)
- [mira\\_SaveImage](#)

Processing results

Pixel classification

- [mira\\_GetFrameDecisions](#)
- [mira\\_GetDecCount](#)
- [mira\\_GetDecName](#)
- [mira\\_GetDecColor](#)

Object segmentation

- [mira\\_GetMaskType](#)
- [mira\\_GetObjCount](#)
- [mira\\_GetObjDataInt](#)
- [mira\\_GetObjDataClassSize](#)
- [mira\\_GetObjDataClassFrac](#)
- [mira\\_SetMinObjSize](#)

Regression

- [mira\\_GetRegVarCount](#)
- [mira\\_GetRegVarName](#)
- [mira\\_GetObjDataRegOutput](#)
- [mira\\_GetFrameRegOutputVar](#)

## mira\_Init

Initialize runtime environment.

```
mrkernel* mira_Init(const char* path)
```

*Input:* Path to a directory with a license file

*Output:* Runtime environment pointer

*Description:*

The mira\_Init function initializes runtime environment. It returns a pointer used for any other API function that interacts with the runtime.

The input is a path to a directory where a license file with .lic extension can be found. Pass "." for the current directory.

After initialization, mira\_GetErrorMsg provides welcome string listing software version or error message.

*Example:*

```
mrkernel* pmr=mira_Init(".");
printf("Init: %s", mira_GetErrorMsg(pmr));
if( pmr==NULL ) return;
```

## Error codes

- 101 Passing NULL pointer
- 102 mira\_GetDeviceName: Device index out of bounds
- 103 mira\_StartAcquisition: Project not loaded
- 104 mira\_StartAcquisition: Classifier model not loaded
  
- 110 mira\_LoadModel: Error loading model from file
- 111 mira\_LoadModel: Wrong file format
- 112 mira\_LoadModel: Internal error when loading
- 113 mira\_LoadModel: File cannot be opened
  
- 120 mira\_savelImage: Label image does not exist
  
- 130 mira\_LoadCorrection: Loading meta-data from the correction scan failed.
- 131 mira\_LoadCorrection: Error loading dark reference data
- 132 mira\_LoadCorrection: Dark and White reference images have different width or band count.
- 134 mira\_LoadCorrection: Both dark and white reference scans need to be loaded.
- 135 mira\_LoadCorrection: Reference file not present
- 136 mira\_LoadCorrection: Unsupported data layout or data type
  
- 140 Error switching to the computation device
- 141 mira\_RefreshDeviceList: Error setting CUDA backend
- 142 mira\_RefreshDeviceList: Error setting OpenCL backend
- 143 mira\_RefreshDeviceList: listNVIDIA and listOpenCL must be specified as 0 or 1 values
  
- 150 Feature does not exist
- 151 Wrong feature type requested
  
- 160 Max number of objects per frame reached
- 161 mira\_GetObjData\*: Object index out of bounds
- 162 mira\_GetObjData\*: Class index out of bounds (0..9)
- 163 mira\_GetObjDataClassSize: Segmentation not set to required 'All foreground' mode.
- 164 mira\_GetMaskType: Object segmetation not defined
  
- 170 mira\_StartAcquisition: Acquisition already running

- 171 `mira_StopAcquisition`: Acquisition not running
- 172 `mira_StartAcquisition`: Object segmentation cannot proceed
- 180 `mira_GetDecName`: Decision index out of bounds
- 190 `mira_SetForegroundClass`: Foreground class index out of bounds
- 201 `mira_ProcessCube`: Line-scan project type cannot process cubes
- 202 `mira_StartAcquisition`: Label image dimension mismatch
- 203 `mira_ProcessCube`: Missing image geometry description
- 204 `mira_ProcessFrame`: Line-scan processing requires BIL layout
- 205 `mira_GetInputDataLayout`: Undefined data layout
- 206 `mira_GetInputDataType`: Undefined data type
- 207 `mira_ProcessCube`: Unsupported project type
- 208 `mira_ProcessFrame`: Unsupported data type
- 208 `mira_ProcessCube`: Unsupported data type or data layout

## `mira_GetVersion`

Return runtime version

```
const char* mira_GetVersion( )
```

*Input:* None

*Output:* Version string

*Description:*

`mira_GetVersion` returns version string together with the release date. For example "2.1 26-mar-2020".

