perClass Mira 5.0 Documentation

Table of contents

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

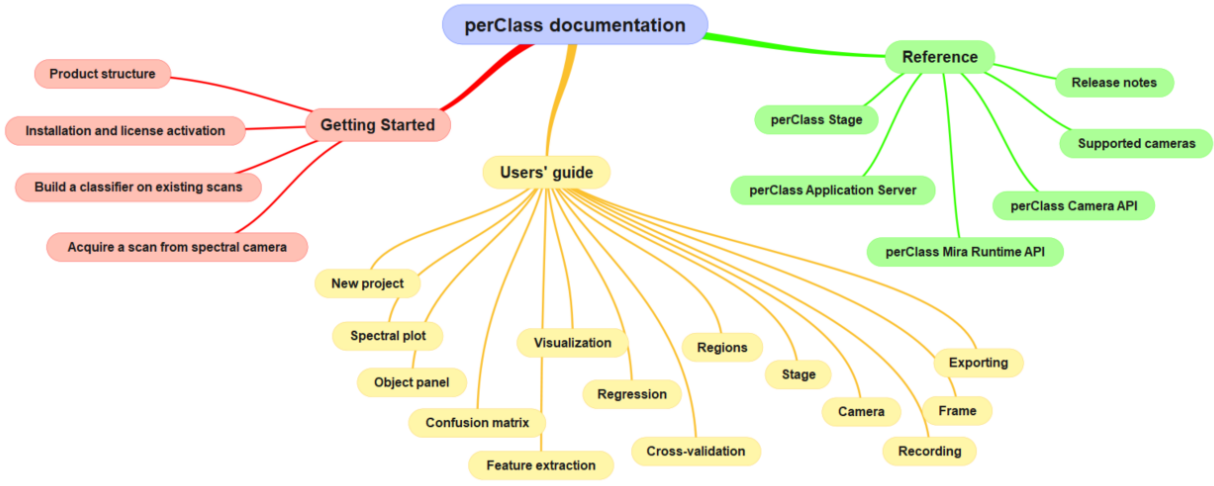
- Acquire high quality spectral images using a range of multi- and hyper-spectral cameras
- Build classification and regression solutions
- Export specific information from detected objects or user-defined regions
- Batch process new scans using user-defined solutions
- Deploy solutions to a live data 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 suporting 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 release notes**
  - Software integration
  - Supported sensors
  - perClass Stage

- perClass Mira Runtime API
- perClass Camera API
- perClass Application Server API

Getting started

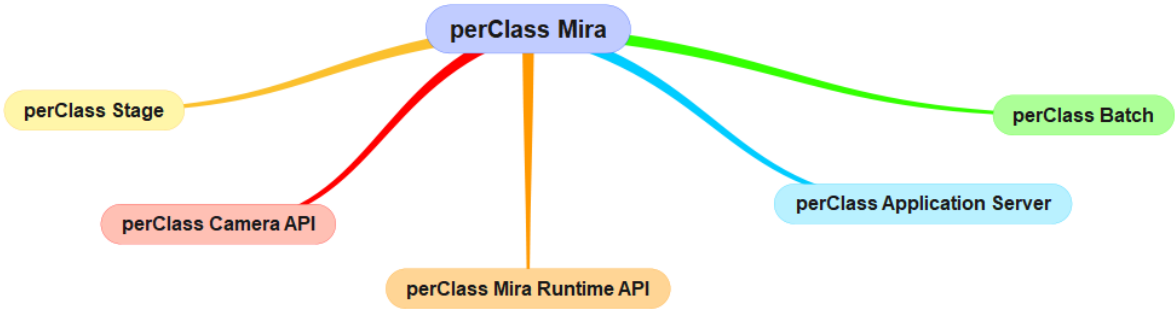
This document describes perClass product structure and provides quick guides to:

- 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

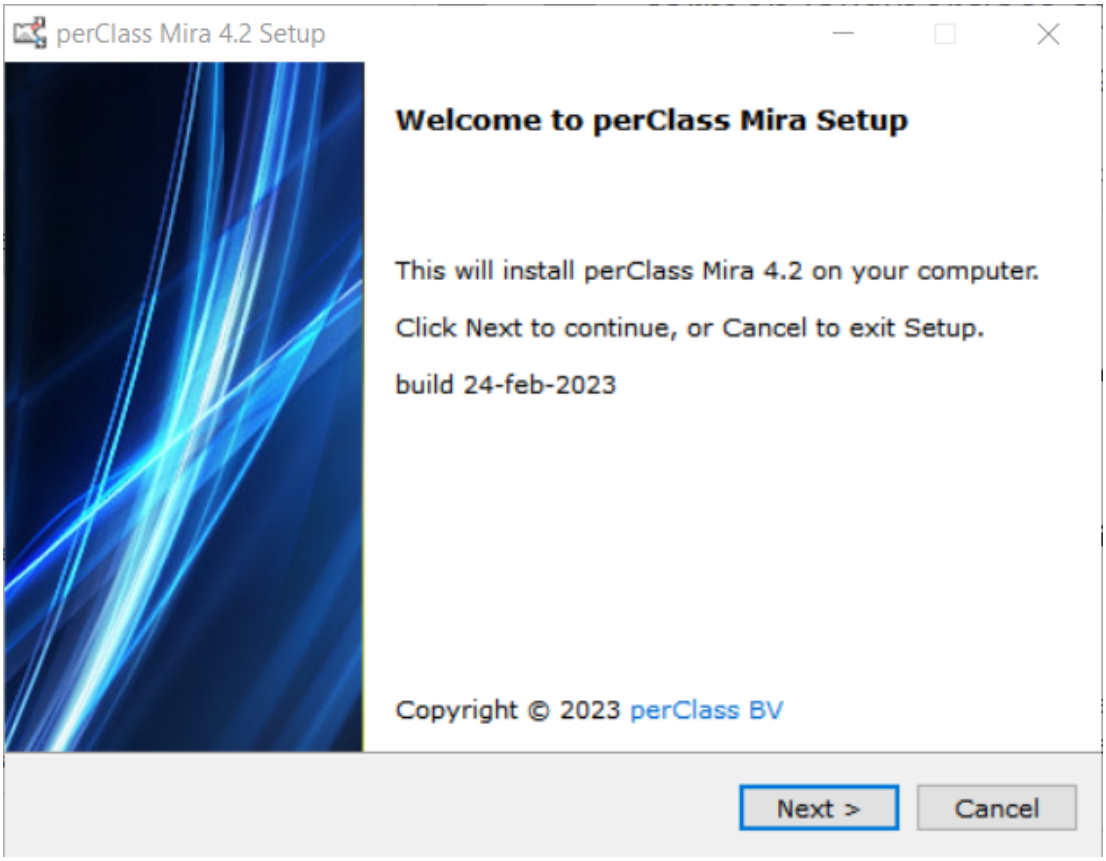
- perClass Mira User interface emporing users interpret spectral images
- perClass Mira Stage, a hardware lab-scanning kit supporting different spectral cameras.
- perClass Camera API connecting to different spectral cameras
- perClass Mira Runtime API to embed solutions developed in the perClass Mira 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



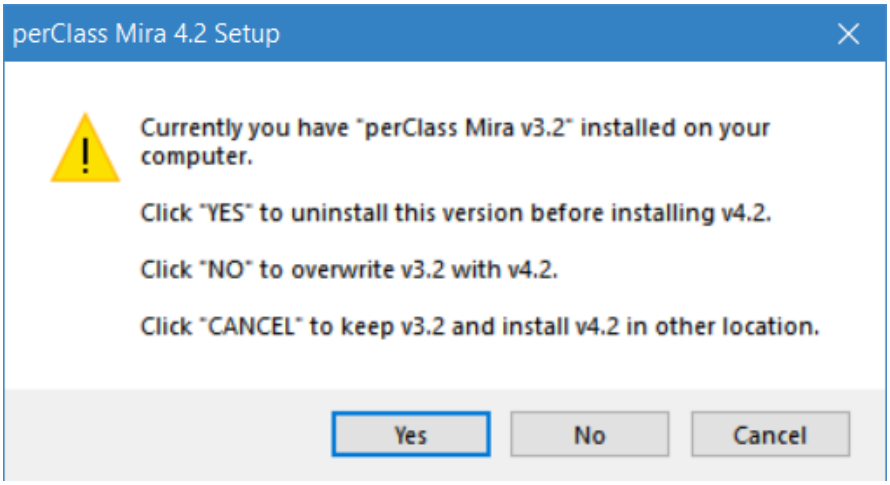
Installation and license activation

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

Run the installation file, the installer dialog will appear:

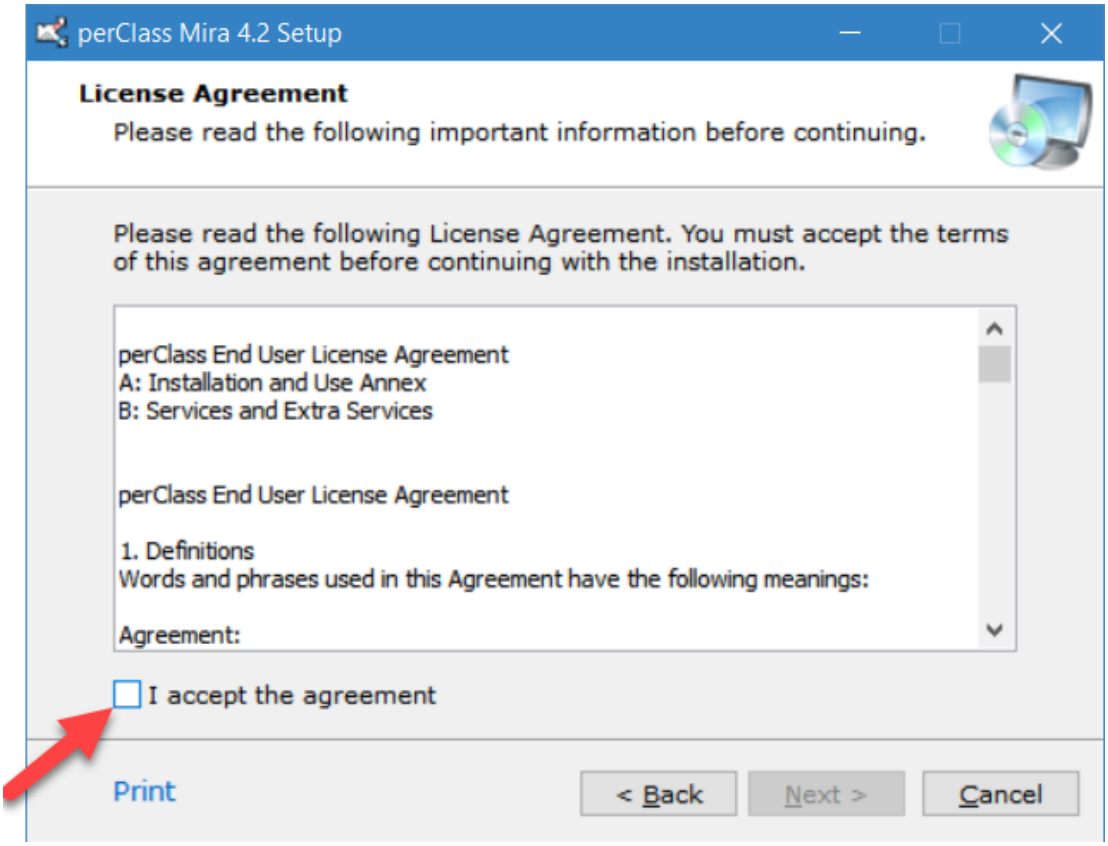


In case you already have an existing perClass Mira installation, the following dialog will also appear:

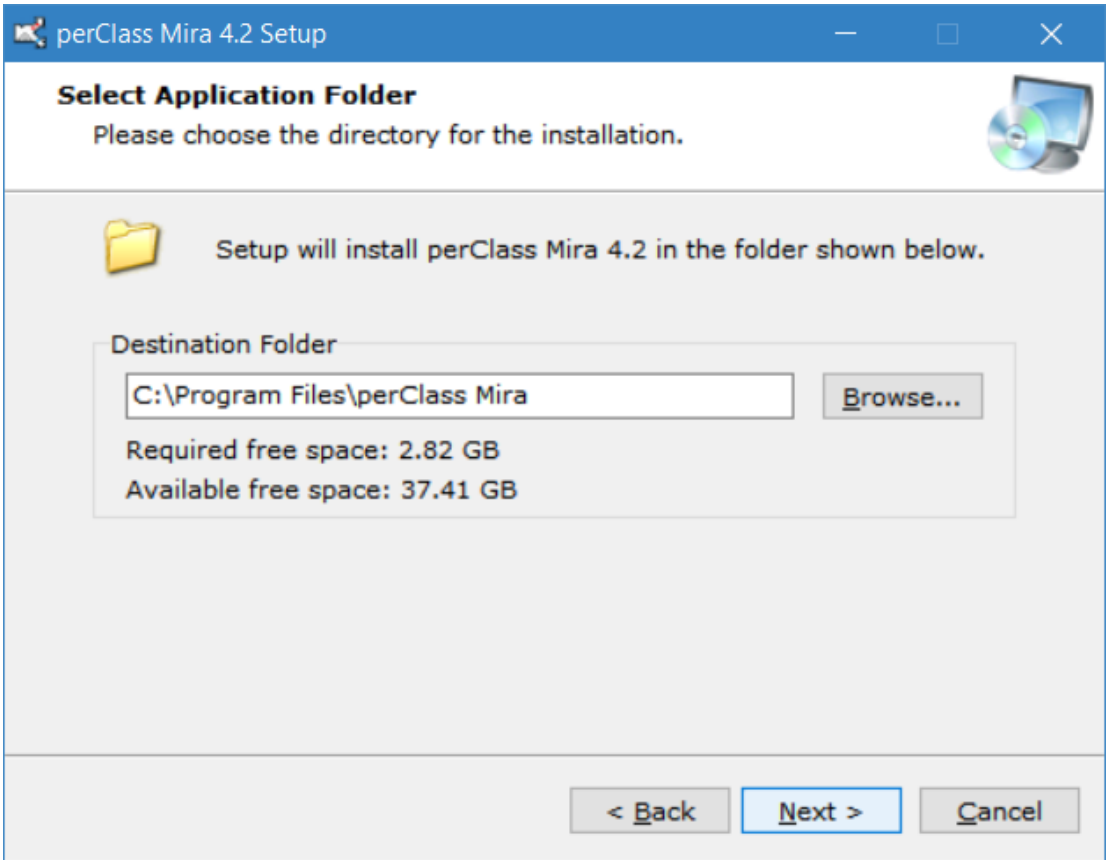


If you wish to install the new release into a directory you specify, click on *Cancel* button.

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

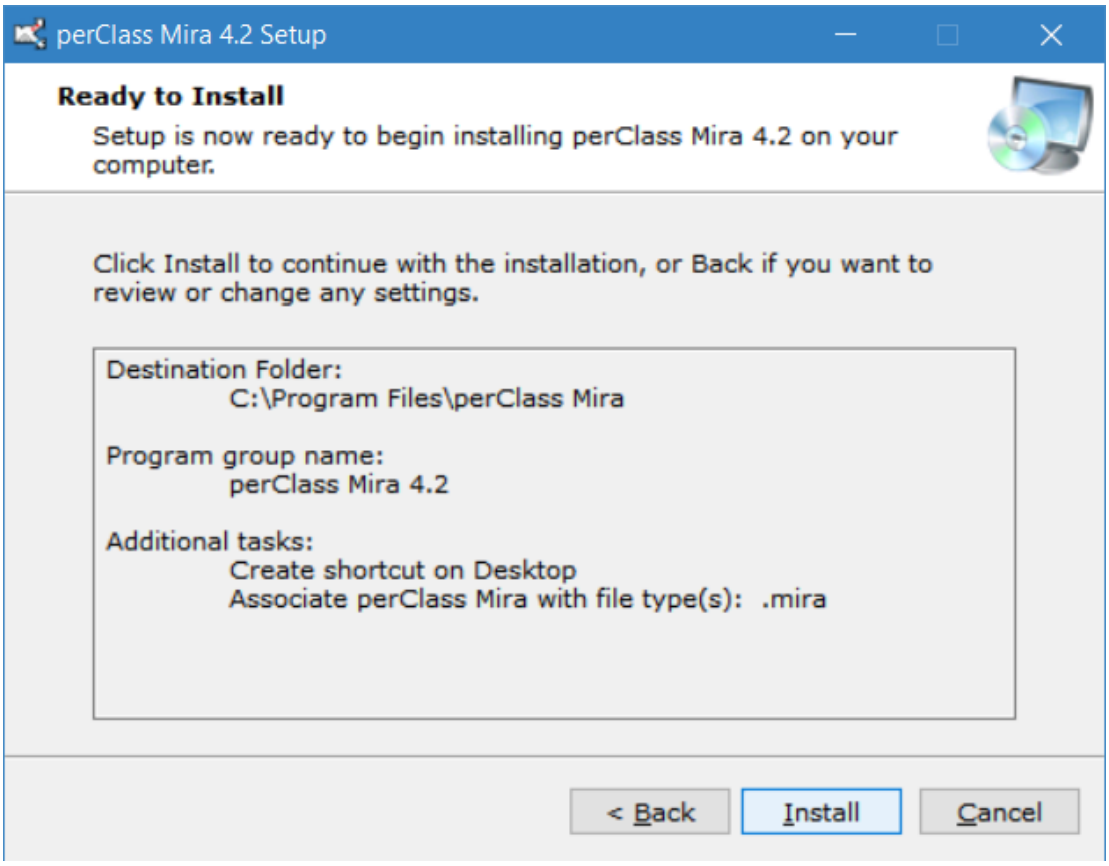


If you selected to customize location, you may now specify the installation directory:

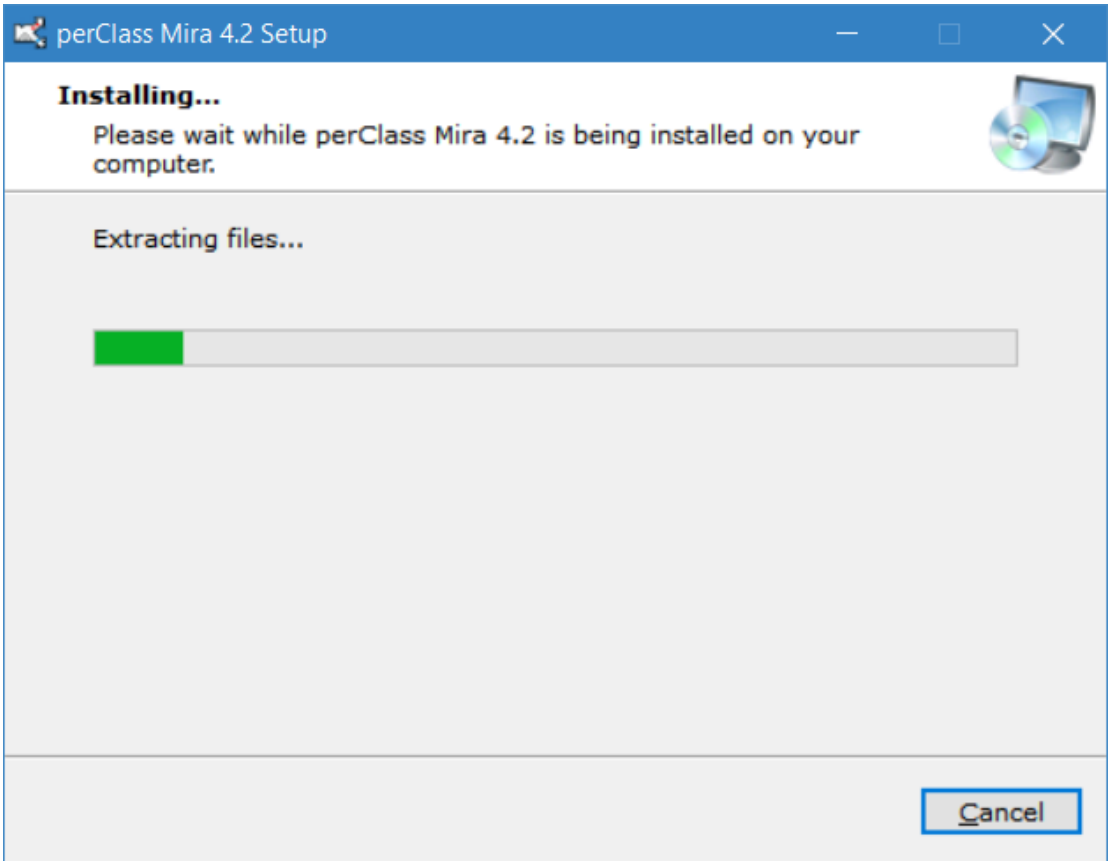


TIP: You may freely install multiple releases of perClass Mira on the same computer. If you, it is recommeded to use full version including date in the file name, for example: "perClass_Mira_4.2_24feb23"

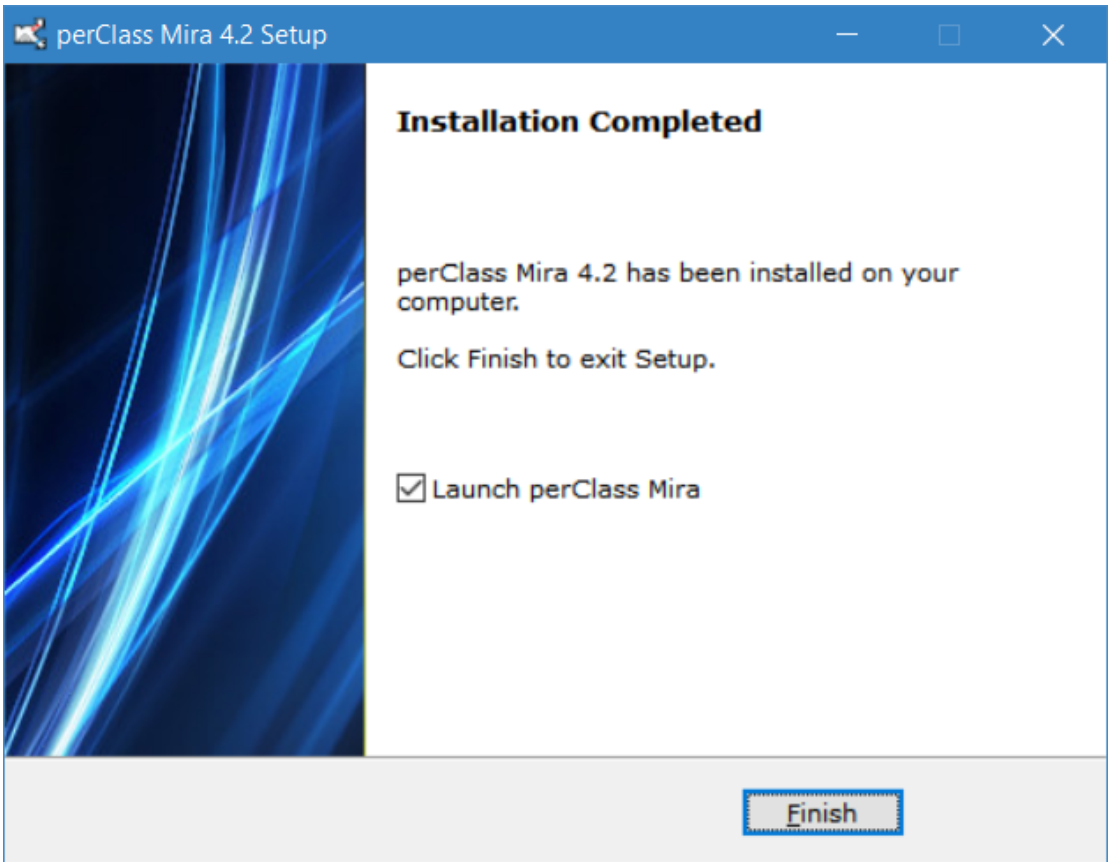
In the last step, you may confirm or cancel the installation:



After extracting the files:

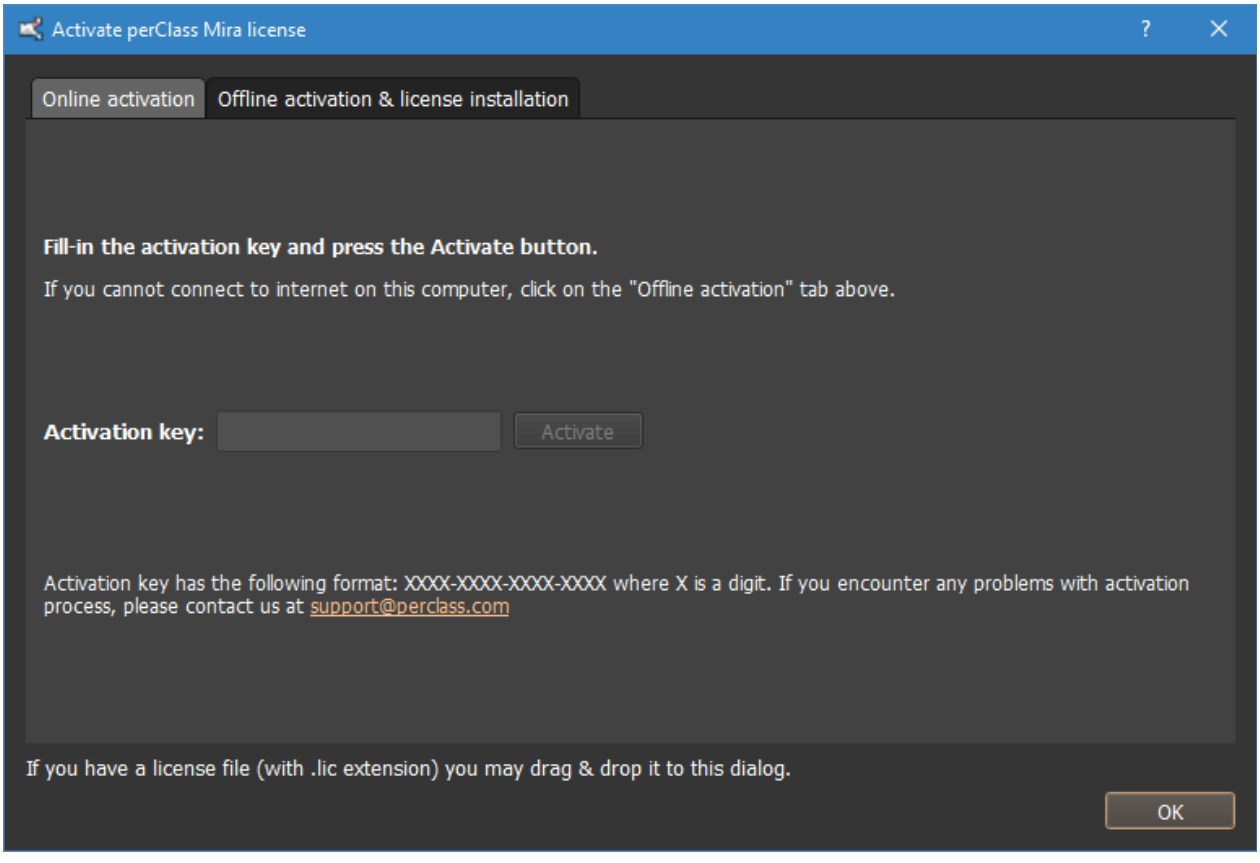


Finally, you may launch the installed software directory from the installation dialog:

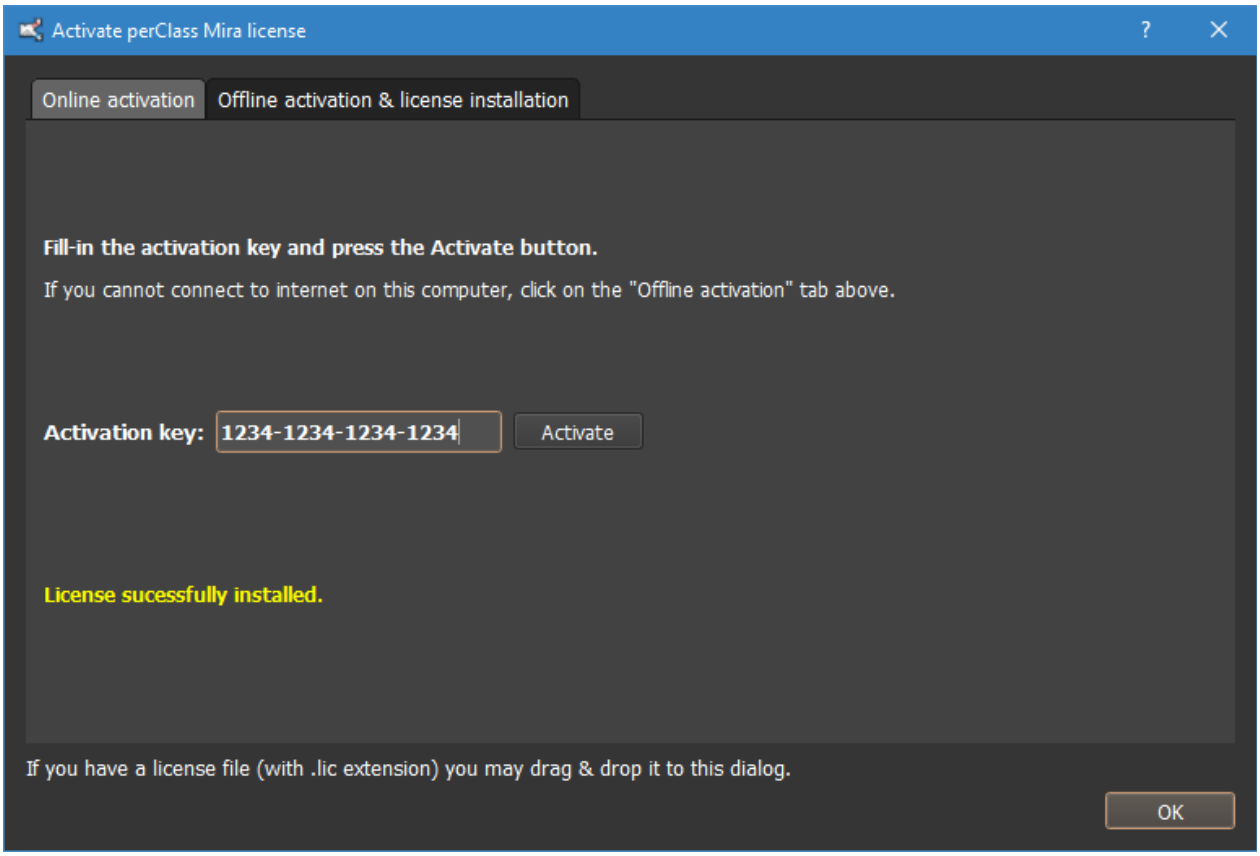


Activation

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



Fill in the activation key and press *Activate*



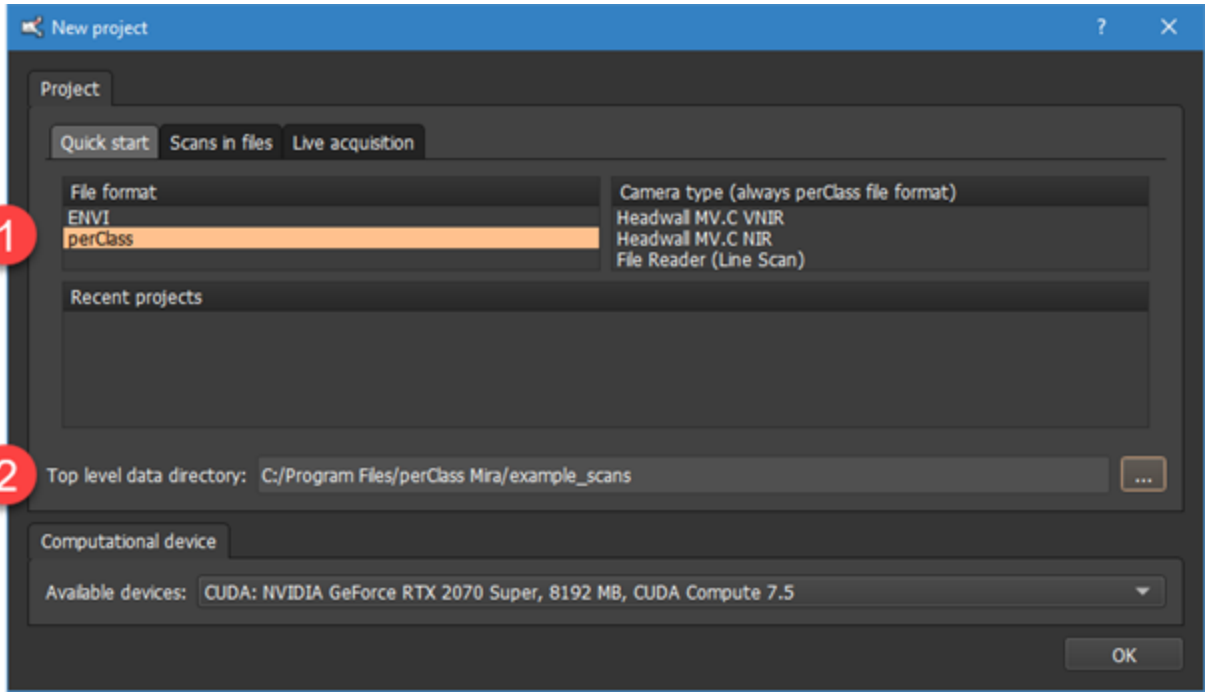
Build classifier on existing scans

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.

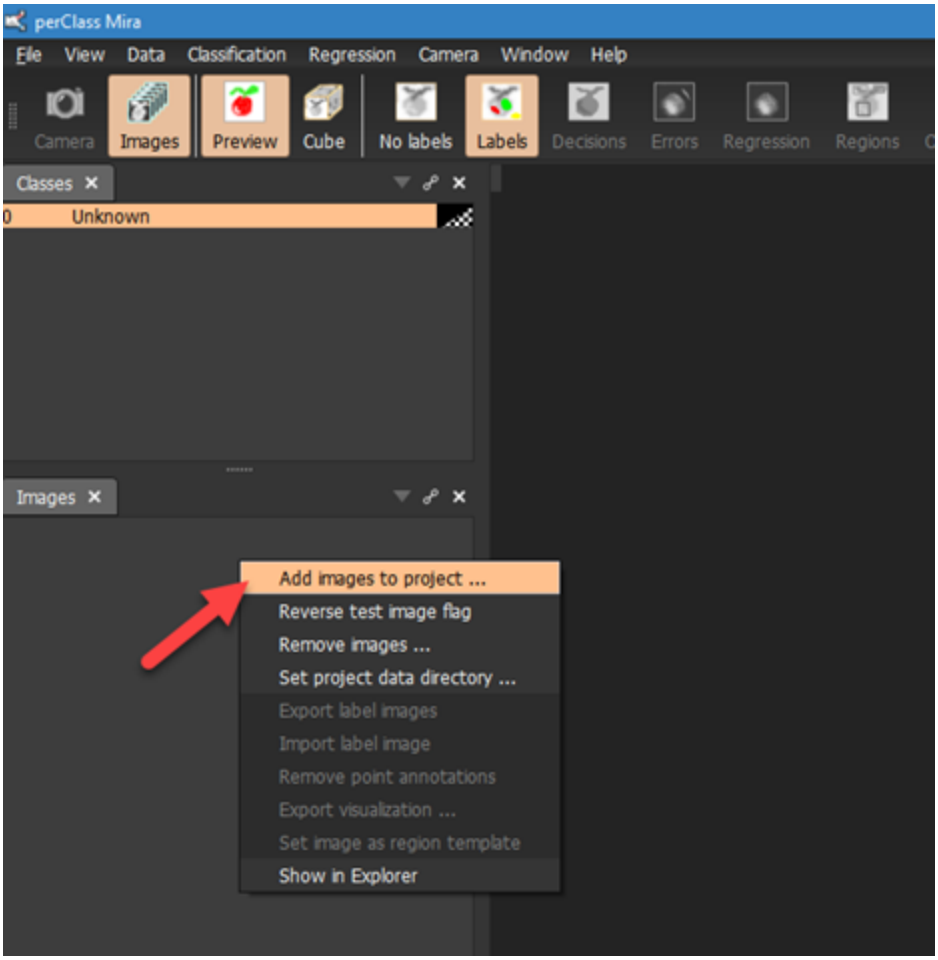
In this tutorial, we load an existing scan. Therefore, select the *perClass* project type 1 .

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.` 2

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

Adding images

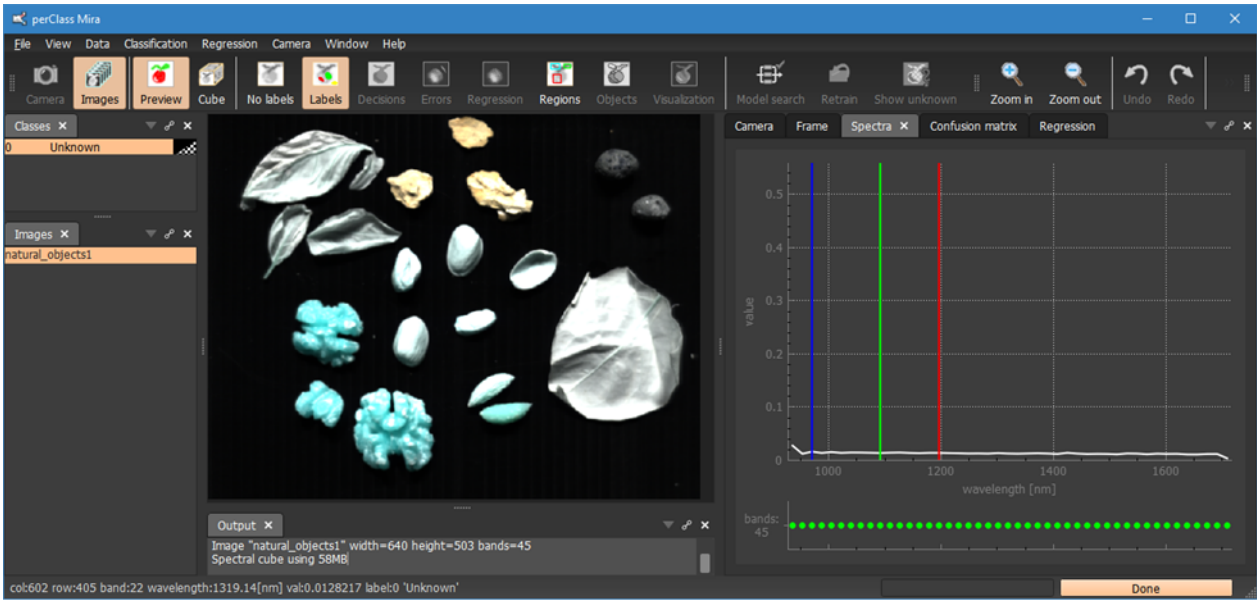
perClass Mira main window will open. In order to add images, right-click with your mouse in the *Images* list and select *Add images to project...* from the context menu.



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.



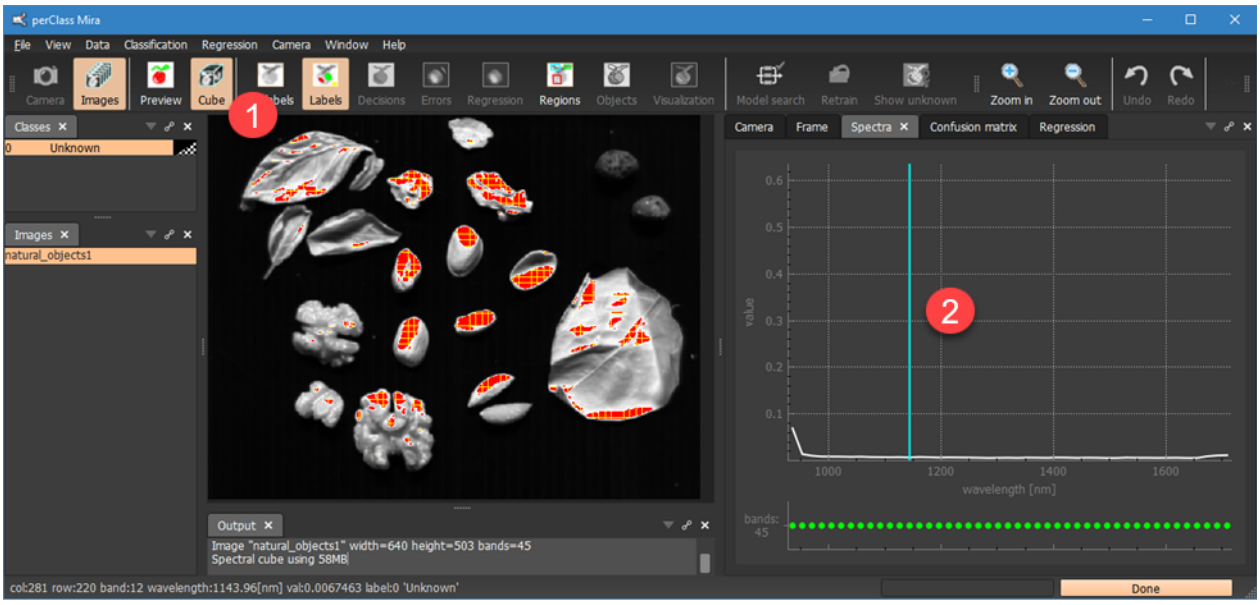
Spectral cube visualization

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 corresond 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 1 on the toolbar.

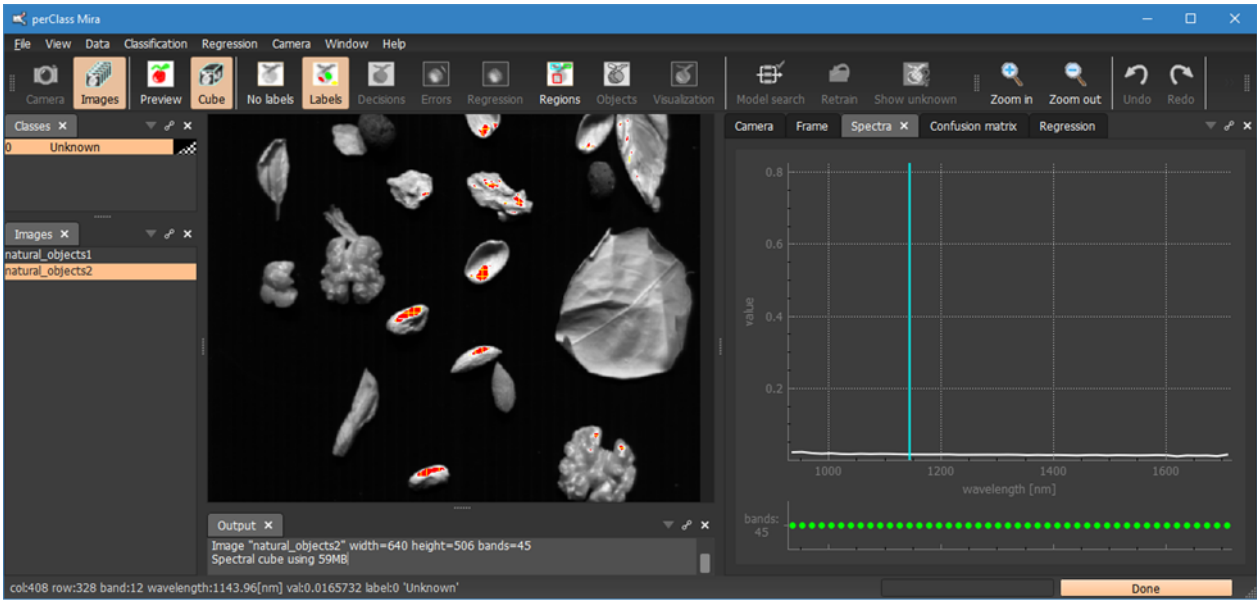


Dragging the blue line 2 in the spectral plot, you may change the band. Note the band index and wavelength in the status bar.

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.

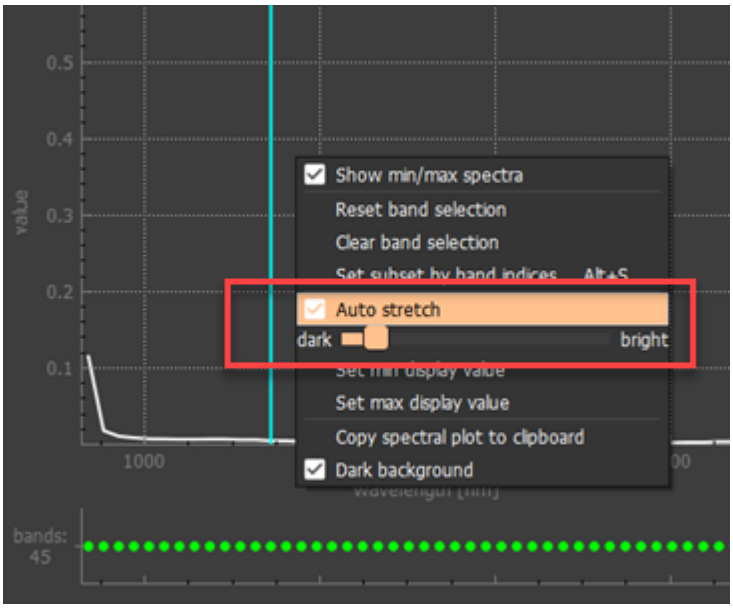
You may adjust the spectral plot range using the mouse wheel or by clicking and dragging in the spectral plot.

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 spectal plot context menu (via right mouse click).



The auto-stretch extends brightness of 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 alwas see "something" irrespective 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

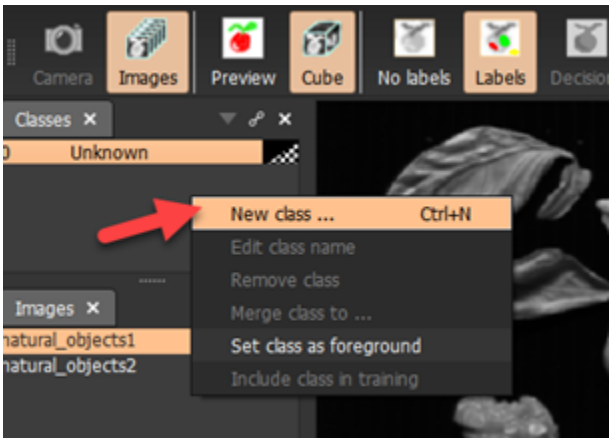
Training a clasifier

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

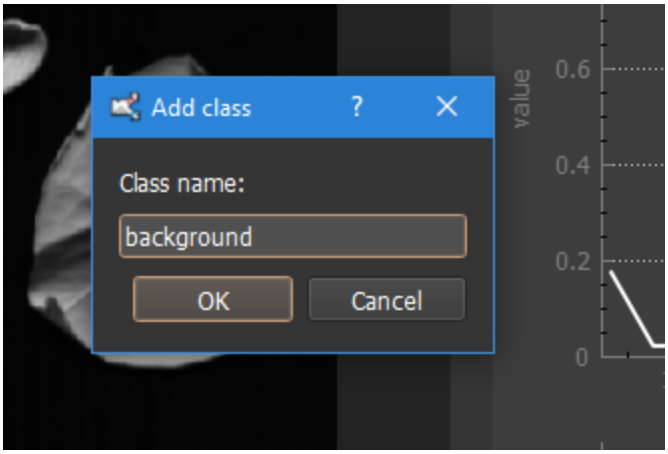
The process of building a classifier comprises three steps:

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

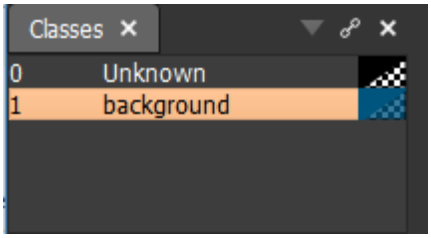
In order to define a class, right click in the Classes list and select *New class...* command:



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

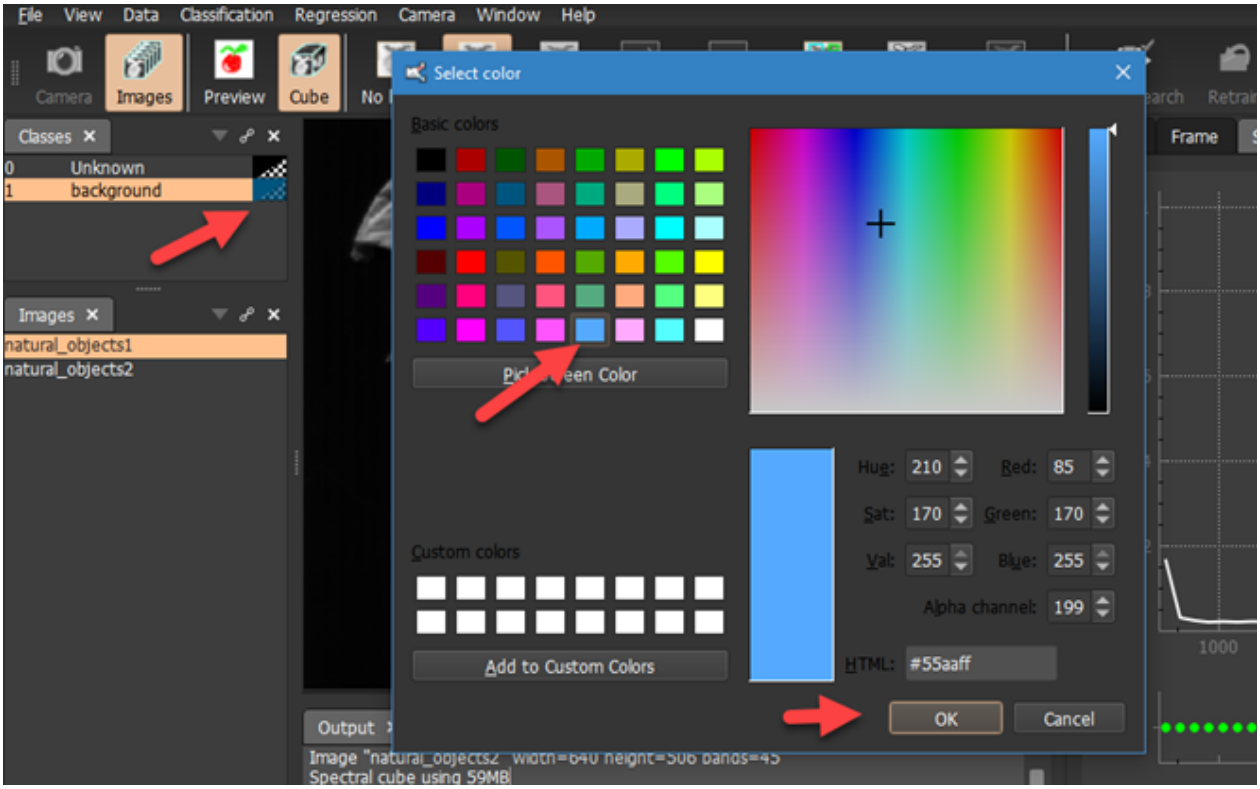


The new class will be added to the class list:

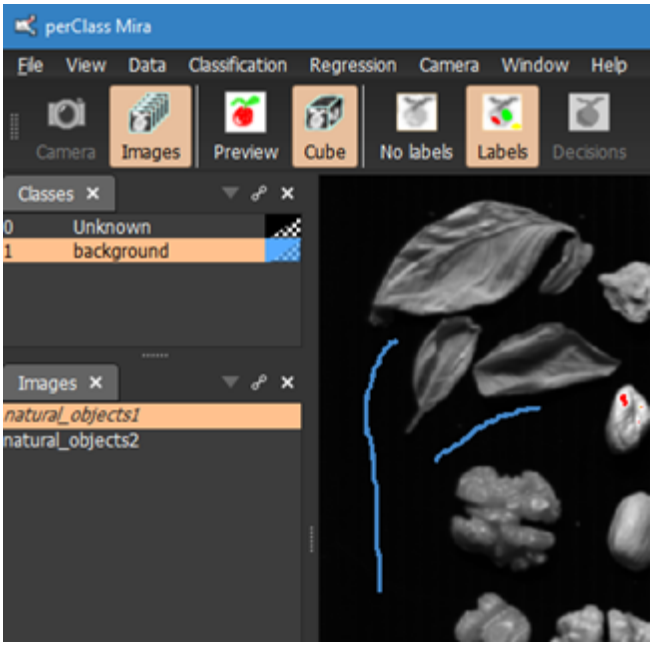


We may now label (paint) some pixels we consider background:

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.