## `mira_GetErrorCode`

Return error code

```
int mira_GetErrorMsg(mrkernel* pmr)
```

*Input:* Runtime environment pointer

*Output:* Error code

*Description:*

`mira_GetErrorCode` returns error code. For a specific string description, [`mira\_GetErrorMsg`](#)

## `mira_GetErrorMsg`

Returns error message.

```
const char* mira_GetErrorMsg(mrkernel* pmr)
```

*Input:* Runtime environment pointer

*Output:* Error message string

*Description:*

`mira_GetErrorMsg` returns error message as string. For a specific error code, use [`mira\_GetErrorCode`](#)

## **mira\_RefreshDeviceList**

Fills the list of computational devices available

```
int miraacq_ScanDevices(makernel *pma)
```

*Input:*

- Runtime environment pointer pma

*Output:* Result: MIRA\_OK or error code

*Description:*

[miraacq\\_ScanDevices](#) searches for acquisition devices available. After calling [miraacq\\_ScanDevices](#), one can get device count using [miraacq\\_GetDeviceCount](#) and names with [miraacq\\_GetDeviceName](#).

*Example:*

```
MIRAACQ_CHECK( miraacq_ScanDevices(pma) );
const int devCount=miraacq_GetDeviceCount(pma);
printf( "\n%d devices:\n",devCount);
for(int i=0;i<devCount;i++) {
    printf( "%d : %s\n",i, miraacq_GetDeviceName(pma,i));
}
```

The [MIRAACQ\\_CHECK](#) macro checks the output result. If error occurs, the program flow is terminated. See [miraacq.h](#) definition.

## **mira\_GetDeviceCount**

Returns the number of GPU devices found

```
int mira_GetDeviceCount(mrkernel* pmr)
```

*Input:* Runtime environment pointer

*Output:* Number of devices found

*Description:*

[mira\\_GetDeviceCount](#) returns the number of devices found by [mira\\_RefreshDeviceList](#).

## **mira\_GetDeviceName**

Returns the name of a specific computational device

```
const char* mira_GetDeviceName(mrkernel* pmr,int deviceInd)
```

*Input:*

- Runtime environment pointer
- Device index

*Output:* String name of a device

*Description:*

[mira\\_GetDeviceName](#) returns the name for a specific device. Before using [mira\\_GetDeviceName](#) or [mira\\_GetDeviceCount](#), the device list needs to be constructed by [mira\\_RefreshDeviceList](#).

## **mira\_SetDevice**

Sets specific computational device

```
int mira_SetDevice(mrkernel* pmr,int deviceInd)
```



*Input:*

- Runtime environment pointer
- Device index

*Output:* Result code (MIRA\_OK or [error](#))

*Description:*

[mira\\_SetDevice](#) sets specific computational device. The device list needs to be constructed by [mira\\_RefreshDeviceList](#).

*Example:*

```
MIRA_CHECK( mira_RefreshDeviceList(pmr,1,0) );
const int devCount=mira_GetDeviceCount(pmr);
printf( "\n%d devices:\n",devCount );
for( int i=0;i<devCount;i++) {
    printf( "%d : %s\n",i, mira_GetDeviceName(pmr,i));
}
int deviceInd=atoi( argv[1] );
MIRA_CHECK( mira_SetDevice(pmr,deviceInd) );
printf( "Device selected: %d '%s'\n",deviceInd,mira_GetDeviceName(pmr,deviceInd));
```

The [MIRA\\_CHECK](#) macro checks the output result. If error occurs, the program flow is terminated. See [perclass\\_mira.h](#) definition.

**[mira\\_LoadModel](#)**

Loads classification model

```
int mira_LoadModel(mrkernel *pmr, const char* filename)
```

*Input:*

- Runtime environment pointer
- Filename (.mira project file)

*Output:* Result code (MIRA\_OK or [error](#))

*Description:*

[mira\\_LoadModel](#) loads a classification model from .mira project file.