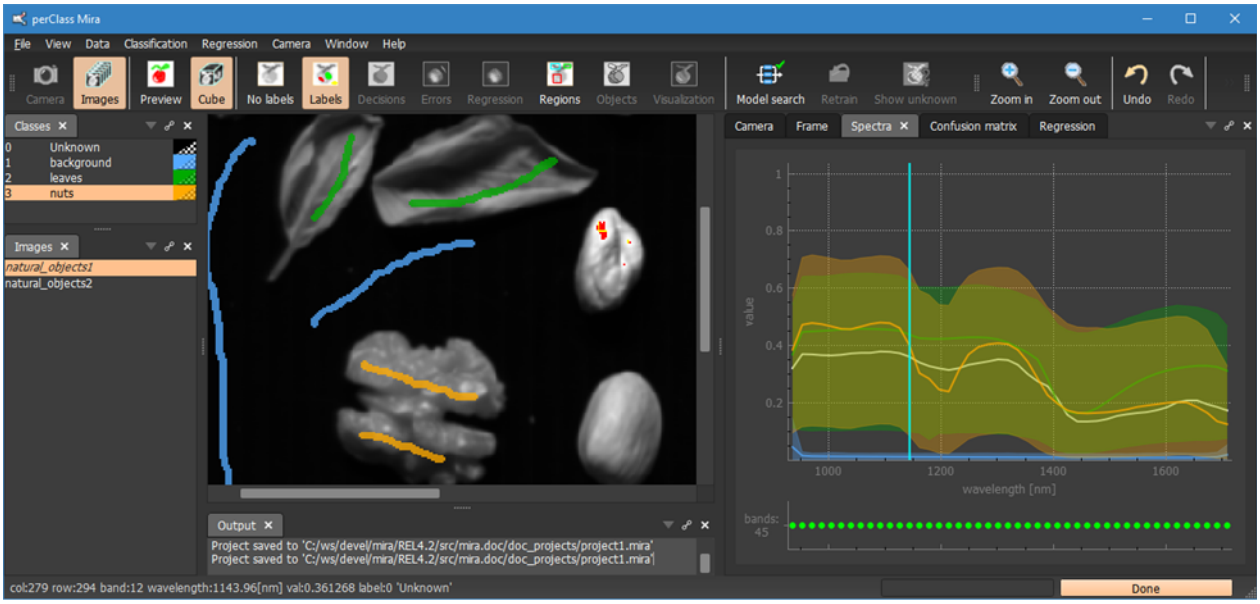
To zoom the view in or out, you may hold Ctrl key and use mouse wheel. This zooms aiming at the mouse pointer. Alternatively, use the Zoom buttons on the toolbar

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

TIP There is always one *Unknown* class available in the class list. Selecting it and painting also removes any existing pixel labels.

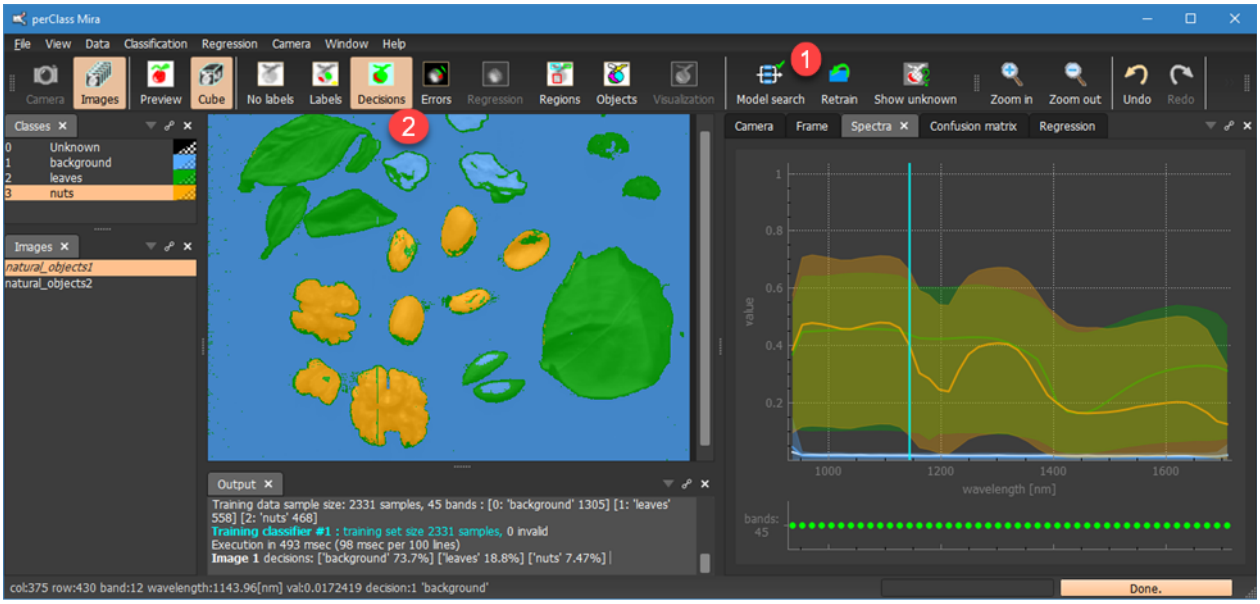
The brush size may be adjusted using the *Brush* button on the toolbar.

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.

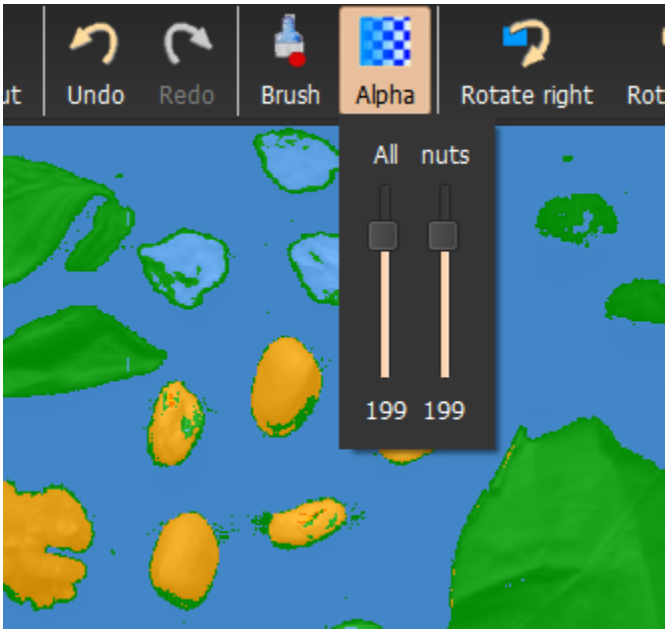


Switching between labels and decisions

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 trasparent 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-time 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

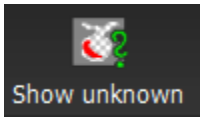
There are several ways we may improve the classification results:

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

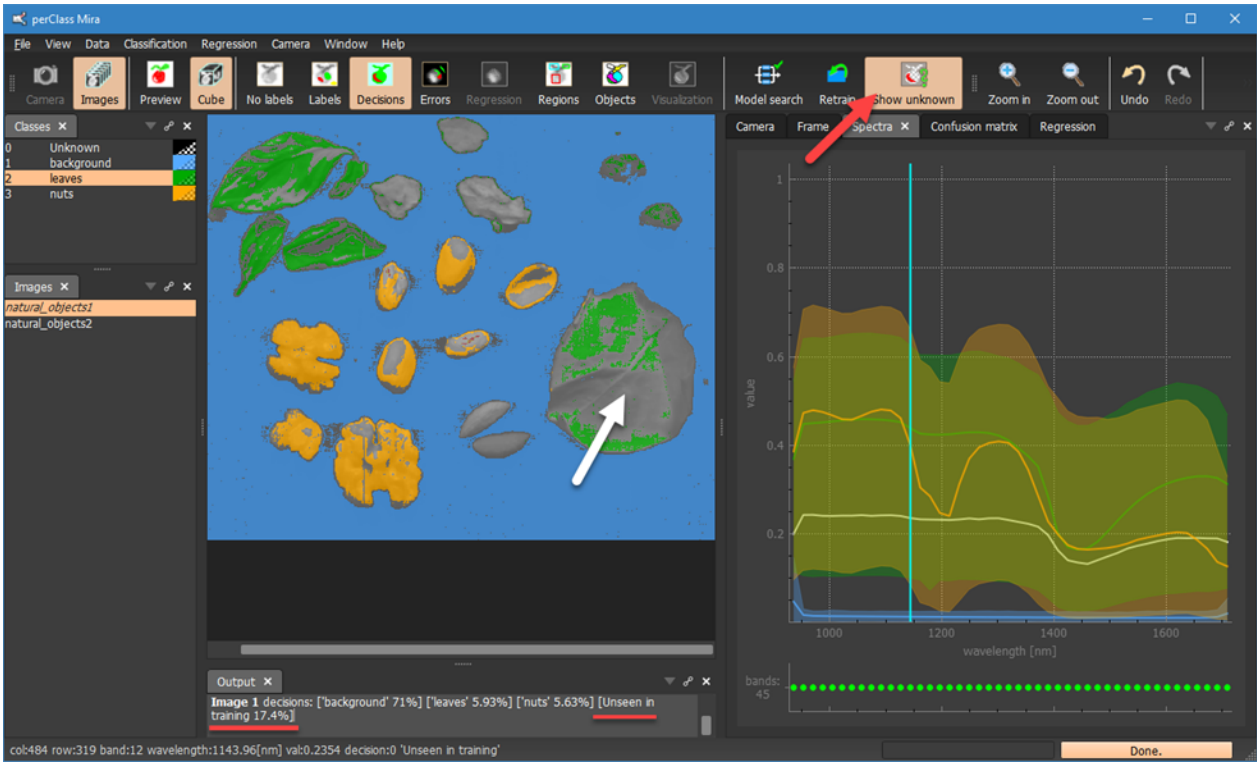
Subjectively, we may judge classifier performance visually by applying it to new images unseen in training. In order to assess objective classification performance, we may estimate error using confusion matrix tool on test images, unseen in training.

Improving the labeling

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

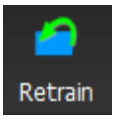
You may enable it using the *Show unknown* toolbar button 

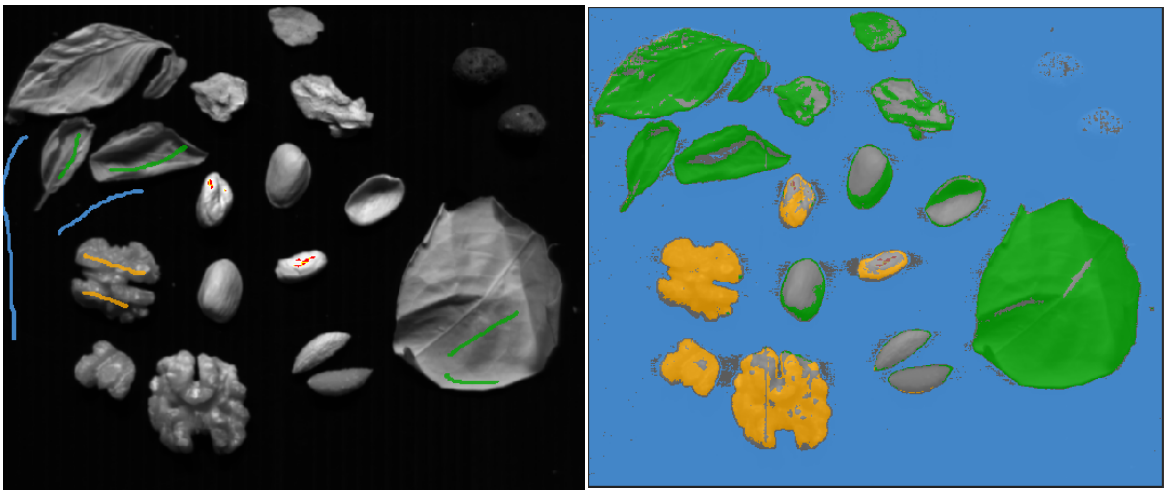
or by pressing **u** (**u**nknown)



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 



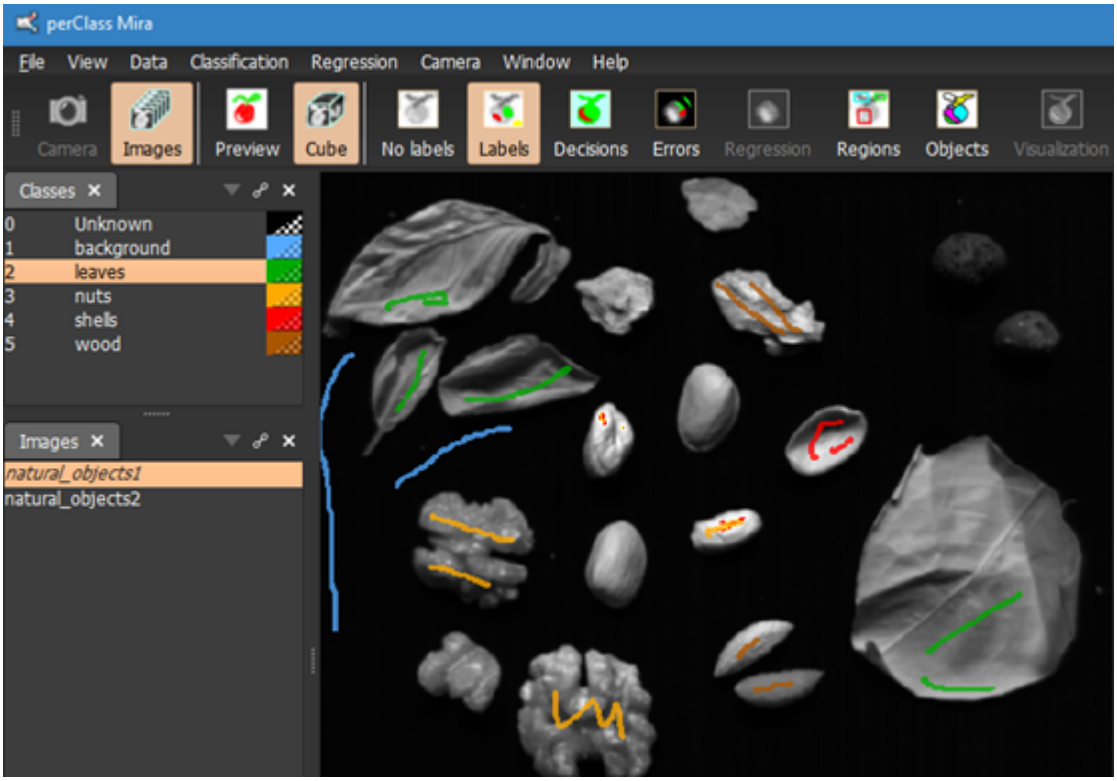
The *Show unknown* tool allows us to label in the areas needed and, thereby, building representative training sets.

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.

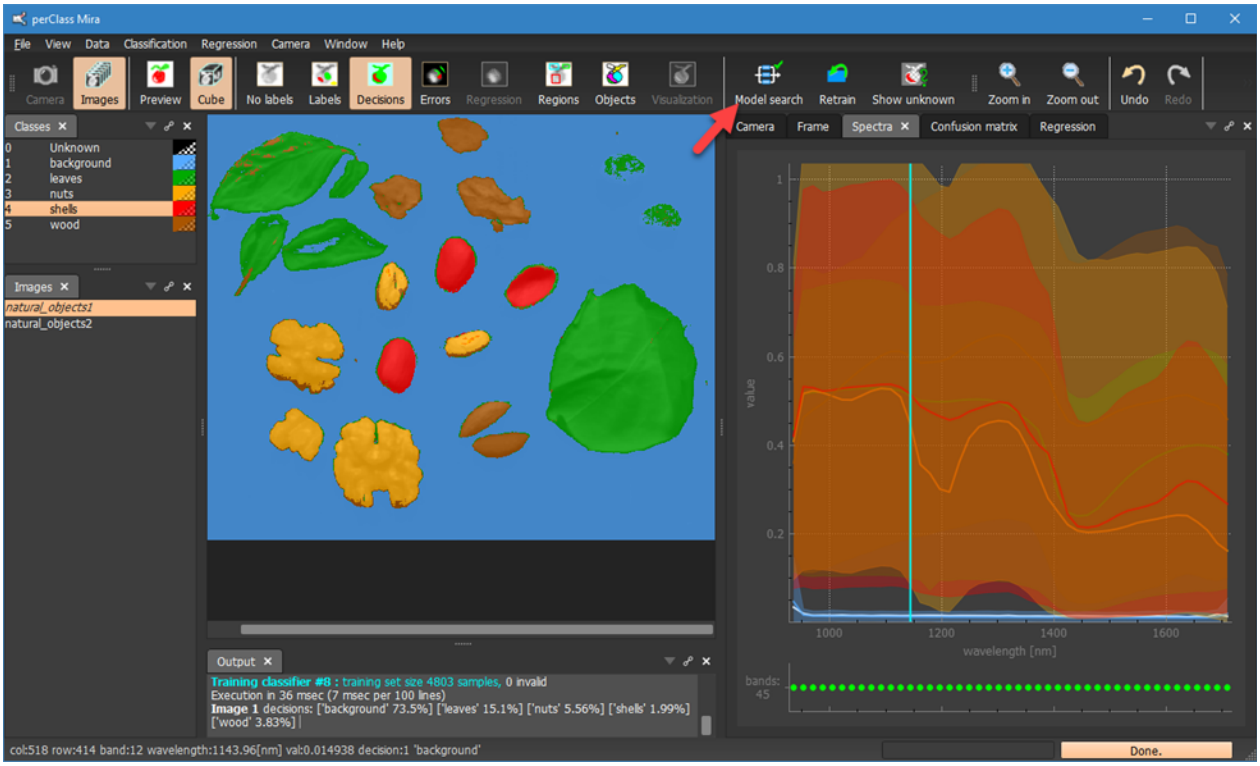
We add several classes to the project:



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

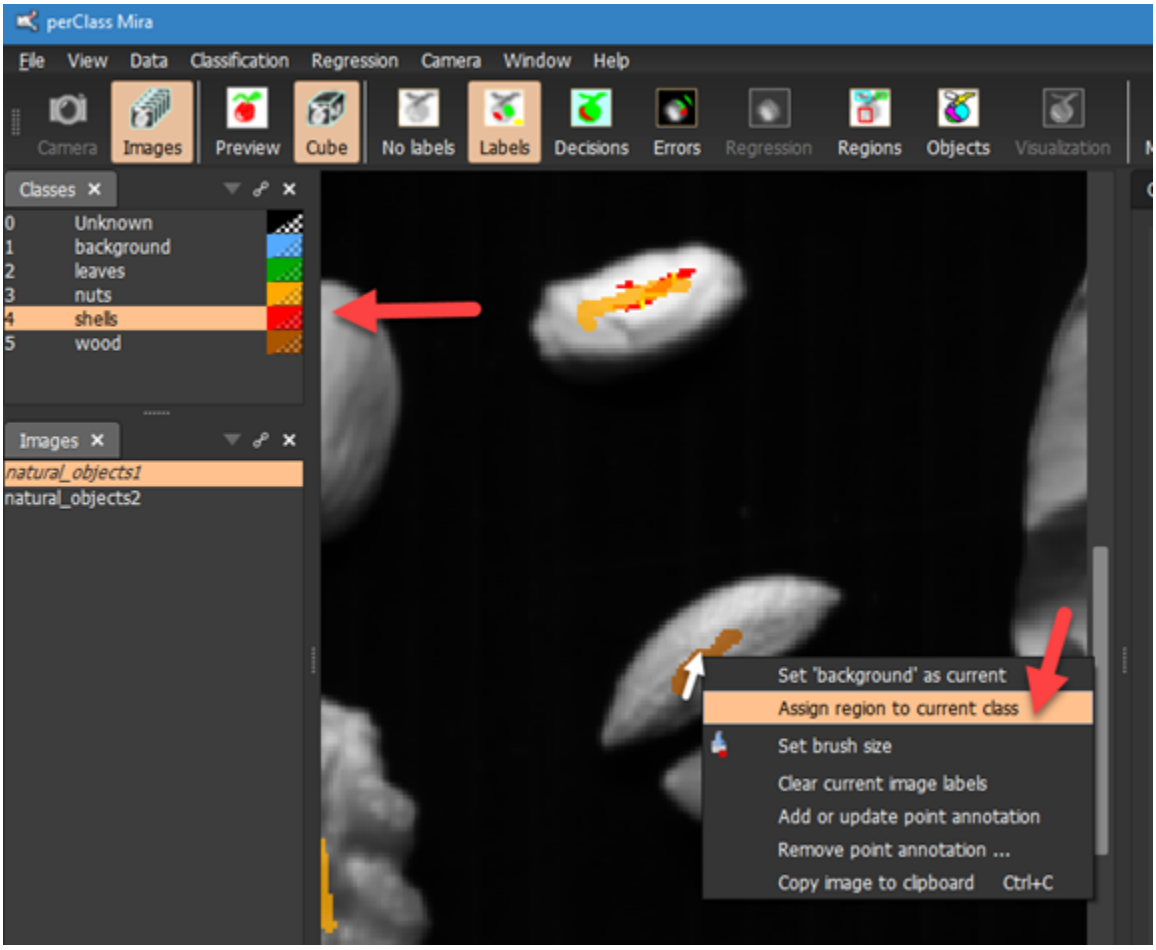
If we only perform slight update of the labeling, for example, around the edges or adding more representative examples of existing materials, Retrain is sufficient. The model search is useful when we add entirely new classes or label quite defferent examples of the existing classes.



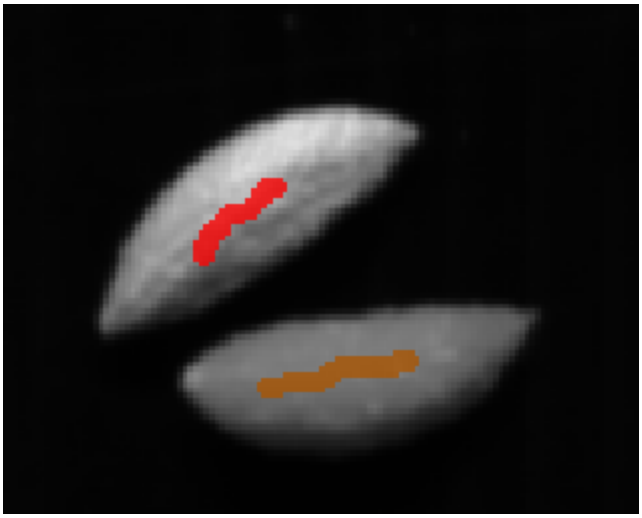
We can see that we have labeled olive kernels as "wood". We may wish to have them labeled as "shell" instead. In perClass Mira, we have a full control on the labeling. We can easily assign individual labeling strokes to different classes.