**[mira\\_LoadCorrection](#)**

Loads white and dark correction data from disk

```
int mira_LoadCorrection(mrkernel *pmr, const char* dirname, const char*scanname);
```

*Input:*

- Runtime environment pointer
- Dirname - a name of a directory containing a scan directory
- Scanname - a name of a scan directory

*Output:* Result code (MIRA\_OK or [error](#))

*Description:***For Headwall project type:**

Correction information is assumed to be in whiteReference and darkReference scans. To load references, pass the path to a directory containing the whiteReference and darkReference ENVI scans. The third

argument is NULL.

Header files must have .hdr extensions. Both spectral cubes can have arbitrary extensions. Reference cubes must be in BIL data layout. Both uint16 and float data types are supported.

Example:

```
res = mira_LoadCorrection(pmr, "path_to_dir_with_references", NULL);
```

In this way, both reference files are loaded at the same time.

#### For Specim project type:

`mira_LoadCorrection` loads dark and white correction information from `dirname` directory. The assumption is the a `scanname` is a name of a directory inside the `dirname` directory and that it conforms Specim LUMO scanner directory structure. This means that inside `scanname` directory is a capture sub-directory. Inside the capture sub-dir, the following files are needed:

- WHITEREF\_scanname.hdr
- WHITEREF\_scanname.raw
- DARKREF\_scanname.hdr
- DARKREF\_scanname.raw
- scanname.hdr

The `scanname.hdr` defines wavelengths, band count and pixel count of a scan. Note, that the `scanname.raw` is not needed when loading correction.

All ENVI cubes are supposed to be in BIL data layout and use uint16 data type.

If `mira_LoadCorrection` is not called, the assumption is that the input data stream is already reflectance corrected and in float data type. This can be checked using `mira_GetInputDataType`.

### `mira_SetMinObjSize`

Set the minimum object size for segmentation

```
int mira_SetMinObjSize(mrkernel *pmr, int minSize);
```

*Input:*

- Runtime environment pointer
- `minSize` - minimum object size in pixels

*Output:* Result code (MIRA\_OK or [error](#))

*Description:*

`mira_SetMinObjSize` sets the minimum object size in pixels. Objects with size larger or equal than `minSize` are reported.

### `mira_SetSegmentation`

Set the minimum object size for segmentation

```
int mira_SetSegmentation(mrkernel *pmr, int enable);
```

*Input:*

- Runtime environment pointer
- `enable` - flag is to enable (1) or disable (0) object segmentation

*Output:* Result code (MIRA\_OK or [error](#))

*Description:*

`mira_SetSegmentation` enables or disables object segmentation. For all type of projects it is disabled

by default (Note: before 2.3, it was enabled by default for line-scan projects).. Use before starting the acquisition. Note, that the model needs to have some class or classes flagged as foreground to perform segmentation.

### **mira\_GetInputWidth**

Get the expected width of the input image stream

```
int mira_GetInputWidth(mrkernel* pmr);
```

*Input:*

- Runtime environment pointer

*Output:* Input image width if > 0 or an error code if < 0

The image width in the line scan use case is the number of pixels of one line i.e. the pixels across the belt.

### **mira\_SetInputWidth**

Set the width of the input image stream

```
int mira_SetInputWidth(mrkernel* pmr, int width);
```

*Input:*

- Runtime environment pointer
- Input width of the data stream

*Output:* Result code (MIRA\_OK or [error](#))

Input width of the data stream may be set manually. The width overrules the setting in the loaded project.

### **mira\_GetInputHeight**

Get the expected height of the input image stream

```
int mira_GetInputHeight(mrkernel* pmr);
```

*Input:*

- Runtime environment pointer

*Output:* Input image height if > 0 or an error code if < 0

This call is only meaningful in snapshot use-case where entire spectral cube is to be processed with `mira_ProcessCube` function.

### **mira\_GetInputBands**

Get the expected number of spectral bands of the input image stream

```
int mira_GetInputBands(mrkernel* pmr);
```

*Input:*

- Runtime environment pointer

*Output:* Input band count if > 0 or an error code if < 0

This call returns the number of spectral bands expected in each pixel.

## mira\_GetInputDataType

Get the expected data type in the input image stream

```
int mira_GetInputDataType(mrkernel* pmr);
```

*Input:*

- Runtime environment pointer

*Output:* Input data type if  $\geq 0$  or an error code if  $< 0$

```
MIRA_DATATYPE_UNKNOWN 0
MIRA_DATATYPE_UINT16  1
MIRA_DATATYPE_FLOAT   2
MIRA_DATATYPE_UINT8   3
```

This call returns the data type expected in the input image stream based on the loaded model.

## mira\_GetInputDataLayout

Get the expected data layout of the input image stream

```
int mira_GetInputDataLayout(mrkernel* pmr);
```

*Input:*

- Runtime environment pointer

*Output:* Input data layout if  $\geq 0$  or an error code if  $< 0$

```
MIRA_DATA_LAYOUT_UNKNOWN 0
MIRA_DATA_LAYOUT_BIP      1 /* spectrum-by-spectrum (dimensions: bands-width-height) */
MIRA_DATA_LAYOUT_BIL      2 /* frame-by-frame (dimensions: width-bands-height) */
MIRA_DATA_LAYOUT_BSQR     3 /* spatial frame by frame (dimensions: width-height-bands) */
```

This call returns the data layout expected in the input image stream based on the loaded model.

## mira\_GetMaskType

Get the mask type of the loaded object segmentation model

```
int mira_GetMaskType(mrkernel* pmr);
```

*Input:*

- Runtime environment pointer

*Output:* Input data layout if  $\geq 0$  or an error code if  $< 0$

```
MIRA_MASK_EACH_FOREGROUND 1
MIRA_MASK_ALL_FOREGROUND  2
```

This call returns the mask type for the loaded model. The 'each-foreground' type is used for single material per object situations (object detection). The 'all-foreground' mask is used for complex objects composed of multiple materials where the union of classes defines object mask (object classification). An example: A potato can have healthy flesh or rotten defect - these are the trained classes. The segmentation mask is set of 'all-foreground' and, therefore, entire piece of potato is segmented out. For each object, perClass Mira Runtime provides pixel count for each foreground class. This allows object sorting based on composition.

## **mira\_StartAcquisition**

Starts the acquisition

```
int mira_StartAcquisition(mrkernel* pmr)
```

*Input:*

- Runtime environment pointer

*Output:* Result code (MIRA\_OK or [error](#))

*Description:*

[mira\\_StartAcquisition](#) starts the acquisition process. Computational device must be defined, model loaded and correction information defined.

Individual frames can then be processed using [mira\\_ProcessFrame](#)

## **mira\_ProcessFrame**

Process a single individual raw spectral frame from a line-scan camera

```
int mira_ProcessFrame(mrkernel* pmr, void* pData);
```

*Input:*

- Runtime environment pointer
- Pointer to external buffer with raw spectral frame data

*Output:* Result code (MIRA\_OK or [error](#))

*Description:*

[mira\\_ProcessFrame](#) passes data of a raw spectral frame from a line-scan camera. Acquisition process need to be running (started using [mira\\_StartAcquisition](#)).

The input data stream from a line scan camera is expected to be in BIL layout (pixels on the spatial line times spectral bands). The expected geometry and data type are defined by the loaded solution. This information can be queried by the [mira\\_GetInputWidth](#), [mira\\_GetInputBands](#), and [mira\\_GetInputDataType](#).

After a frame is processed, per-pixel decisions may be read out using [mira\\_GetFrameDecisions](#) or object information extracted using [mira\\_GetObj\\*](#) functions.

## **mira\_ProcessCube**

Process a single spectral cube

```
int mira_ProcessCube(mrkernel* pmr, void* pData);
```

*Input:*

- Runtime environment pointer
- Pointer to external buffer with raw spectral frame data

*Output:* Result code (MIRA\_OK or [error](#))

*Description:*

[mira\\_ProcessCube](#) passes data of a entire spectral cube. Acquisition process need to be running (started using [mira\\_StartAcquisition](#)).

The expected geometry and data type are defined by the loaded solution. This information can be queried by the [mira\\_GetInputHeight](#), [mira\\_GetInputWidth](#), [mira\\_GetInputBands](#), [mira\\_GetInputDataLayout](#) and [mira\\_GetInputDataType](#).

After a cube is processed, per-pixel decisions may be read out using [mira\\_GetFrameDecisions](#) or object information extracted using [mira\\_GetObj\\*](#) functions.