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

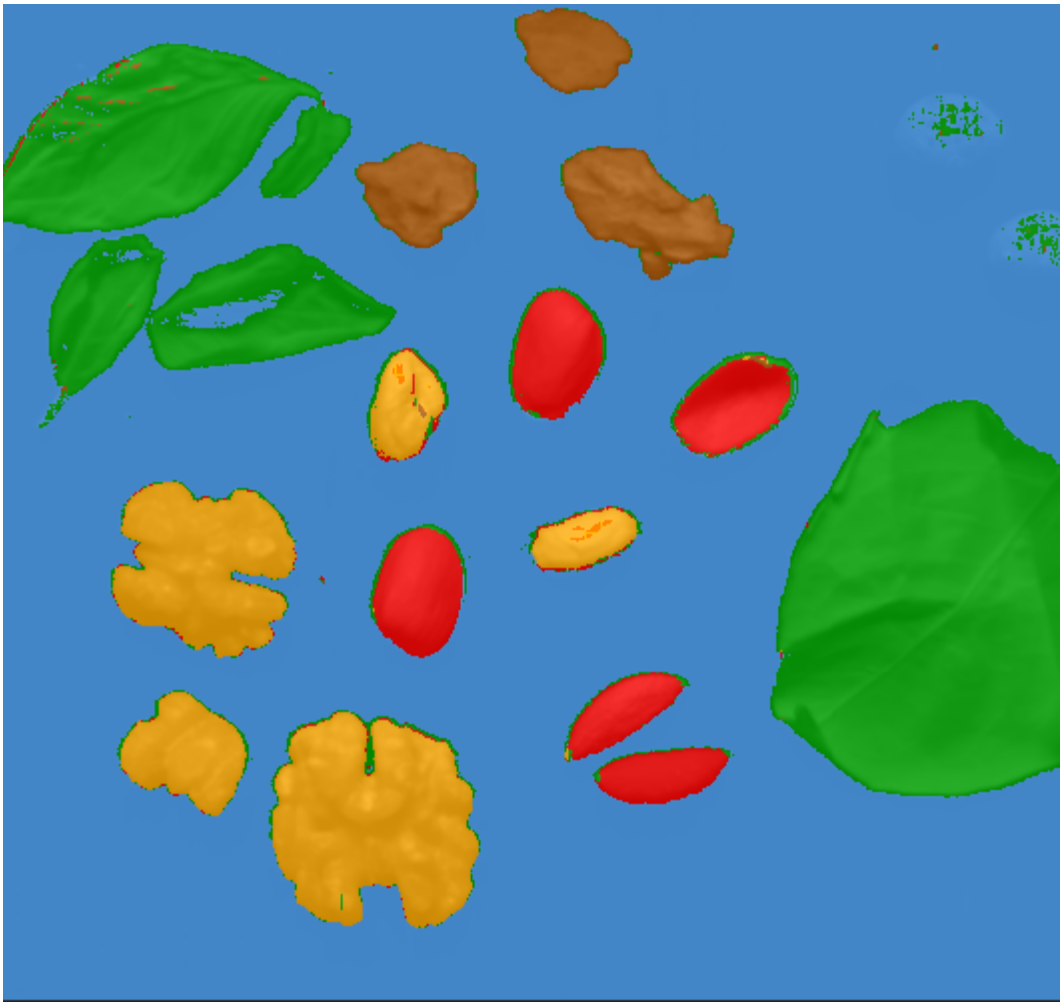
In the *Classes* list, select the class you wish to assign the stroke to (Shells). Then, right-click with the mouse over the brown label stroke on top of the olive kernel.
In the context menu, select *Assign region to current class.*



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.)



Where to go next?

In this Getting Started section, we saw how to load scans from disk, define classes and build a pixel classifier.

The next step, in many applications, would be to define object segmentation to extract objects. This is useful to either classify objects or to extract further information from the objects (such as mean spectra, spectral indices or shape information)

Acquire data and interpret

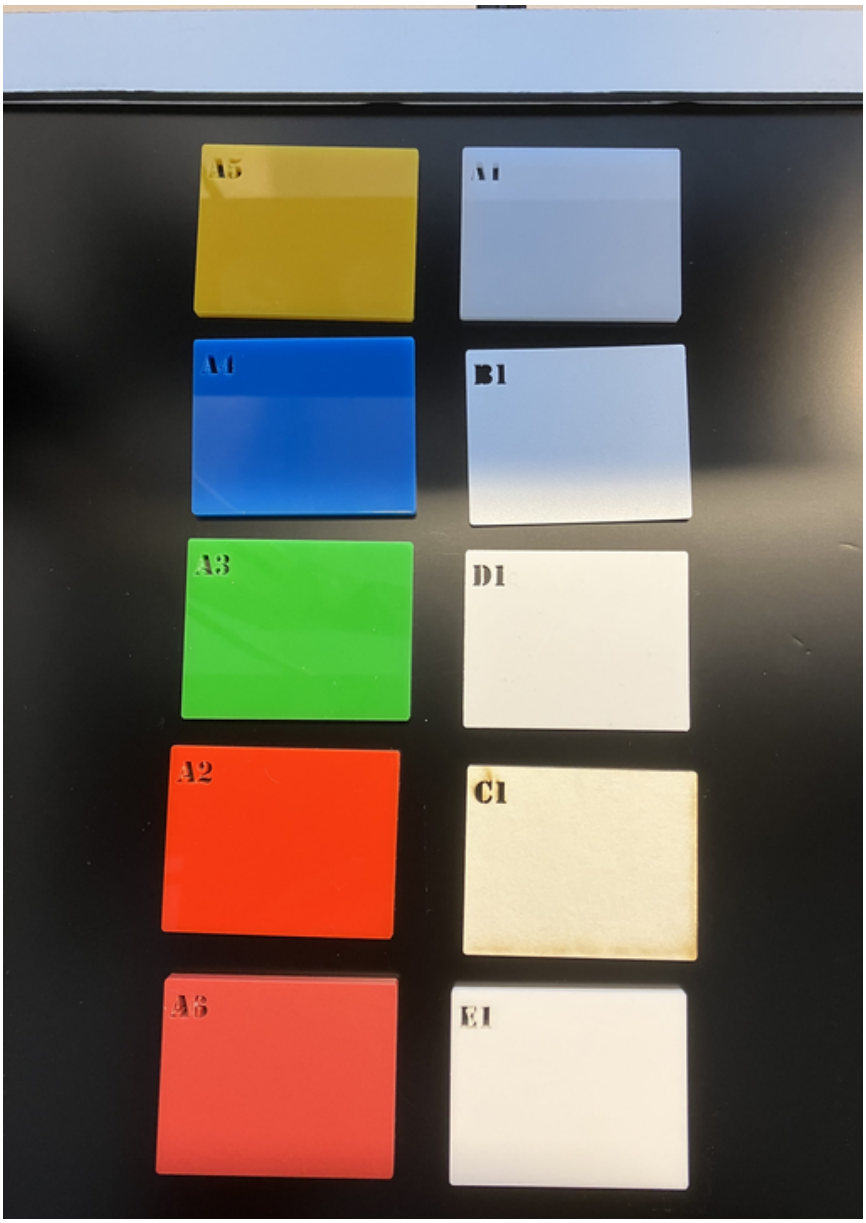
This tutorial illustrates how to

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

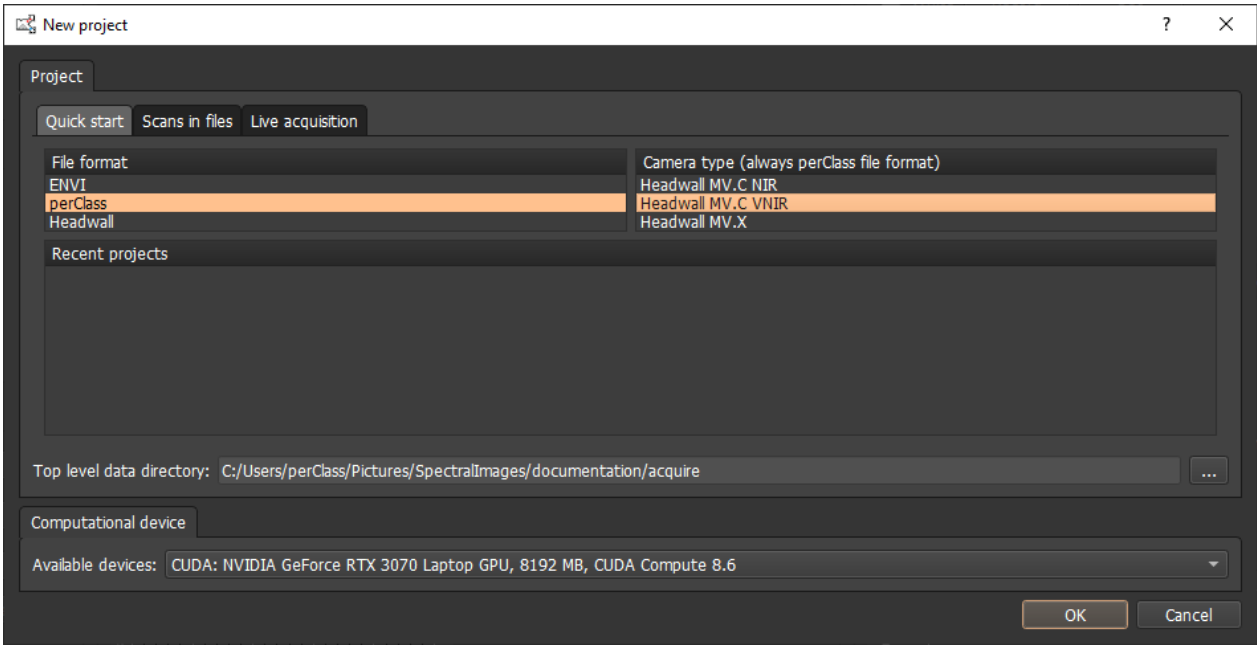
In this example, we use the default set of plastic samples included with each perClass Stage. These are tiles of different technical plastics with etched labels such as **A1-A6** and **B1-E1**. The letter corresponds to a material and the number to its variant. For example, **A1** and **A3** represent 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.

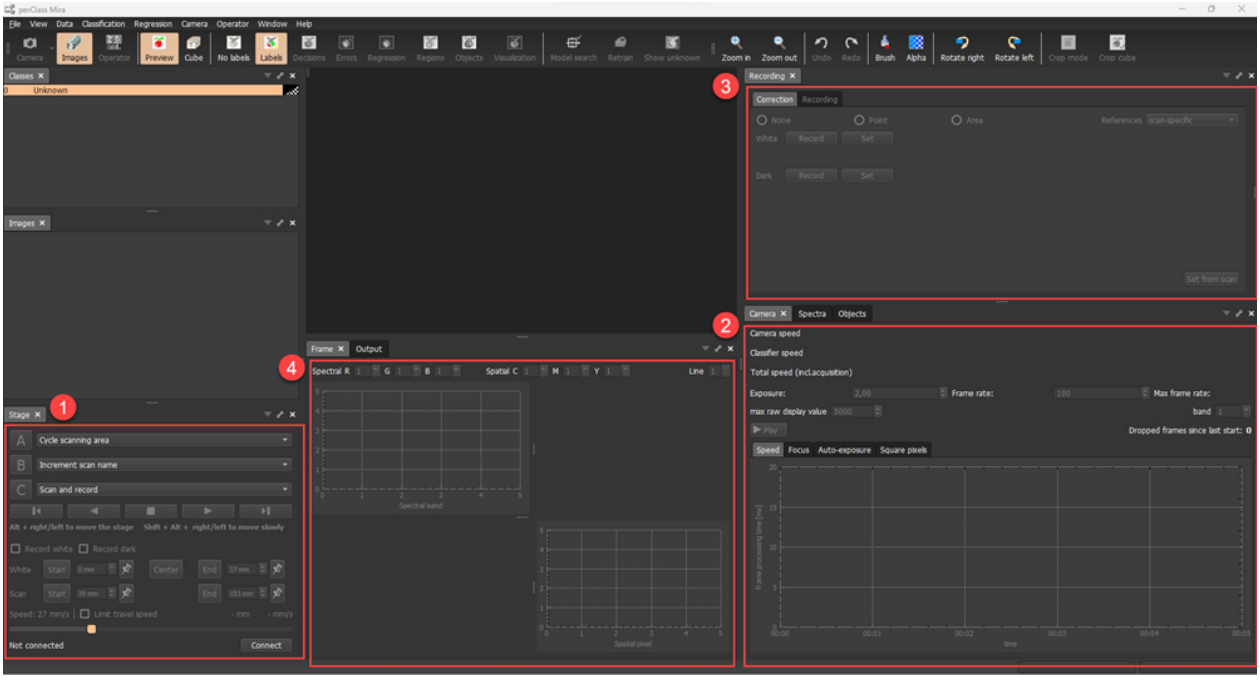


TIP: The Quick start tab only lists camera and project types we have flagged as favorite. You may select from all supported acquisition devices in *Live acquisition* tab and project types in *Scans in files* tab.

Connecting to the stage

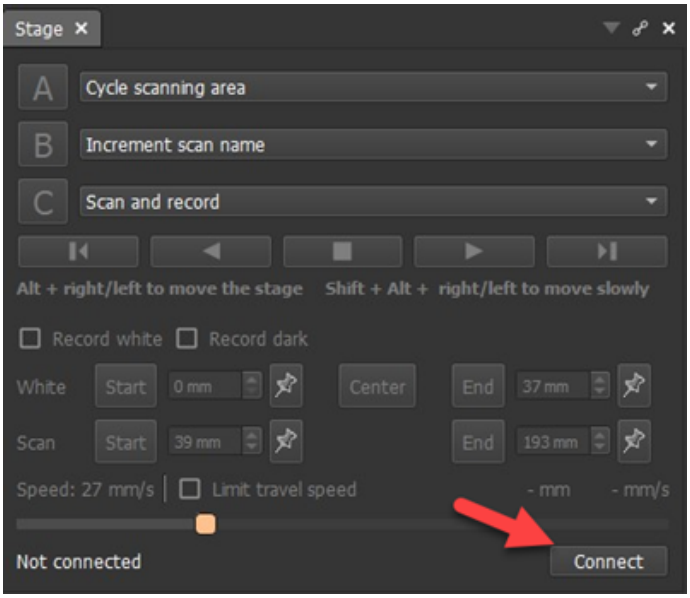
In order to scan, we need to connect to perClass Stage and to the camera.

Recommended screen organization for scanning is as follows:

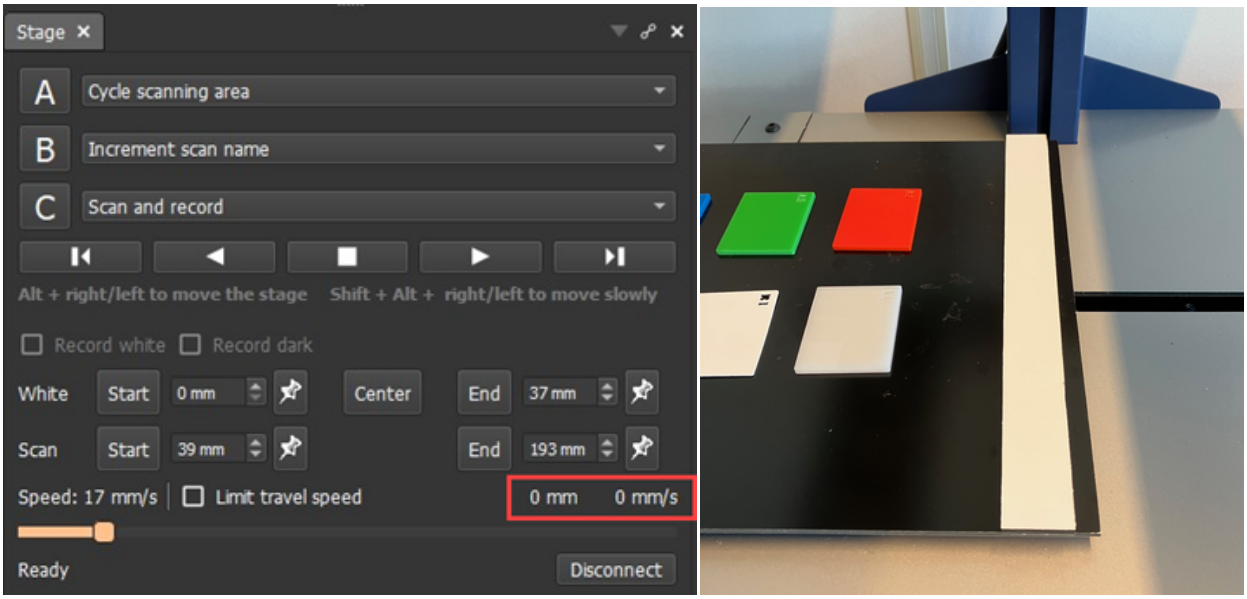


- the **Stage** panel in the bottom-left provides all stage controls.
- the **Camera** panel shows controls of the acquisition
- the **Recording** panel in top-right allows us to define references and scan recording settings
- The **Frame** panel in the center shows the raw signal from the camera

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

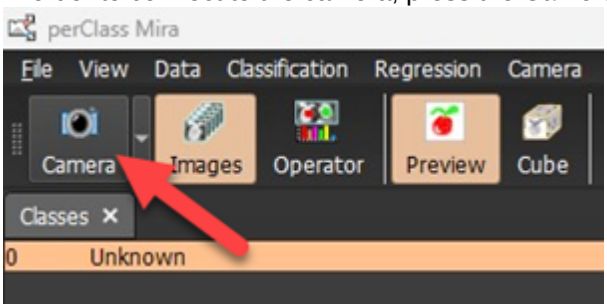


The stage performs a homing run. The white reference block will then be under the camera. This represents the "position 0" of the stage.

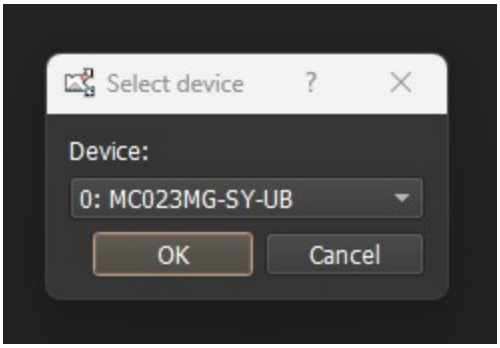


Connecting to the camera

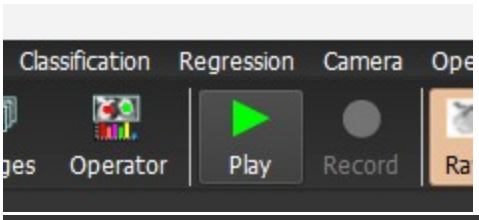
In order to connect to the camera, press the *Camera* button on the left side of the toolbar:

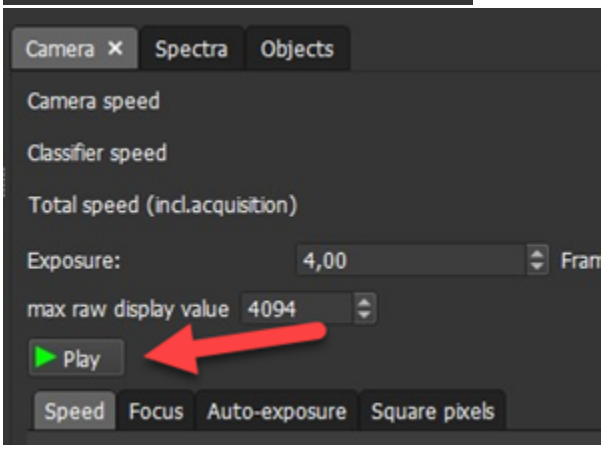


When the camera is initialized for the first time, we are offered a dialog to select the device:



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





The *Camera* panel then shows the speed of acquisition, namely the camera speed (light blue) and the total

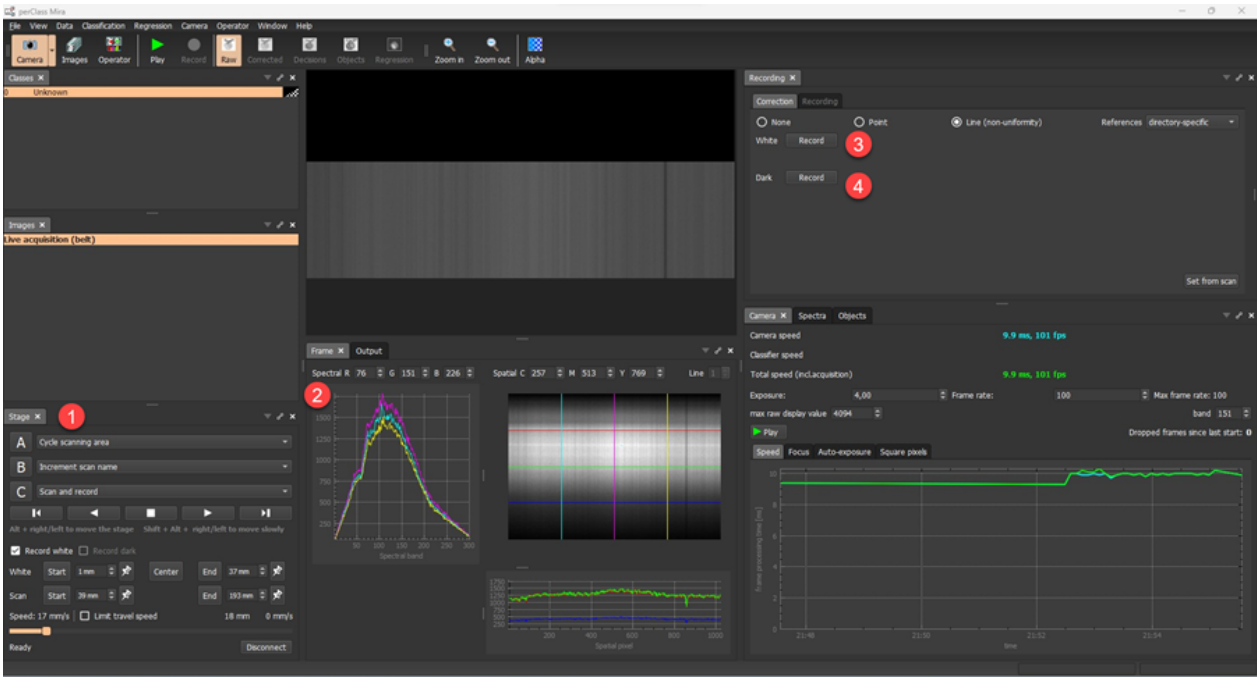
speed including also user interface updates and all other activities (in green).

Acquisition can be paused by the *Pause* button (1). The *Camera* panel can be used to adjust acquisition parameters such as exposure (integration time) (2) and frame rate (3). In this example, we keep the default settings.

Recording references

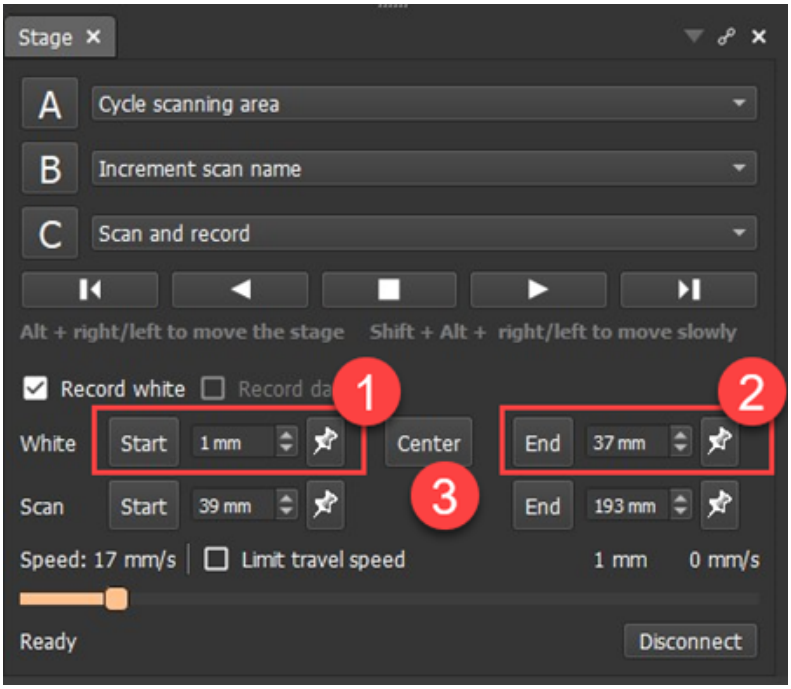
In order to interpret spectral images, it is highly recommended to correct the raw data from the sensor into reflectance. This operation makes the data robust to illumination changes and thereby makes the interpretation models more generally applicable.