## **mira\_StopAcquisition**

Starts the acquisition

```
int mira_StopAcquisition(mrkernel* pmr)
```

*Input:*

- Runtime environment pointer

*Output:* Result code (MIRA\_OK or [error](#))

*Description:*

[mira\\_StopAcquisition](#) stops the acquisition process. Statistics on number of processed frames, speed per frame and frame-rate is available via a subsequent [mira\\_GetErrorMsg](#) call.

*Example:*

Average alg time: 2.79676 ms/frame (357.557 fps), processed 1500 frames, first 500 skipped for warm-up.

## **mira\_GetFrameDecisions**

Returns a pointer to pixel decisions on the last processed line

```
const unsigned char *mira_GetFrameDecisions(mrkernel* pmr);
```

*Input:*

- Runtime environment pointer

*Output:* Pointer to decisions on the last processed line

*Description:*

[mira\\_GetFrameDecisions](#) returns pointer to decisions at the last processed line. The values are zero-based indices. Class name corresponding to each index can be obtained using [mira\\_GetDecName](#)

## **mira\_GetDecCount**

Returns the number of decisions provided by the classifier

```
int mira_GetDecCount(mrkernel* pmr);
```

*Input:*

- Runtime environment pointer

*Output:* Number of decisions provided by the classifier

*Description:*

[mira\\_GetDecCount](#) returns the number of decisions provided by the classifier. The decision index, returned for each pixel by [mira\\_GetFrameDecisions](#) is a value smaller than decision count (zero-based indexing).

## **mira\_GetRegVarCount**

Returns the number of regression variables available by the regression model

```
int mira_GetRegVarCount(mrkernel* pmr);
```

*Input:*

- Runtime environment pointer

*Output:* Number of regression variables provided by the project

*Description:*

`mira_GetRegVarCount` returns the number of regression variables provided by the project. The `mira_GetRegVarName` function can be then used to obtain specific variable names.

If the project does not contain any trained regression model, zero is returned. Therefore, this function can be used to check whether regression modeling is enabled in the project.

**mira\_GetRegVarName**

Returns regression variable name given its index

```
const char* mira_GetRegVarName(mrkernel* pmr, int regVarInd);
```

*Input:*

- Runtime environment pointer
- Regression variable index (0 to the count returned by `mira_GetRegVarCount` - 1)

*Output:* String name of a regression variable

*Description:*

`mira_GetRegVarName` returns the name for a specific regression variable.

*Example:*

```
varCount=mira_GetRegVarCount(pmr);
if( varCount>0 ) {
    /* regression variables */
    printf("\nregression vars:\n");
    for(int i=0;i<varCount;i++) {
        printf("%d : %s\n",i,mira_GetRegVarName(pmr,i));
    }
}
```

*Output:*

```
regression vars:
0 : brix
1 : acidity
```

**mira\_GetObjDataRegOutput**

Read information on segmented out objects

```
int mira_GetObjDataRegOutput(mrkernel* pmr, int entryInd, const float**
ppObjData);
```

*Input:*

- Runtime environment pointer
- entryID - zero-based index of an object
- ppObjData - pointer to a pointer to a table with regression results per object (floating point\_

*Output:* Result (MIRA\_OK or [error](#))

*Description:*

`mira_GetObjDataRegOutput` provides per-object regression output. The second parameter `entryInd` is a zero-based object index (0..number of objects found -1). The third parameter `ppObjData` represents regression values for a given object. The example below illustrates that we declare a pointer to float called `pObjData` and initialize its value to NULL. In an acquisition loop, after processing a frame, if an object is found, we call `mira_GetObjDataInt` in a for loop extracting object information.

When we wish to access regression information for a given object, we use the `mira_GetObjDataRegOutput` function. We pass **the address** of the `pObjData` to the



`mira_GetObjDataInt`, not the pointer itself. The actual regression value can be accessed using `pRegData[v]`, where `v` is the zero-based regression variable index. No memory allocation is needed on the side of user code.

*Example:*

```
int objCount=0;

const int varCount=mira_GetRegVarCount(pmr);

float* pRegData=NULL;

while( frameInd<frames ) {
    res=mira_ProcessFrame(pmr,ptrf);

    objCount=mira_GetObjCount(pmr);

    if( objCount>0 ) {
        for(int i=0;i<objCount;i++) {
            /* we pass address of a pointer to receive object table
               allocated by the runtime */

            mira_GetObjDataInt(pmr,i,&pObjData);

            /* pObjData allows us to access object details */
            printf("\n obj%d : %d,d ",
                pObjData[MIRA_OBJECT_ID],
                pObjData[MIRA_OBJECT_FRAME], /* along the belt */
                pObjData[MIRA_OBJECT_POS] ); /* across the belt */

            if( varCount>0 && mira_GetObjDataRegOutput(pmr,i,&pRegData)
==MIRA_OK ) {

                printf("\t reg:");
                for(int v=0;v<varCount;v++) {
                    printf(" %3.3f ",pRegData[v]);
                }

            } /* end of for loop */
        } /* end of if objects */
    } /* end of frame acquisition */
}
```

## `mira_GetFrameRegOutputVar`

Returns a pointer to pixel decisions on the last processed line

```
const float *mira_GetFrameRegOutputVar(mrkernel* pmr, int varInd, int
maskBackground, float maskVal);
```

*Input:*

- Runtime environment pointer
- regression variable index
- flag specifying if background should be masked
- masking value put on background pixels (if `maskBackground==1`)

*Output:* Pointer to per-pixel floating point regression value for the frame

*Description:*

`mira_GetFrameRegOutputVar` returns pointer to floating point regression values at the last processed

line. The pointer can be dereferenced for each pixel of the processed line (from 0 to InputDataWidth-1 inclusive). The output is provided for a regression variable defined by the index given in the second parameter). The third parameter specifies if background pixels should be masked (maskBackground==1) or not (maskBackground==0). If masking is requested, the last parameter specifies floating point value copied into all background pixels. The masking procedure simplifies post-processing of the per-pixel regression output.

## **mira\_GetDecName**

Returns decision (class) name given decision index

```
const char* mira_GetDecName(mrkernel* pmr, int decInd);
```

*Input:*

- Runtime environment pointer
- Decision index (0 to number of decisions - 1)

*Output:* String name of a class (decision)

*Description:*

`mira_GetDecName` returns the name for a specific decision index.

*Example:*

```
printf("classifier decisions:\n");
const int decCount=mira_GetDecCount(pmr);
for(int i=0;i<decCount;i++) {
    printf("%d : %s\n",i,mira_GetDecName(pmr,i));
}
```

*Output:*

```
classifier decisions:
0 : background
1 : product
2 : foreign object
```

## **mira\_GetDecColor**

Returns R,G,B color of a given decision

```
const char* mira_GetDecColor(mrkernel* pmr, int decInd, unsigned char*
R, unsigned char* G, unsigned char* B);
```

*Input:*

- Runtime environment pointer
- Decision index (0 to number of decisions - 1)
- pointer to red, green and blue color

*Output:* String name of a class (decision)

*Description:*

`mira_GetDecColor` returns R,G, and B colors for a given decision

## **mira\_GetObjCount**

Returns the number of objects found after processing a frame

```
int mira_GetObjCount(mrkernel* pmr);
```

*Input:*

- Runtime environment pointer

*Output:* Number of objects found after processing a given frame

*Description:*

`mira_GetObjCount` returns the number of objects found after processing a given frame. If non-zero, the object information can be read using `mira_GetObjData*` functions, [see this example](#).

Note, that the object-specific information is replaced after next frame processing.

## `mira_GetObjDataInt`

Read information on segmented out objects

```
int mira_GetObjDataInt(mrkernel* pmr, int entryInd, int** ppObjData);
```

*Input:*

- Runtime environment pointer
- entryID - zero-based index of an object
- ppObjData - pointer to a pointer to a table with object information

*Output:* Result (MIRA\_OK or [error](#))

*Description:*

`mira_GetObjDataInt` returns details on a specific object found. The first parameter is a zero-based object index (0..number of objects found -1). The second parameter represents an object table. The example below illustrates that we declare a pointer to int called pObjData and initialize its value to NULL. In an acquisition loop, after processing a frame, if an object is found, we call `mira_GetObjDataInt` in a for loop extracting object information. Note, that we pass address of the pObjData to the `mira_GetObjDataInt`.