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. There are several ways how to move the stage. The easiest is to use the *Alt + right / left* keystrokes. We can see actual raw data from the image sensor in the *Frame* panel (2)

Defining the position of white reference bar



The recommended best practice is to move to the start position *on top* of the white reference and fix this **white start** 1 in the *Stage* panel. It is important that the entire field of view is fully occupied by the white reference. In order to fine-tune the position, we may move at slow speed with *Shift* + *Alt* + *right / left* keystrokes. We can use the pin button 1 to fix the position.

Similarly, we will fix the **white end** 2 .

We can then move to the center of the white reference by pressing the *Center* button 3

Alternative ways to move the stage are:
1. Using the dedicated movement buttons
2. Via Stage A, B and C buttons. We may assign specific movement commands in via the respective combo-boxes in the upper part of the *Stage* panel

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.

TIP The keystrokes to invoke A, B and C stage buttons are the bottom three numerical pad keys: the "digit 0", "the decimal dot" and "Enter key". This allows one to easily invoke stage buttons via keyboard without looking.

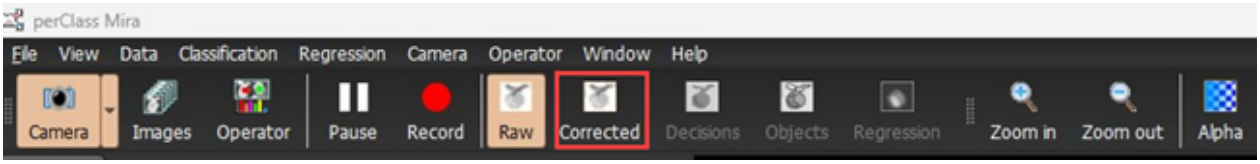
Acquiring references

Once we are on top of the white reference, we can acquire it by pressing the *Record* button 1 in the *Correction* tab or the *Recording* panel.
Because MV.C VNIR camera does not have integrated shutter, we need to cover the optical lens before acquiring the dark reference using button 2. For other hyperspectral cameras, such as Headwall MV.X, MVC.NIR or Specim FX, the integrated shutter will be closed automatically before aquiring the reference.

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

TIP After acquiring references, the Output panel of perClass Mira lists the min, max values and the mean and standard deviation statistics across all bands. This helps us to easily spot common errors such as shifted/inhomogeneous white reference or not fully closed shutter when acquiring the dark reference.

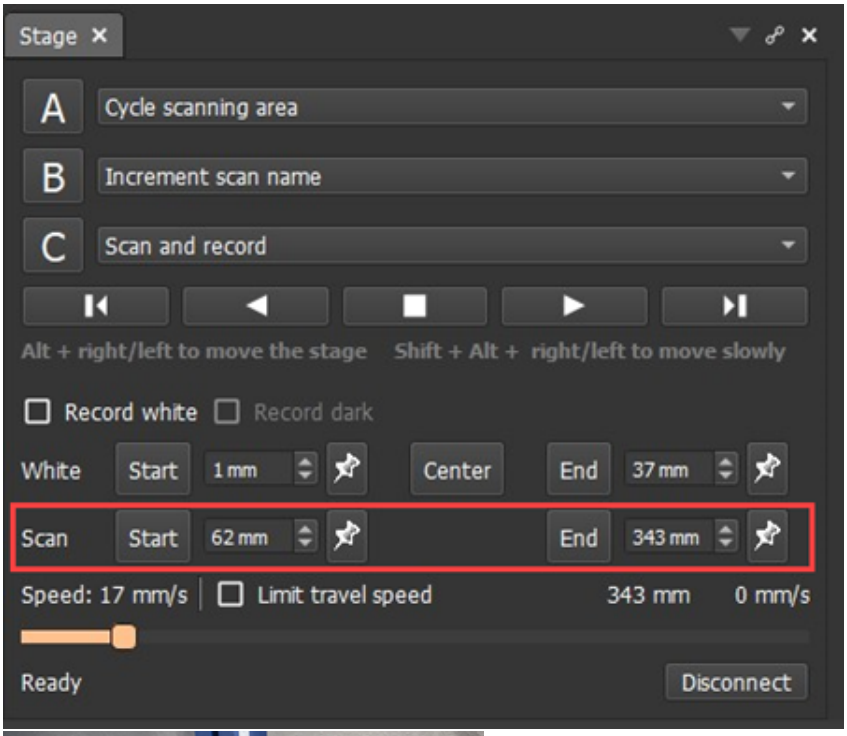
Once the references are acquired, we may switch to the *Corrected* stream visualization in *Camera* mode.



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 to be easily spotted.

Defining a scan area

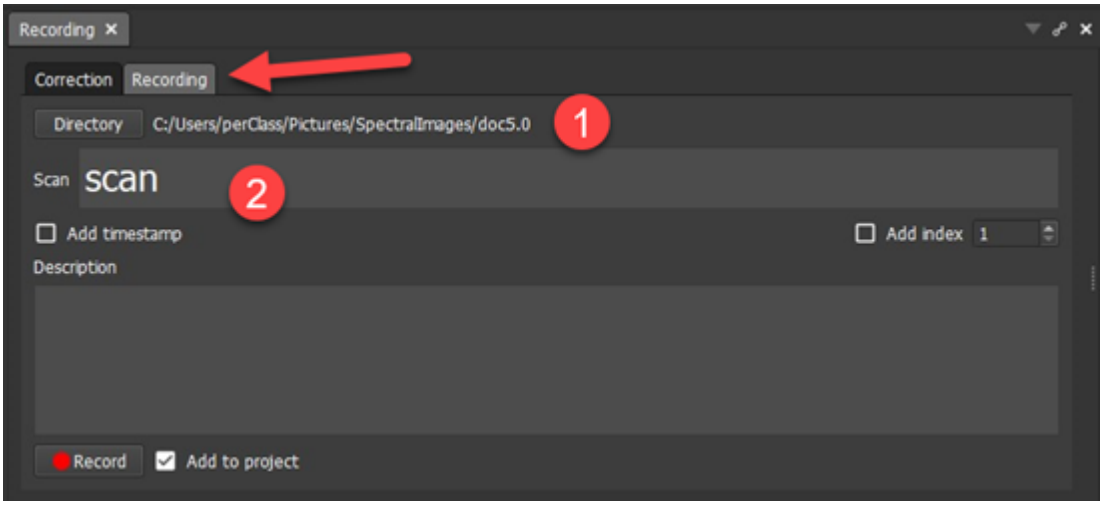
In order to record a scan, we define what area of the scanning table we wish to acquire. We start the camera (*Play* toolbar button in *Camera* mode) and move the stage by **pressing and holding** the *Move right* button.



We can then define the start and stop of the scan area by pressing the respective pin buttons

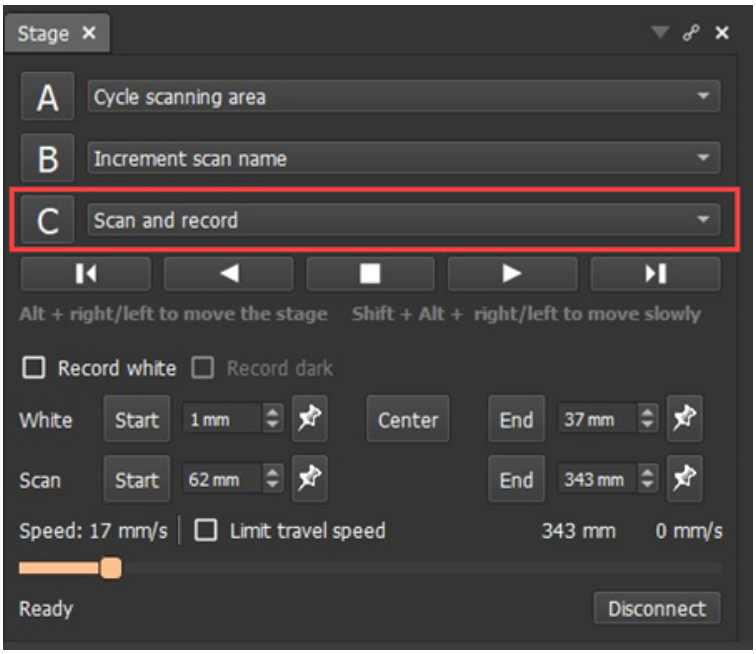
Recording a scan

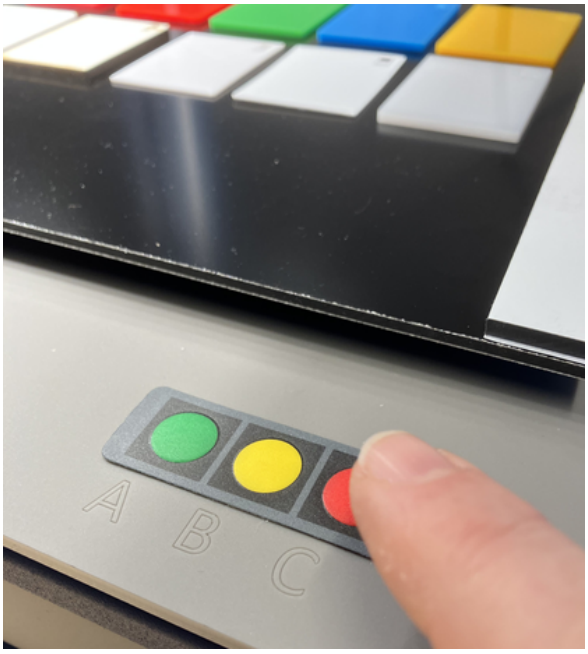
In order to record a scan, we can switch to the *Recording* tab next to *Correction:*



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

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 pressing 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.

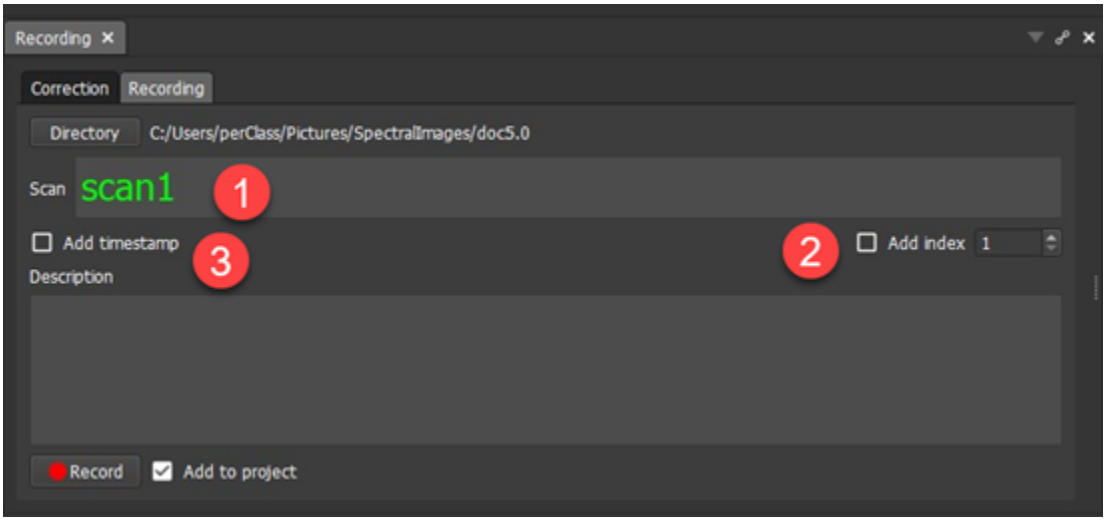
If you press the *Record* button in the *Recording* panel, only acquisition and recording into a file will start, not the stage movement. This is useful, when recording data without perClass Mira Stage, for example on an industrial conveyor belt. In order to simultaneously move the stage and record, the *Scan and record* command is needed.

It is a good practice to scan multiple images for the sake of testing any developed models on data unseen in training. In order to help with naming multiple scans, perClass Mira provides three options:

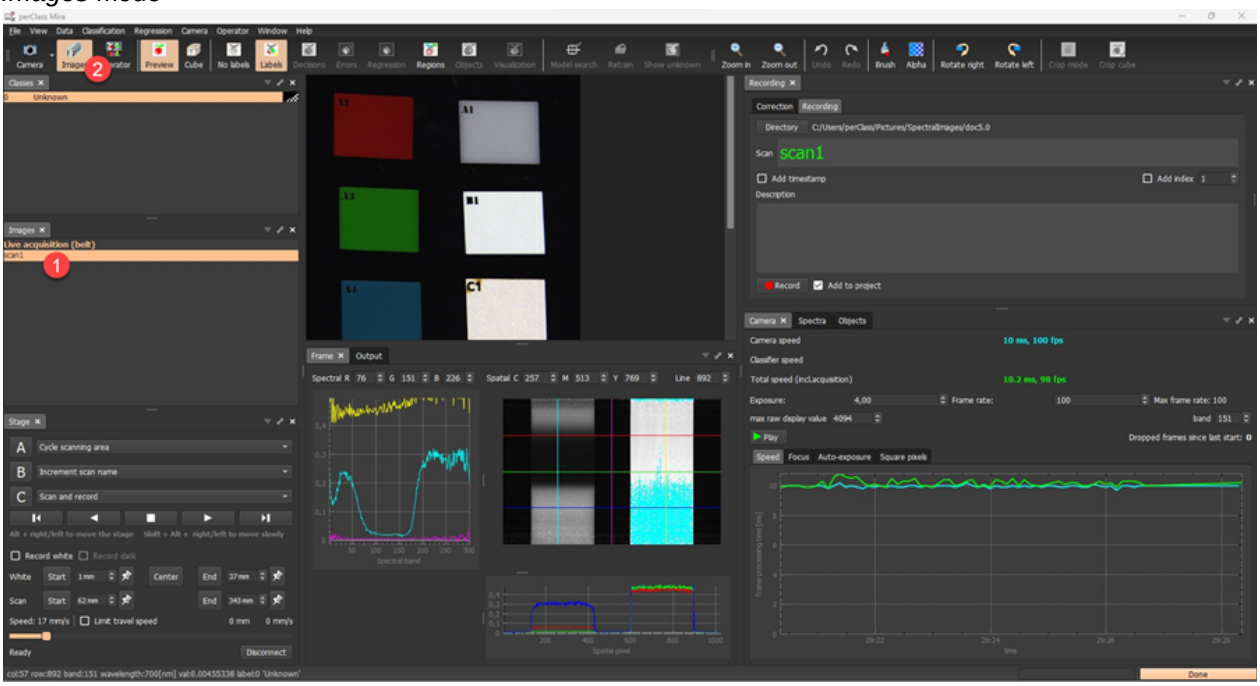
1. If the scan name contains a trailing digit, the name turns into greeen. Using cursor keys or mouse wheel, we may increment or decrement this traling index. Words, seprataed by undescores, form different groups. This can be used to add metadata on scanning conditions (day, variety, supplier etc.)

TIP: Current index can be also increased by the stage button assigned to the *Increment scan name* command. In this way the scanning process can be fully controlled by stage without use of keyboard or mouse

2. By checking the *Add index* checkbox, the underscor + index will be added to each scan name. Starting index can be changed in the adjacent spinbox.
3. By enabling the *Add timestamp* checkbox, the current time stamp at the start of acquisition is appended to the file name (Note, that timestamps are also stored in header files, so adding it in the filename is not necessary in all use-cases)



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

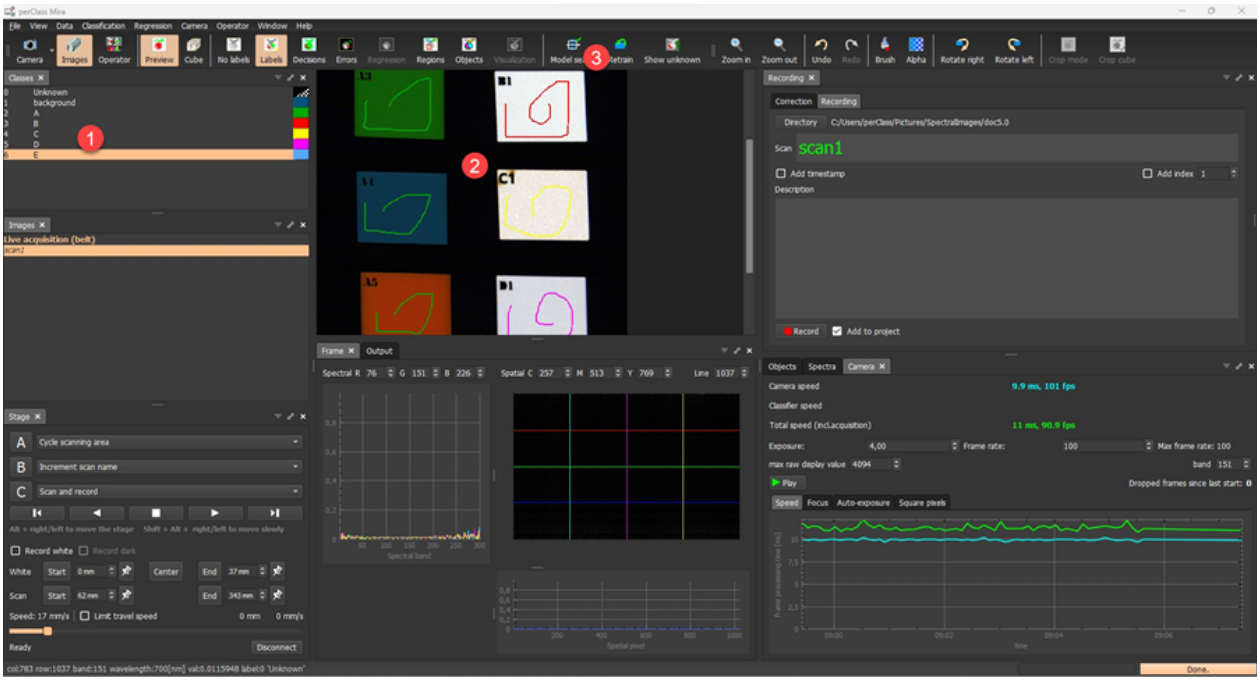


perClass Mira interface is optimized for efficient acquisition of a large number of scans. All you need to do is to place new physical sample on the scanning table, increment the scan index by **B** button and acquire another scan by pressing the **C** button.

We're now ready to interpret our first scan.

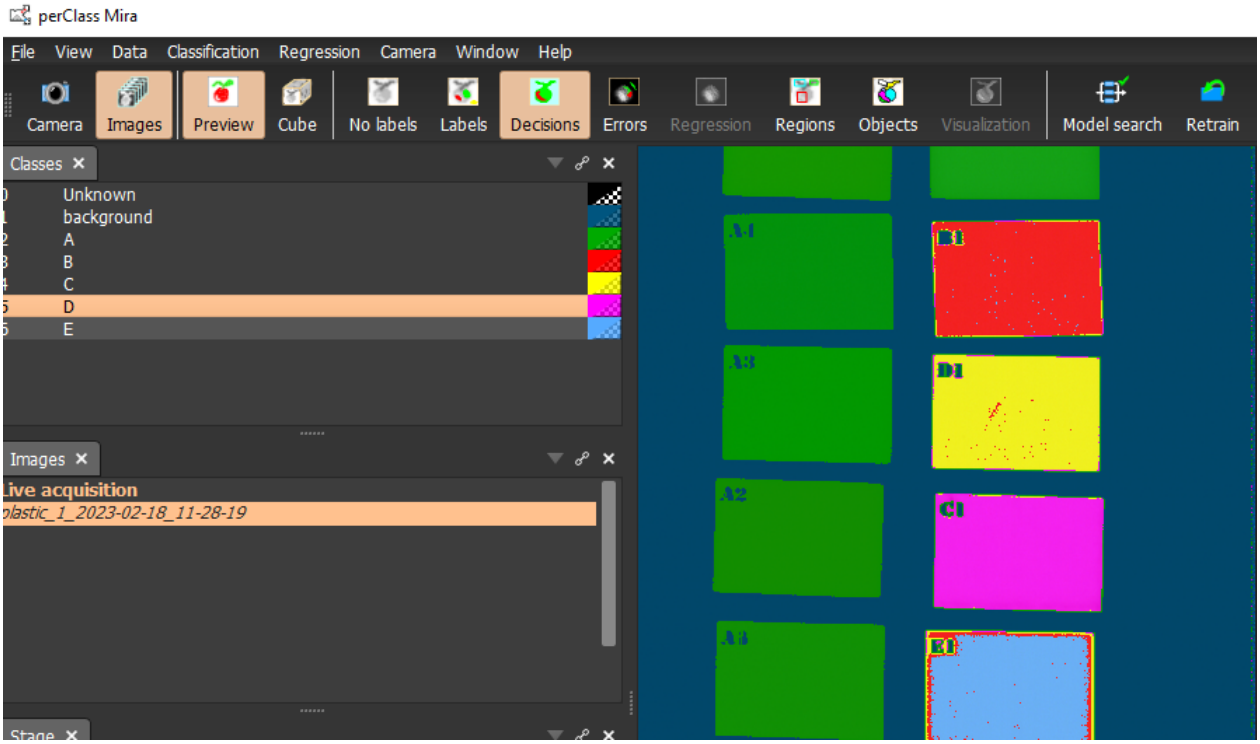
Building a classifier and applying to live data

We can design our classifier in the familiar way. We define classes, paint labels and build a model. The plastic samples, included with each perClass Mira Stage, are marked with a letter, followed by a number. The letter describes the material, and the number its variant. In our example, we want to distinguish different plastic materials, irrespective of color. Therefore, we define background and material classes A - E 1

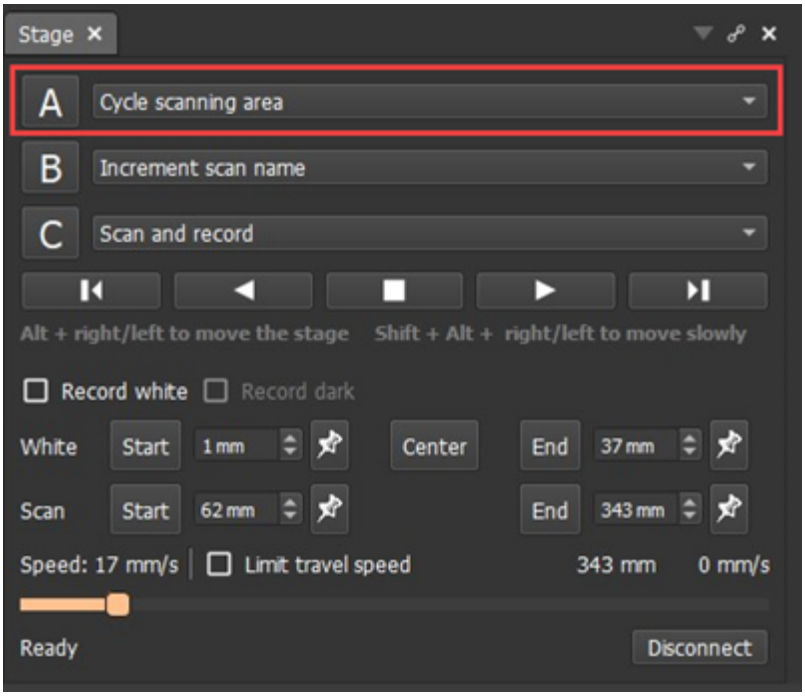


Then, we pain the labels 2 and create a model by pressing *Model search* 3

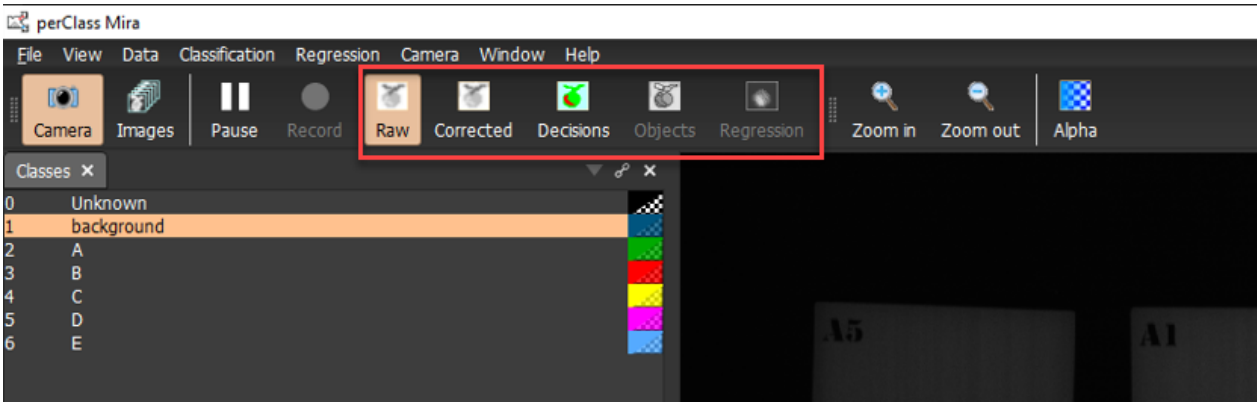
We can see that all different variants of material A can be separated from different white plastics B-E. This demonstrates the unique value of spectral imaging where we may base our interpretation on material composition, not appearance.