*The object table:*

MIRA_OBJECT_ID	0	Unique object identifier
MIRA_OBJECT_FRAME	1	Frame index for the object centroid
MIRA_OBJECT_POS	2	Position of the object centroid across the belt
MIRA_OBJECT_MINFRAME	3	Bounding box coordinates:
MIRA_OBJECT_MAXFRAME	4	
MIRA_OBJECT_MINCOL	5	
MIRA_OBJECT_MAXCOL	6	
MIRA_OBJECT_SIZE	7	Object size in pixels
MIRA_OBJECT_CLASS	8	Object class index

*Example:*

```
int objCount=0;
int* pObjData=NULL; /* pointer to object table data */
while( frameInd<frameCount ) {

    mira_ProcessFrame(pmr, pFrame);

    objCount=mira_GetObjCount(pmr);
    if( objCount>0 ) {
        for( int i=0; i<objCount; i++) {
            /* we pass address of a pointer to receive object table
            allocated by the runtime */

            mira_GetObjDataInt(pmr, i, &pObjData);

            /* pObjData allows us to access object details */
            printf("\n obj%d : %d,d ",
                pObjData[MIRA_OBJECT_ID],
                pObjData[MIRA_OBJECT_FRAME], /* along the belt */
```

```

        pObjData[MIRA_OBJECT_POS] ); /* across the belt */
    } /* end of for loop */
} /* end of if objects */
} /* end of frame acquisition */

```

### **mira\_GetObjDataClassSize**

For complex object segmentation, returns number of class pixels within the object

```
int mira_GetObjDataClassSize(mrkernel* pmr, int entryInd, int classInd);
```

*Input:*

- Runtime environment pointer
- entryInd - Object index (zero-based)
- classInd - Class index (zero-based)

*Output:* Number of pixels of specific class within specific object or error

*Description:*

[mira\\_GetObjDataClassSize](#) returns the number of pixels of specific class within an object. This function is applicable when object segmentation mask is defined as "all foreground" i.e. when complex objects are segmented.

### **mira\_GetObjDataClassFrac**

For complex object segmentation, returns the fraction of class pixels within the object

```
float mira_GetObjDataClassFrac(mrkernel* pmr, int entryInd, int classInd)
```

*Input:*

- Runtime environment pointer
- entryInd - Object index (zero-based)
- classInd - Class index (zero-based)

*Output:* Fraction (0.0 to 1.0) inclusive of pixels of specific class within specific object or error code

*Description:*

[mira\\_GetObjDataClassFrac](#) returns the fraction of specific class within an object. This function is applicable when object segmentation mask is defined as "all foreground" i.e. when complex objects are segmented.

If error occurs the negative error code value is returned.

### **mira\_SaveImage**

Save internal segmentation buffer as PNG image

```
int mira_SaveImage(mrkernel* pmr, const char* filename)
```

*Input:*

- Runtime environment pointer
- Filename

*Output:* Result (MIRA\_OK or error)