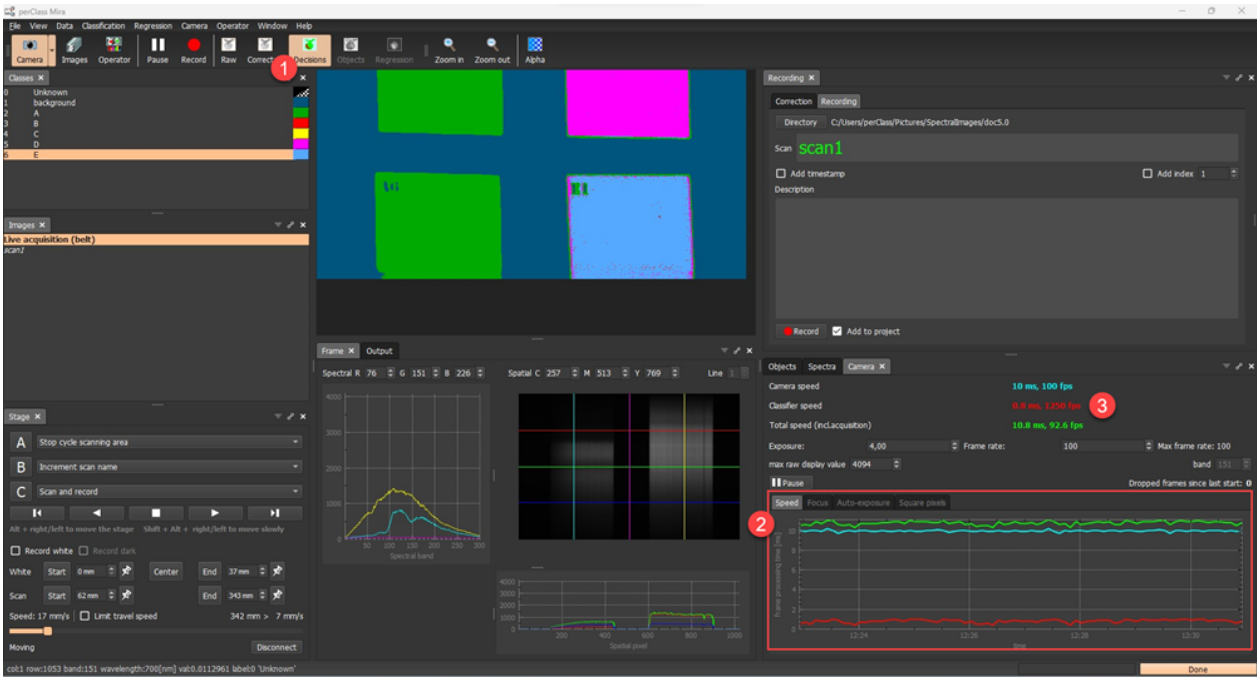
In order to demonstrate how our solution works live, we assign the *Cycle scanning area* command to the button **A** in the *Stage* panel. By pressing **A** button of the stage, the software switches to the *Camera* mode and stage starts cycling.



In the toolbar, we may now see also the *Decisions* button enabled, in addition to *Raw* and *Corrected* buttons. We may choose how do we want to see the live data stream.



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



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.

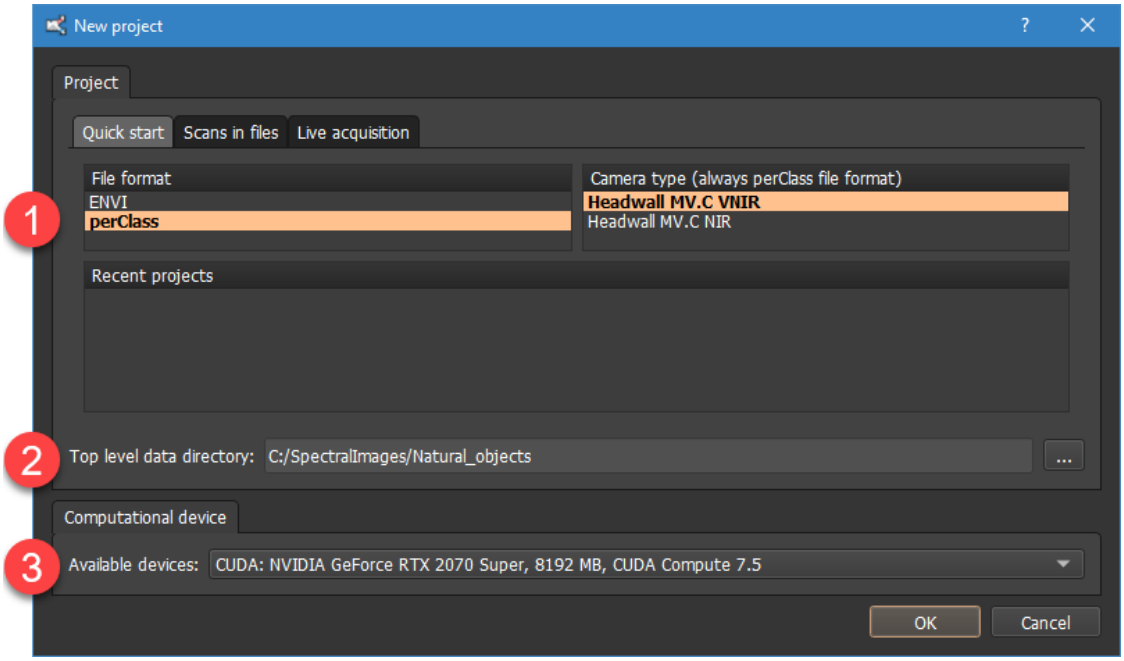
User guide

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 avaialble 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.

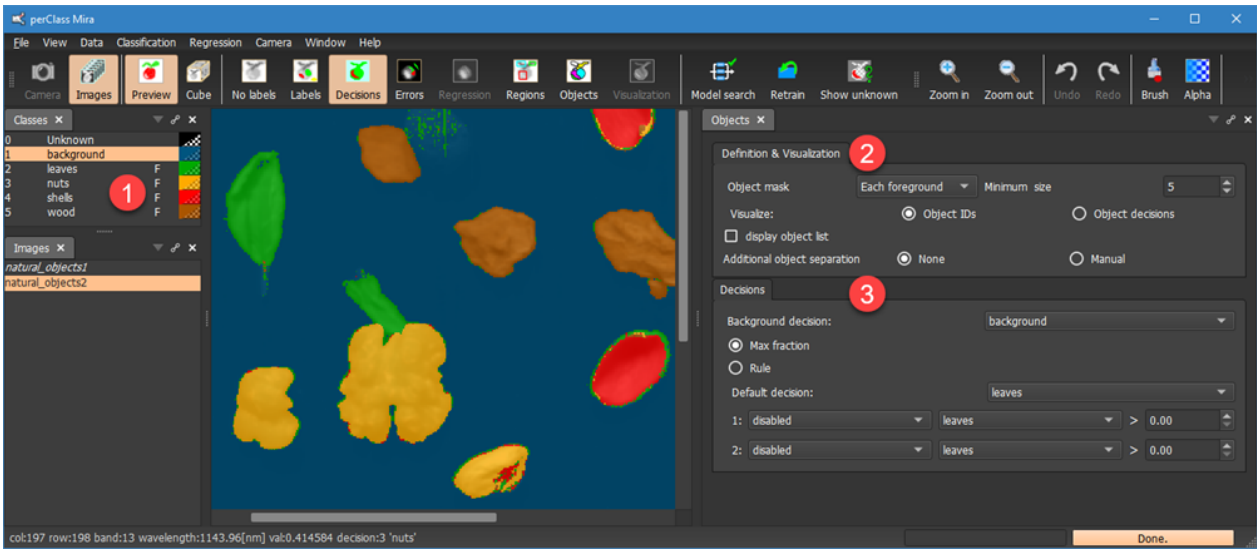
TIP: In case you experience crashes or very slow operation when using a GPU, please update your GPU drivers to the latest available. For most users this resolves the issues.

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 interst, model quality of some objects using regression analysis or export mean spectrum per object for further research.

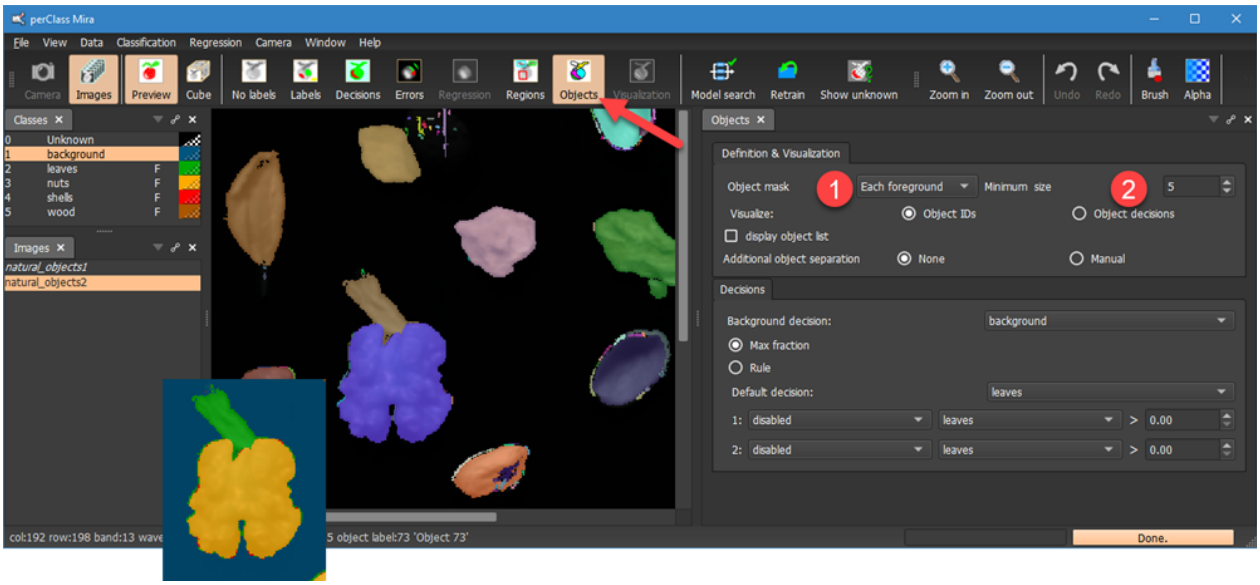
The first step to apply object segmentation is to define one or more *foreground* classes. This can be done using *Image list* context menu and selecting *Flag class as foreground*. Alternatively, the 'F' shortcut toggles the foreground flag for the currently selected class in the *Class list* 1

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

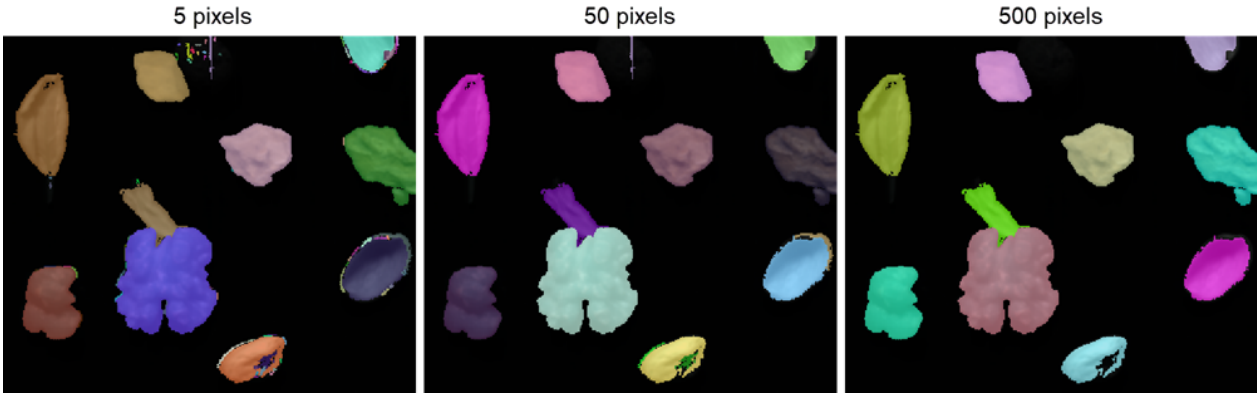


Object segmentation

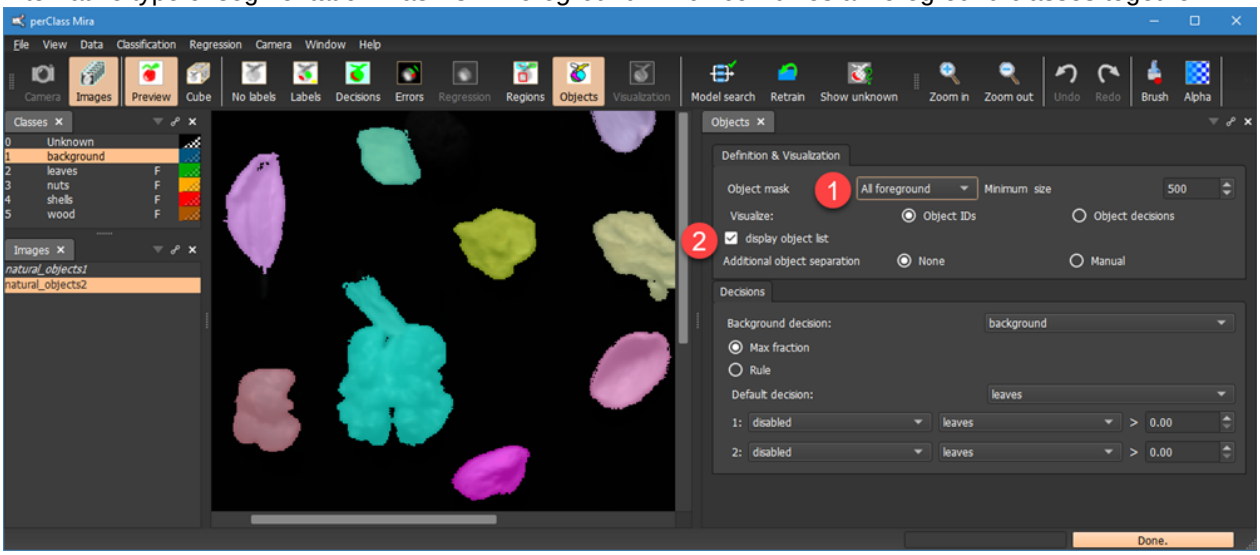
Object segmentation is performed by selecting *Objects* toolbar button. In perClass Mira, object segmentation is controlled by segmentation mask. By default, it is constructed handling each of the foreground classes separately. This is the *Each foreground* mode 1 . Note, how the leave touching the walnut in pixel decisions are segmented separately.



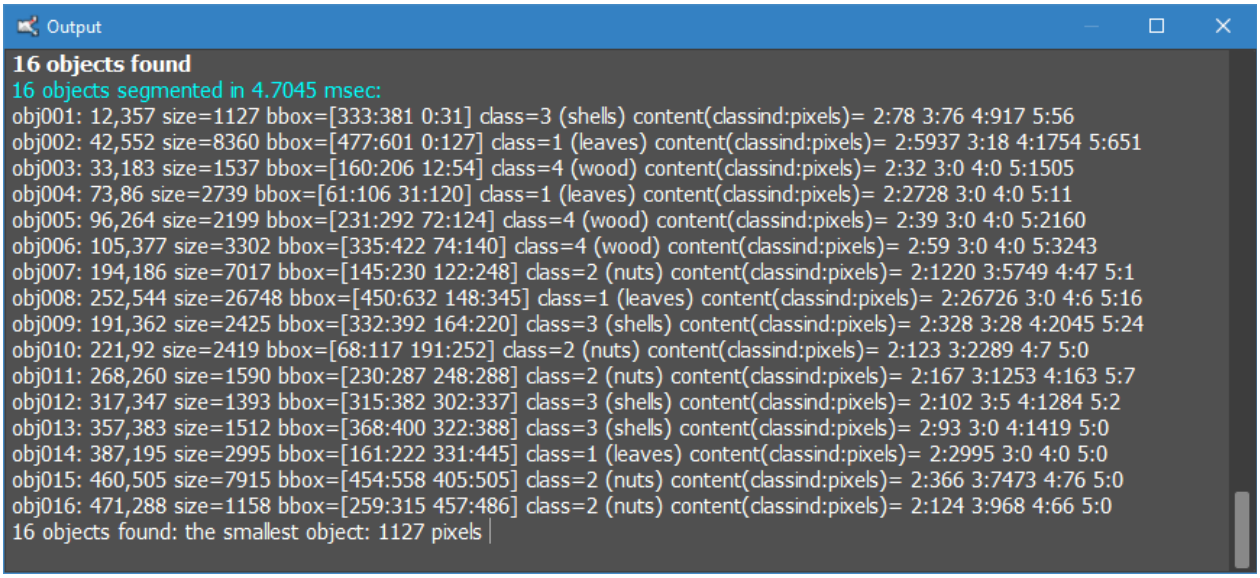
Object segmentation directly removes all objects smaller than specified *minimal size* 2 Changing the minimum size, we can quickly focus on large structures:



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:

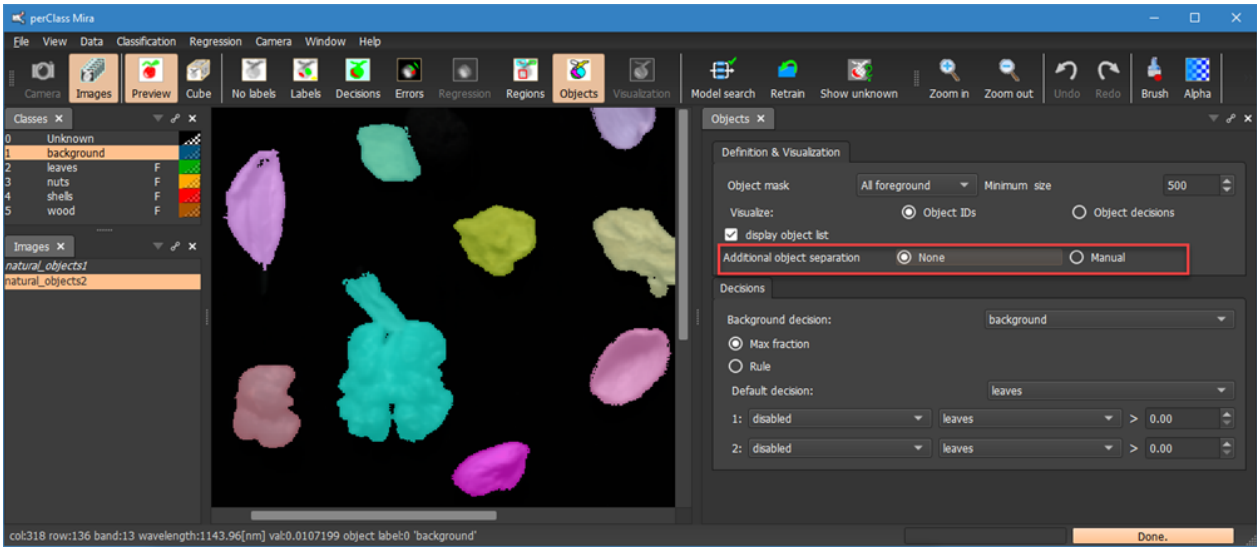


This includes object id, centroid, size in pixels, bounding box and object decision. At the last line, the smallest object size is also provided. This is helpful when adjusting minimum object size.

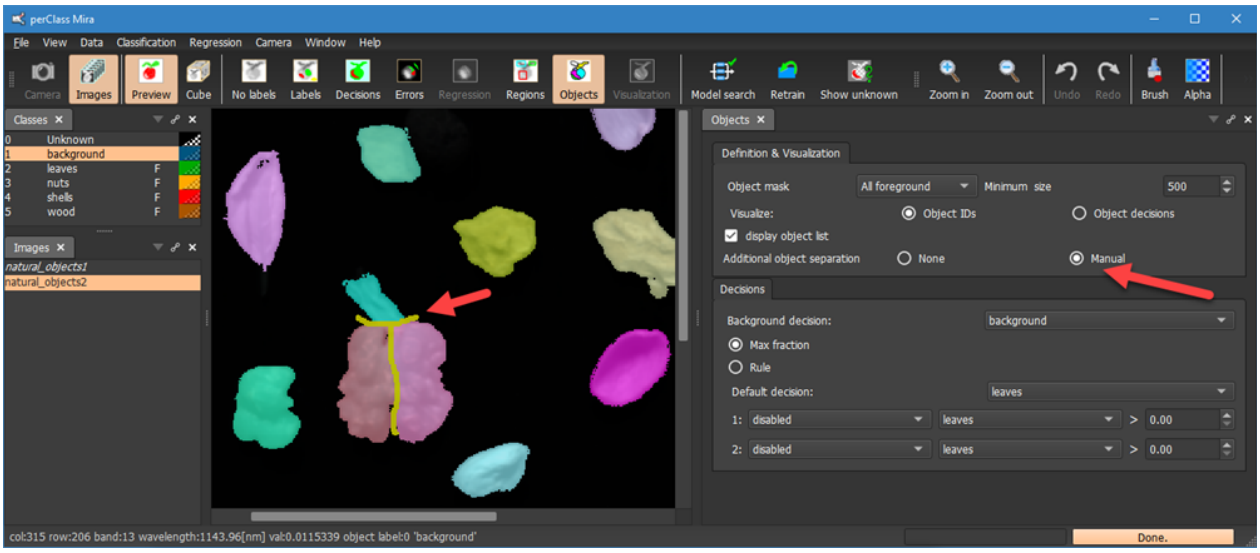
Object separation

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 possile using the *Additional object separation* tool:

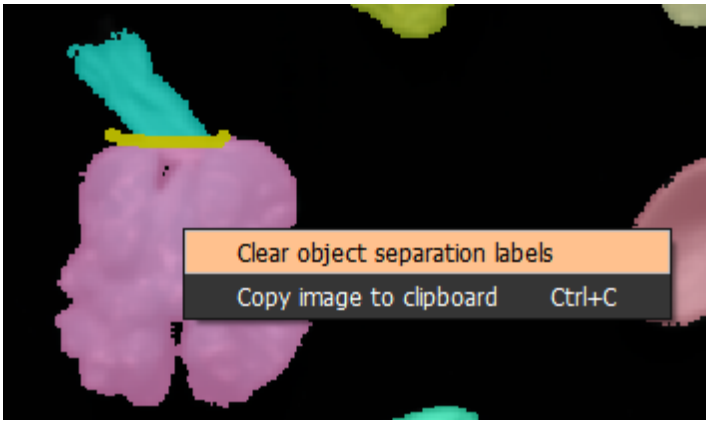


By setting the *Manual* option, we may define object separation by **drawing background labels** in the object segmentation result.In our example, we may force separation of the leaf from the nut even when we use the *All foreground* mode or separate the nut object into two parts (for example, to extract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

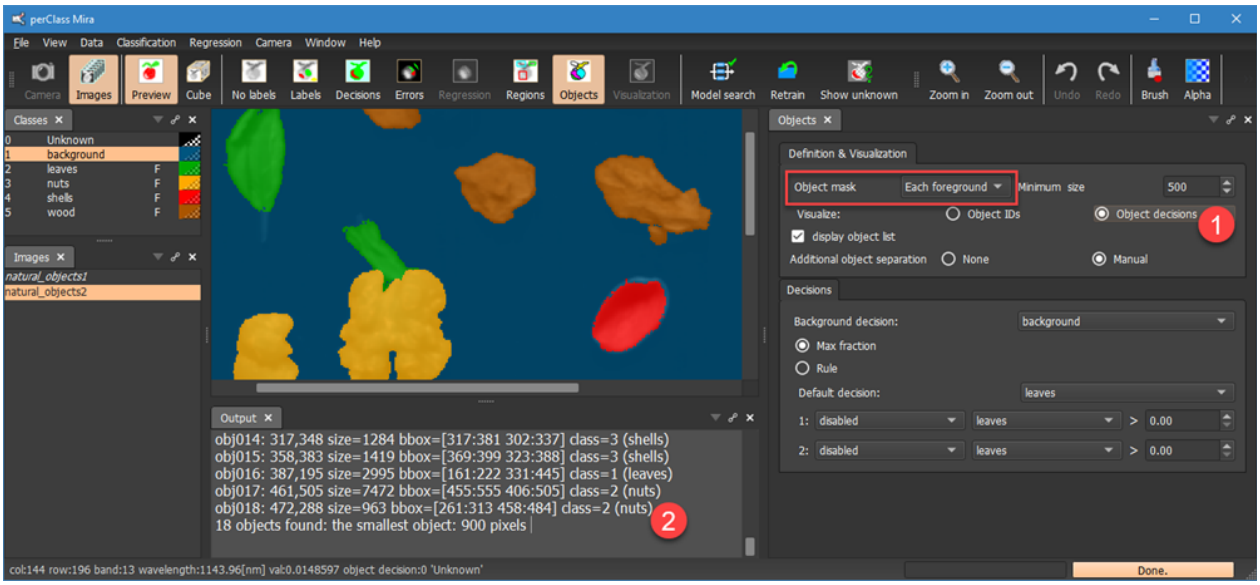


Object classification

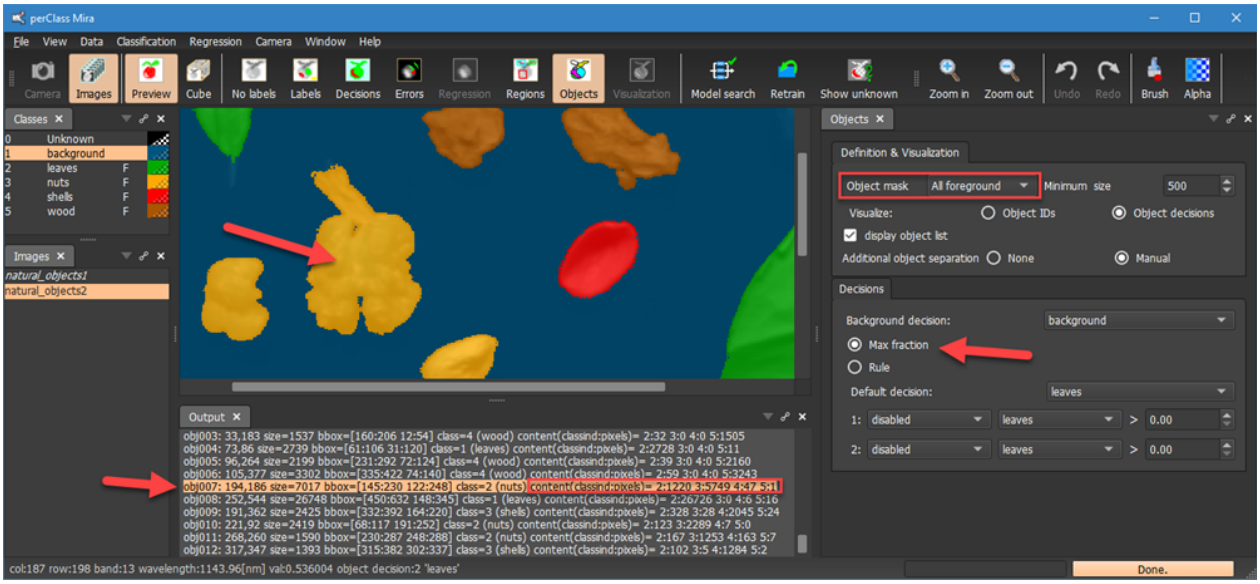
perClass Mira directly applied object classifier after object segmentation. We may view per-object decisions instead of the object IDs by selecting the *Object decisions* option 1

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

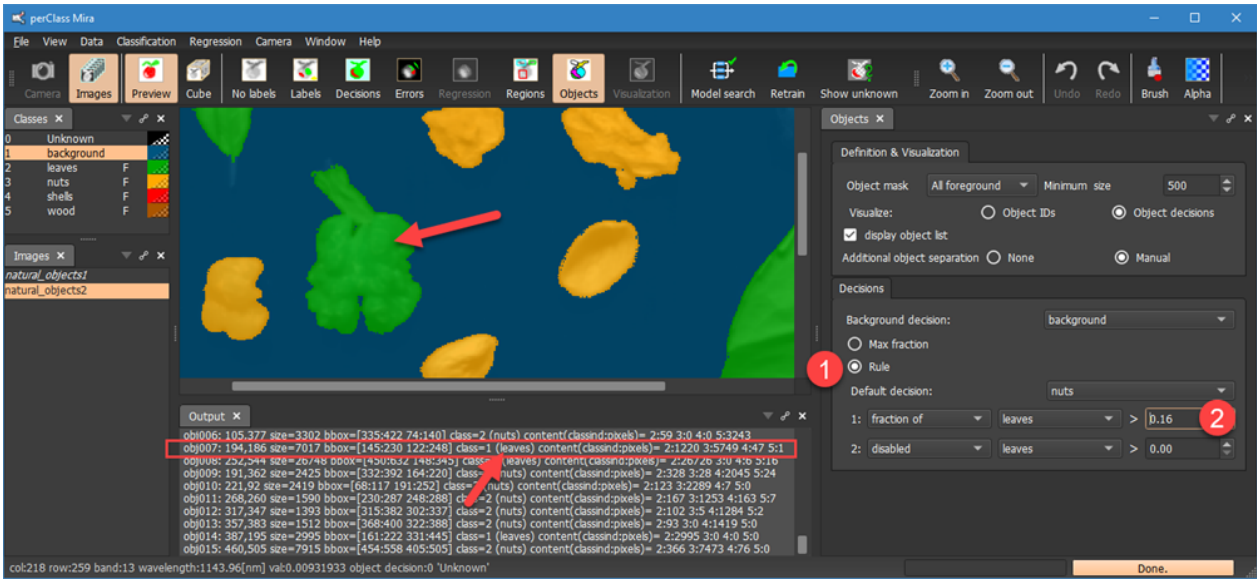
This is indicated by the class color and displayed in the object list 2,
Note, that while the object decision visualization seems very similar to pixel decisions in our example, there are many differences such as removed small objects along the edges.



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 se 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.



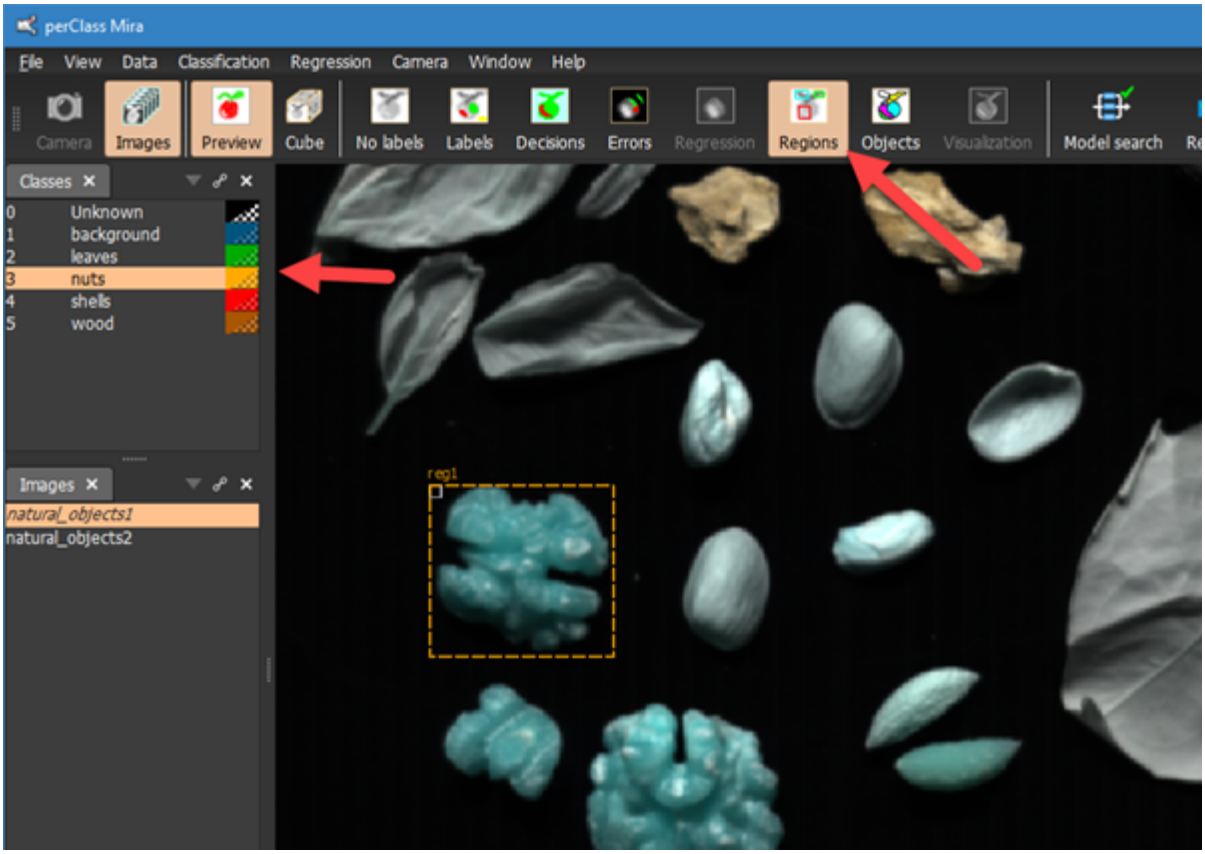
Regions

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

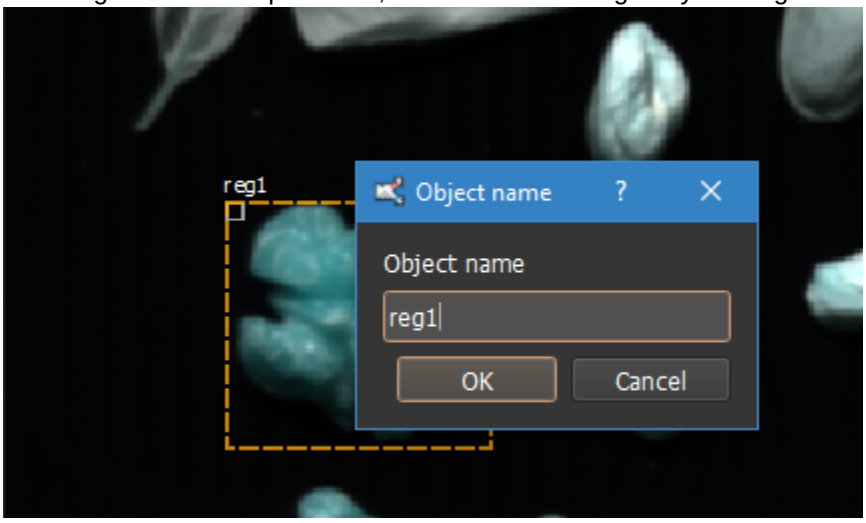
Region annotation

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**.

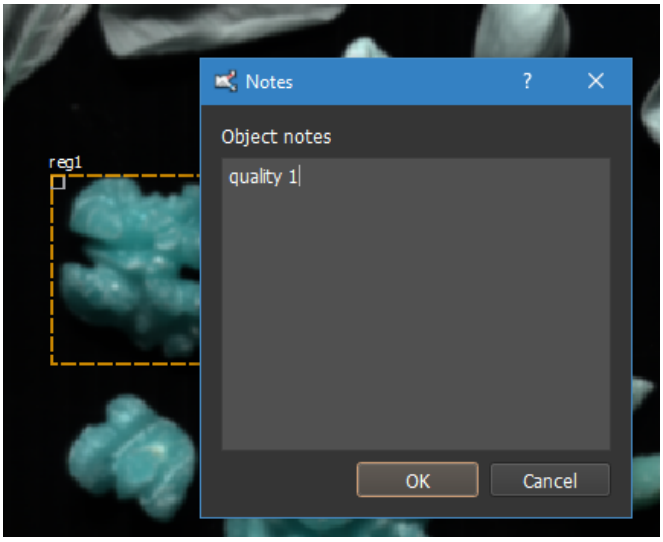
The current class, selected in the class list, is defining the color/class of the region.



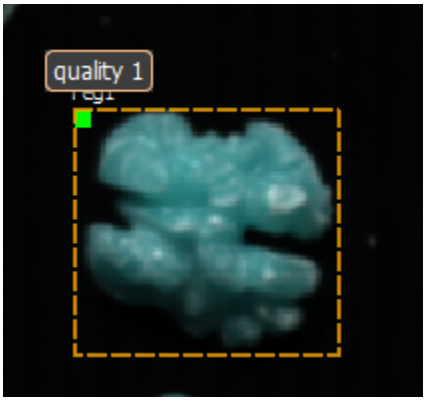
Each region has a unique name, which can be changed by clicking:



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.

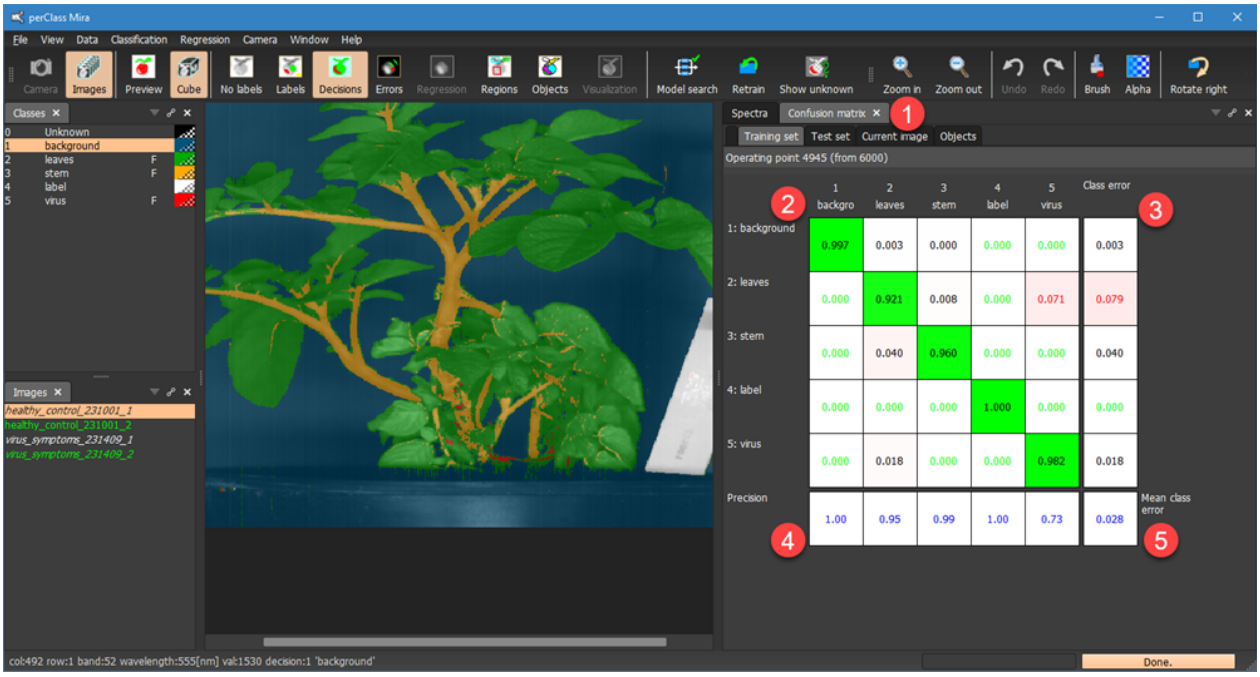
In this section, we use the potato virus data set to illustrate *Confusion matrix* tool 1 and its use in performance understanding and fine-tuning. When a pixel classifier is trained, the confusion matrix on training set is always available in the *Training set* tab of the *Confusion matrix* panel.

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 renderred in red color.

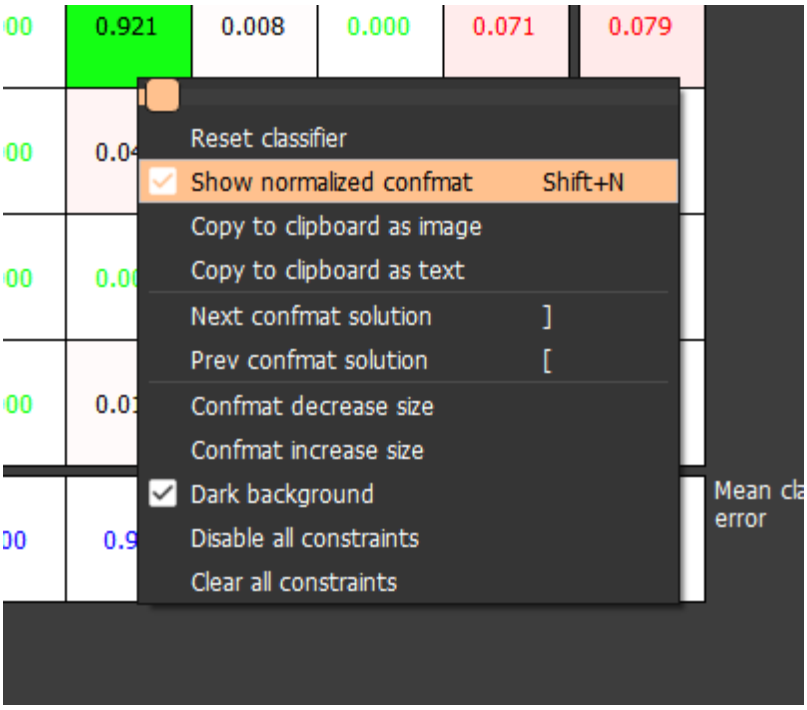
By default, the confusion matrix is normalized by each row. Thie means that the entries represent accuracies on-diagonal and error rates off-diagonal. The sum of the off-diagonal errors i.e. the **class error** is displayed in the right-most column 3 . Class error reflects what percentage of the ground truth pixels is misclassified.

Similarly, the **per-class purity** is displayed in the bottom row 4 . 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.

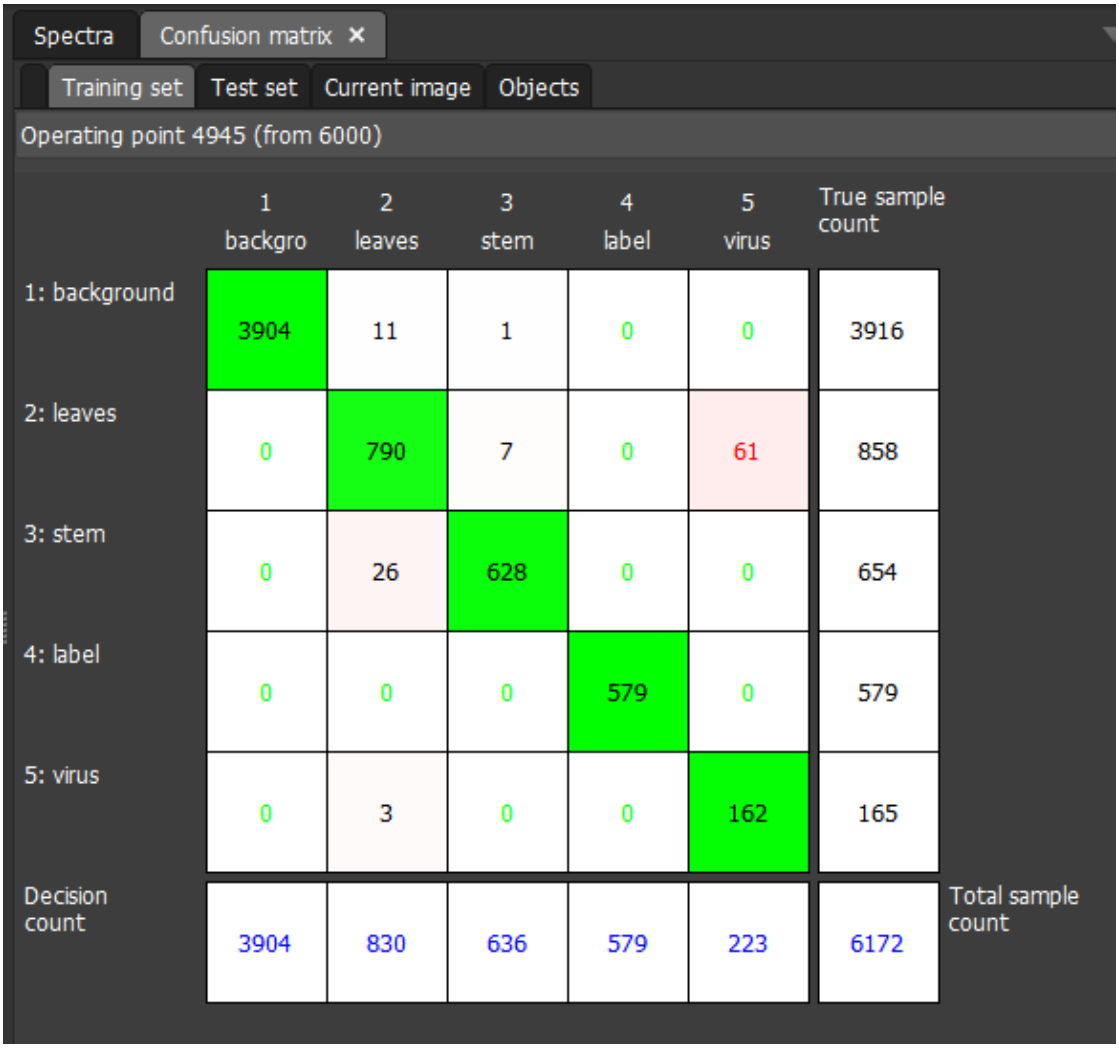
Finaly, the right-bottom corned provides one summary perfrormance indicator: The mean error over classes - the average of per-class error rates.
The value of confusion matrix is in providing detailed understanding of classifier behaviour. While 2.8% mean error does not seem too high, confusion matrix allows us to learn quickly, that 7% of healthy leaves is being misclassieid 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:

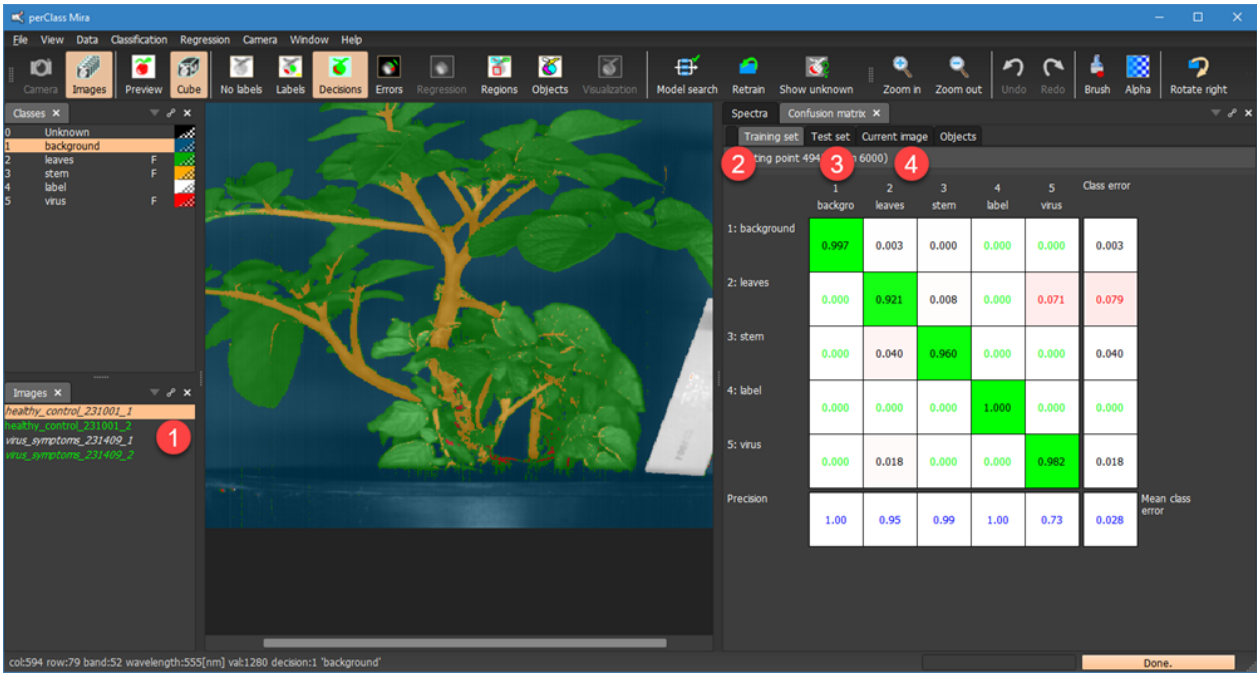


The not-normalized confusion matrix may highlight that some of our classes are undersamped. For example, while our *virus* class contains only 165 labeled pixels, the background contains almost four thousand.