*Description:*

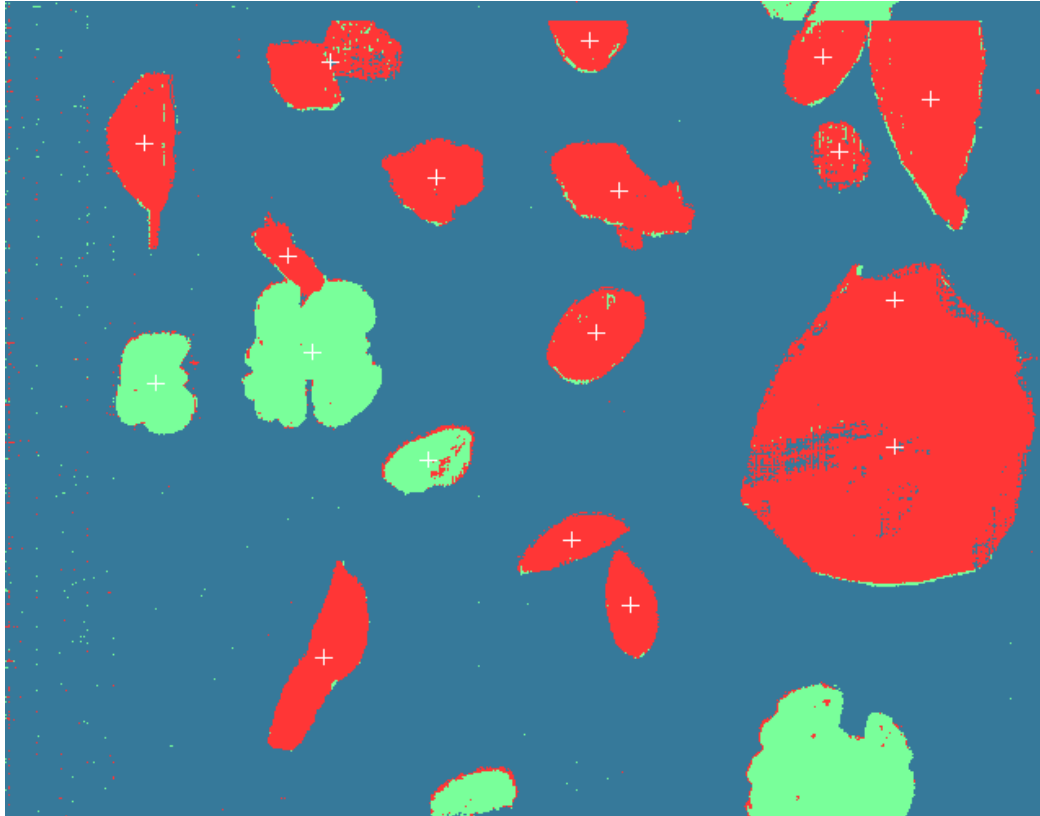
[mira\\_SaveImage](#) saves internal segmentation buffer into a PNG file. It is intended as a quick visualization of what the classifier can "see" in a deployed system.

Segmented objects are highlighted by white crosses.

*Example:*

```
MIRA_CHECK( mira_StopAcquisition(pmr) );
printf( "\n %s", mira_GetErrorMsg(pmr) );
/* We can save the content of the internal buffer. */
mira_SaveImage(pmr, "out.png");
mira_Release(pmr);
```

*Output:* Content of out.png file



## mira\_Release

Release runtime internal session and clean resources.

```
void mira_Release(mrkernel* pmr)
```

*Input:*

- Runtime environment pointer

*Output:* None

*Description:*

`mira_Release` ends the session and releases all memory allocated by the runtime.

## Troubleshooting

If you experience unexpected behaviour or crashes, please contact [support@perclass.com](mailto:support@perclass.com)

Please provide:

- Detail on your license (dongle number or activation key)
  - Dongle:
    - USB dongle is enabled using license file stored in C:\Users\USERNAME\AppData\Roaming\perClassBV directory. The license file filename

- contains dongle number Dxxxx
  - Activation key:
    - Activation key is listed in the mira.ini file in C:\Users\USERNAME\AppData\Roaming\perClassBV
- Specific version of perClass Mira you're using
  - Please make sure you're running the latest available version of the software
  - You can check availability of software updates using *Help / Check for updates* command
- Description of behaviour leading to a crash. This helps us to repeat and fix the problem.
- If the behaviour leading to an issue cannot be repeated, please enable logging, use the software until the crash occurs and send us the mira.log file to support@perclass.com

## How to enable logging

Logging stores information on software internal process in a mira.log text file. This may help us to understand and fix issues.

In order to enable logging:

- Close any running instance of perClass Mira
- Open mira.ini file located in C:\Users\USERNAME\AppData\Roaming\perClassBV
- Enable logging by settig  
logMessagesToFile=true
- Start a new perClass Mira instance. The Output window will list in yellow, that logging is enabled

The log is stored in mira.log file in the same directory as above.

**TIP:** You may quickly open the license directory by *Help / Open license directory* command. However, please note that when any instance of perClass Mira closes, it saves its settings back to the mira.ini file overwriting its content. Therefore, you may loose settings edited externally. That is why we advice to close all instances before editing the file.

**TIP:** Disable logging when not needed. Keeping it enabled will incur a performance penalty