Test set confusion matrix

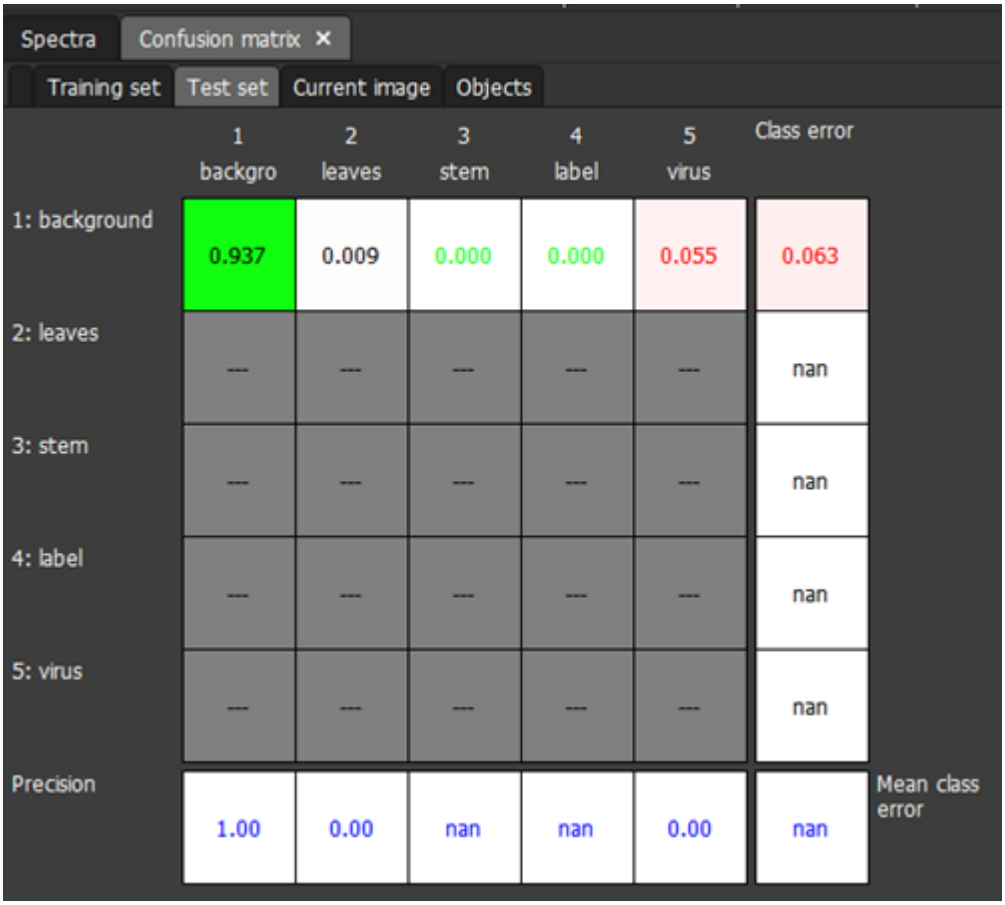
By default, the confusion martix 2 is display that is estimated from training set. This means from all images 1 that are not flagged as a test set.



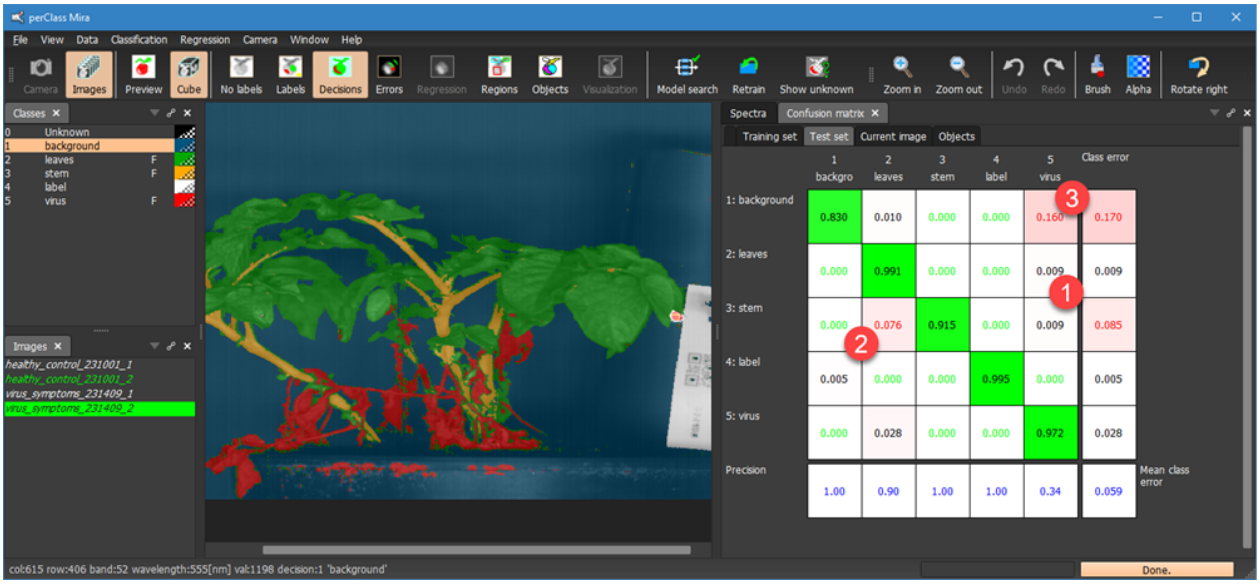
In order to properly evaluate any classification solution, we need **good performance on the independent test set**. The reason is that we need confirmation of **generalization capability** of our classifier on example unseen in training. In perClass, testing is defined on the level of images. Images 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 3. 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.



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:



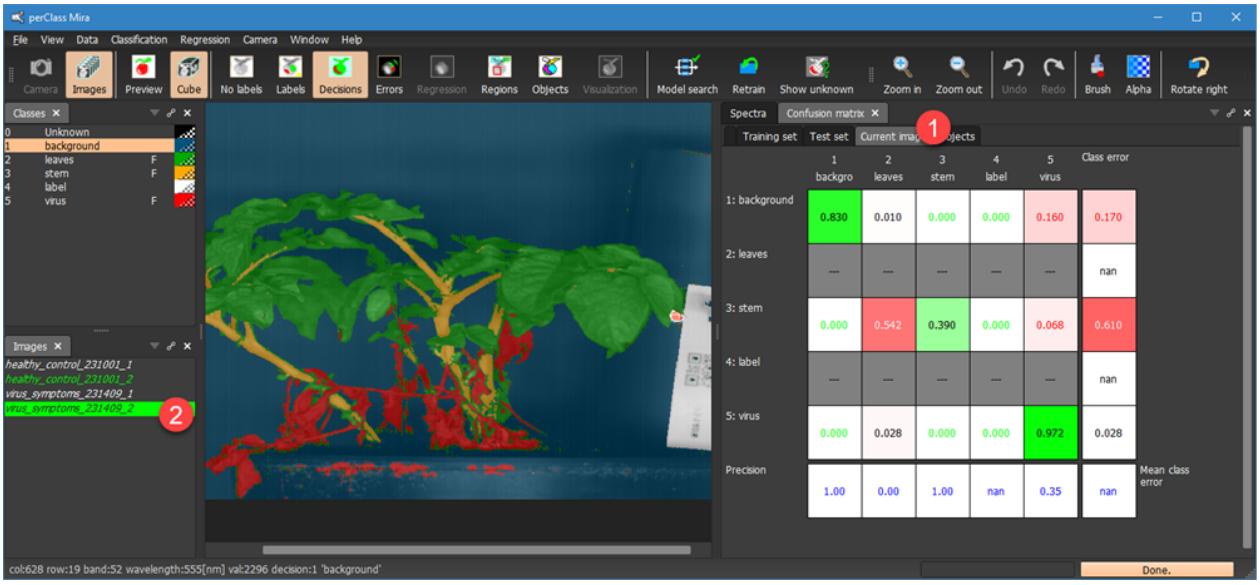
Note, that we do not observe the same misclassification of *leaves* into *virus* (1) as in the training set. In addition, we can see higher errors in two new fields, misclassifying some stem as leaves (2) and background as virus (3).

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 misclassications represent in the image. That is greatly simplified by

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

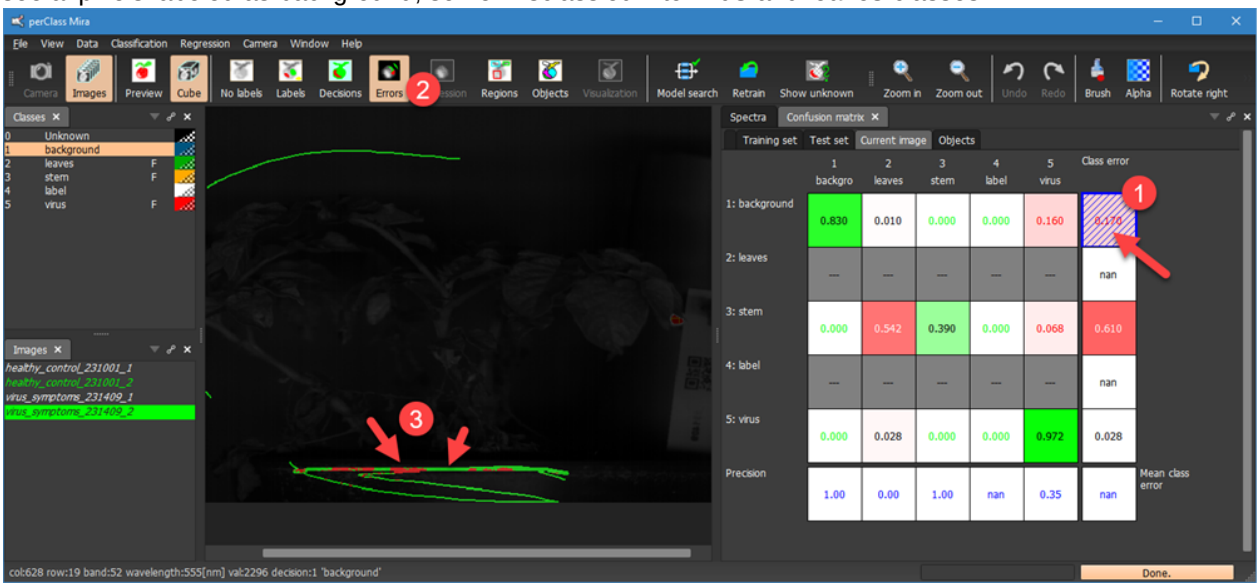
Current image confusion matrix

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

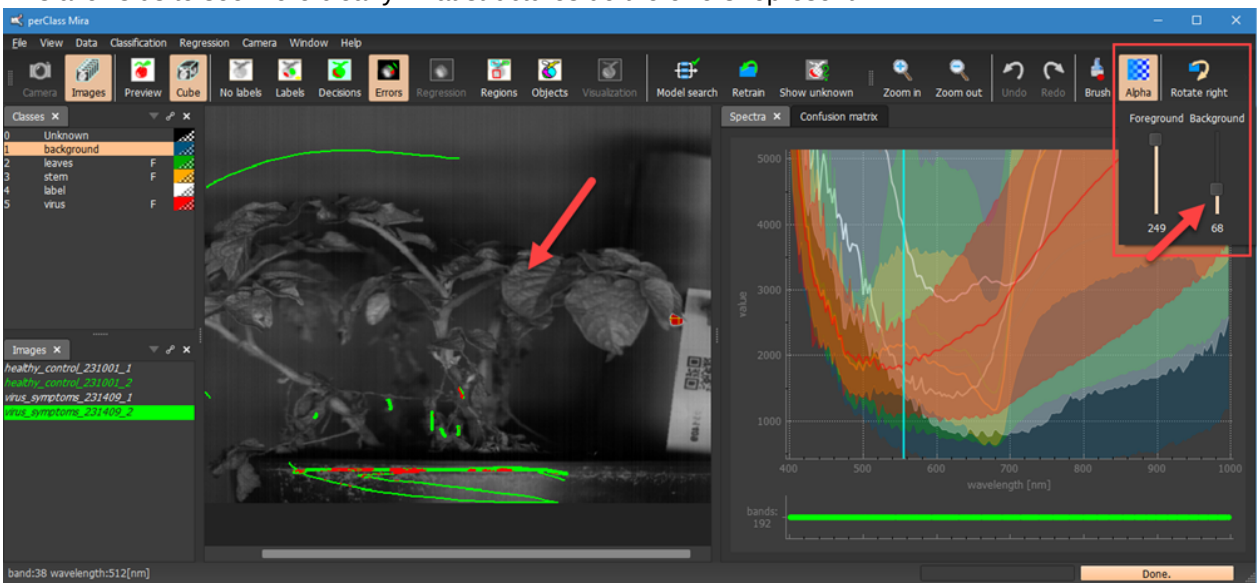


The *Current image* confusion matrix is fully interactive. When we hover over the confusion matrix entries, errors at the pixel level are visualized over the image.

For example, moving the mouse over the *background* class error (1), The *Errors* mode (2) 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 misclassied pixels in red. In our example, we can see al pixels labeled as background, some misclassied 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.

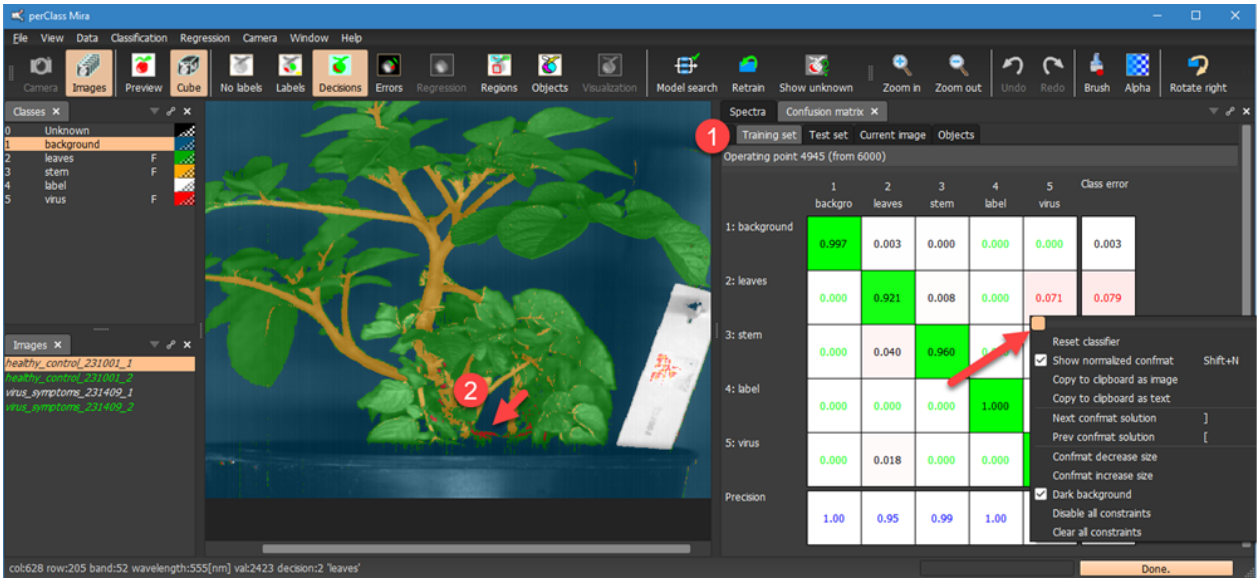
Tuning the classifier perfromance to application-specific requirements is a basic necessity in any practically deployed machine learning system. It is because default way that statistical models make decisions largely depend on the class abundancy in the training set which typically does not correspond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

perClass Mira provides two ways to tune classifier performance in the confusion matrix:
1. Cost sensitive optimization
2. Performance constraints

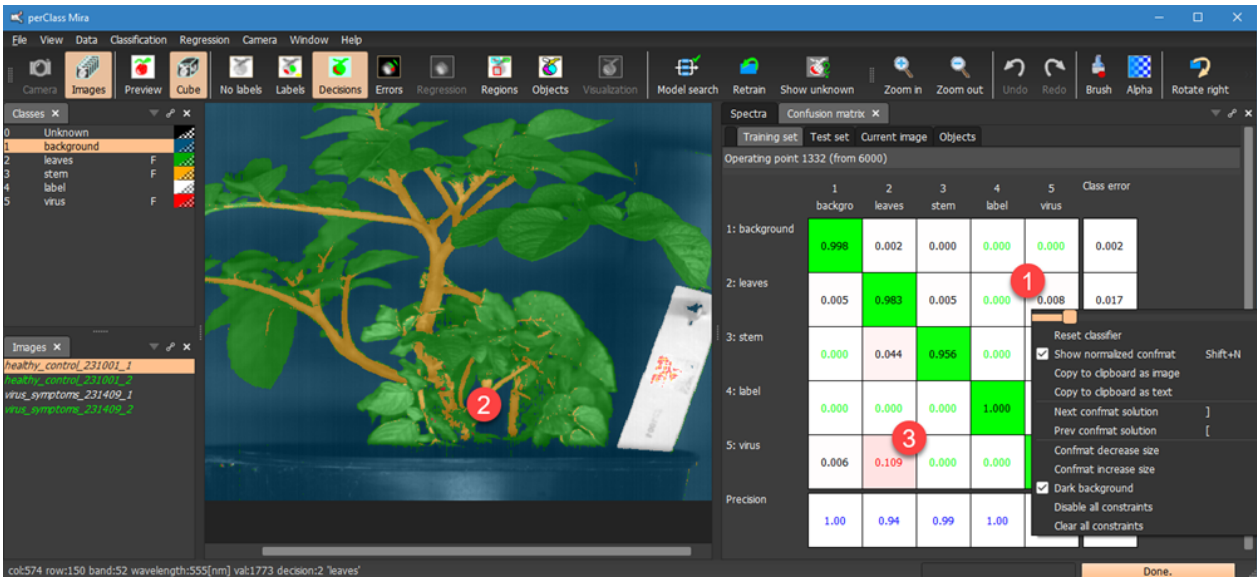
Note, that all classifier optimization in perClass Mira happens on the training set, NOT the test set. This is very important point. Test set, in perClass Mira, is considered only for performance evaluation, not for any form of model tuning.

Cost sensitive optimization

In the *Training set* confusion matrix (1), 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 (2) The slider in the context menu allows us to increase the cost for this entry:



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

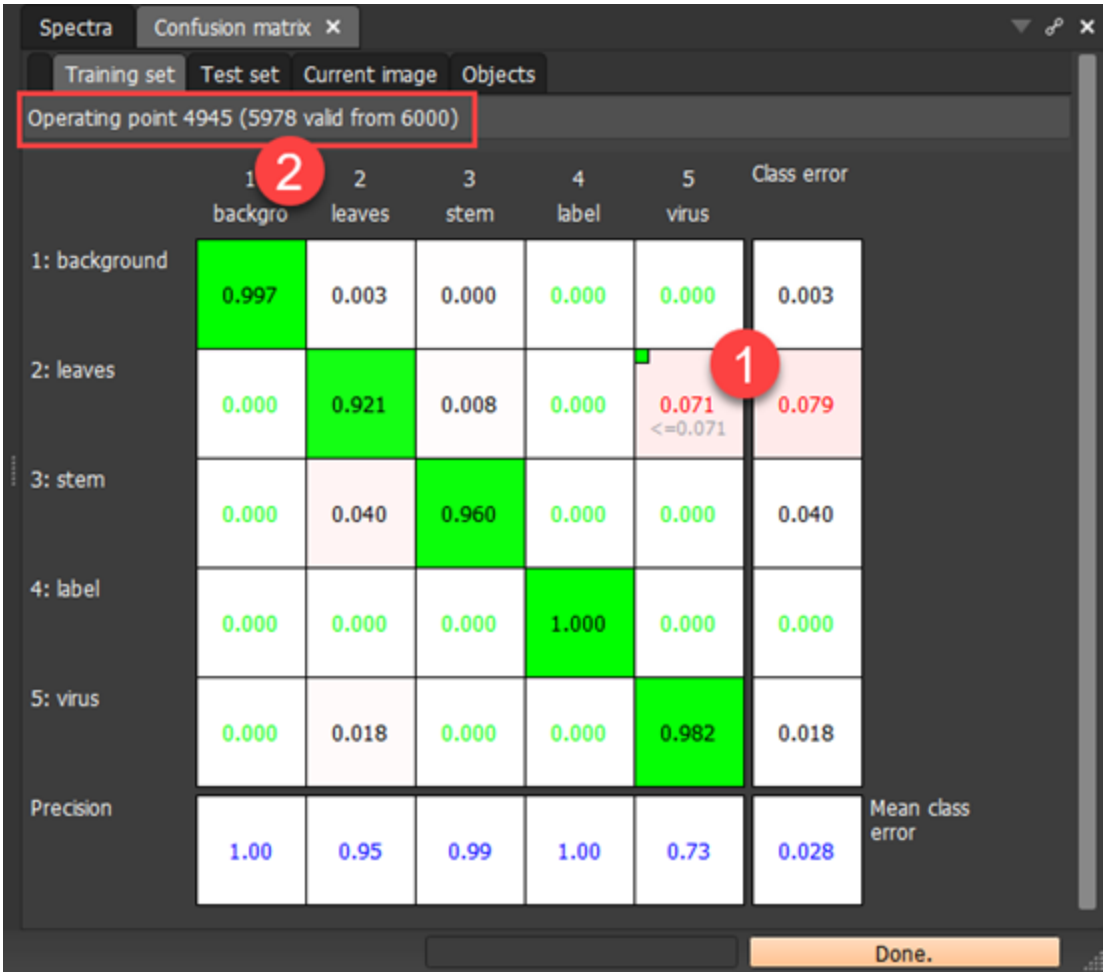


We cannot see these piels on the current image of the healthy (control) plant as it does not contain any virus infection. We can switch to training image with virus symptoms to investigate impact of this adjustment on true virus class.

Performance contraints

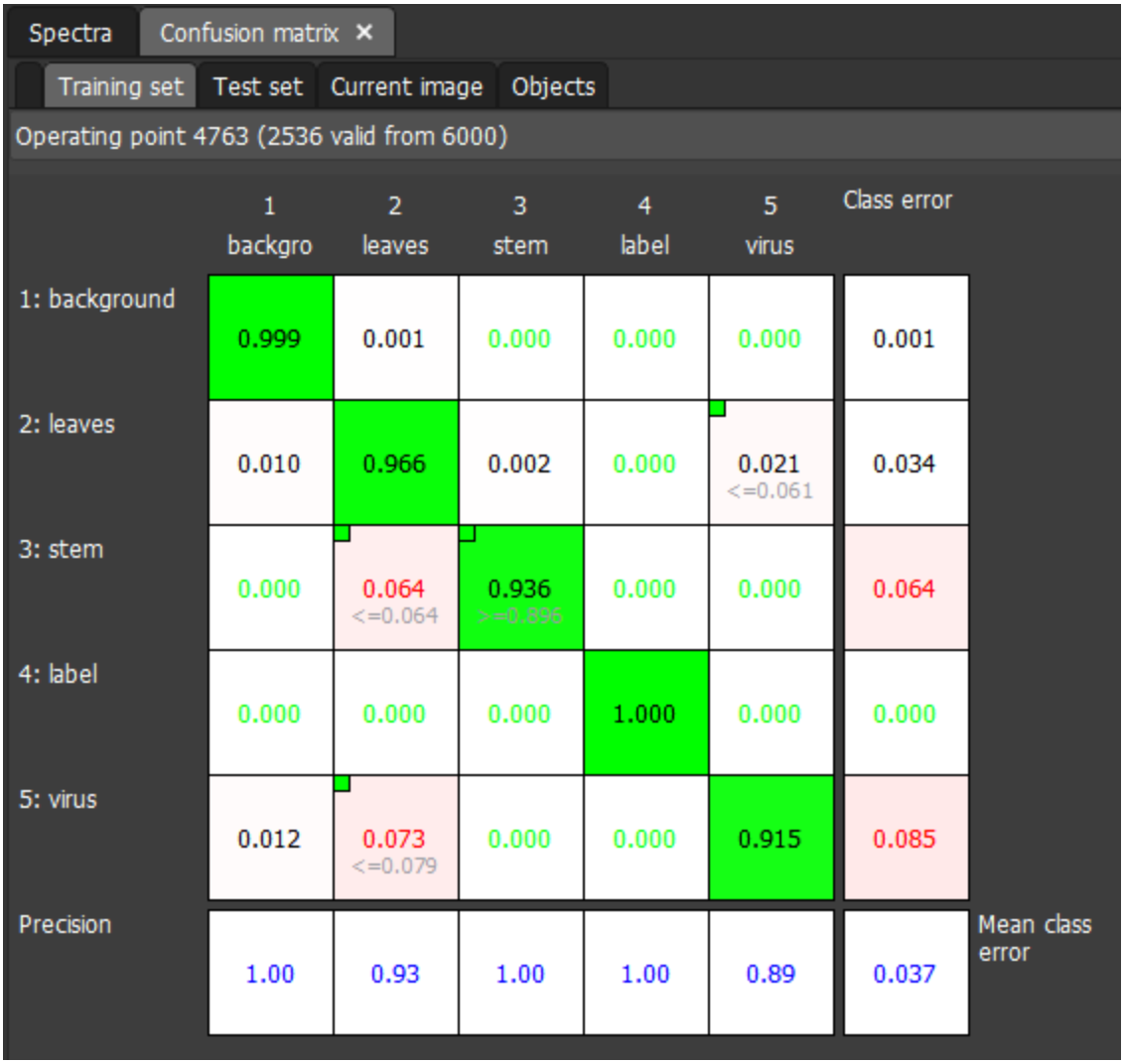
The alternative way of fine-tuning the classifier performance is by defining performance contraints.This means that we limit the error or accuracy on certain field or fields. We can do that by double clicking any field of the matrix.

In our example, we double click on the *leaves* misclassified as *virus* 1. 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 2 is decreased by each new contrain.

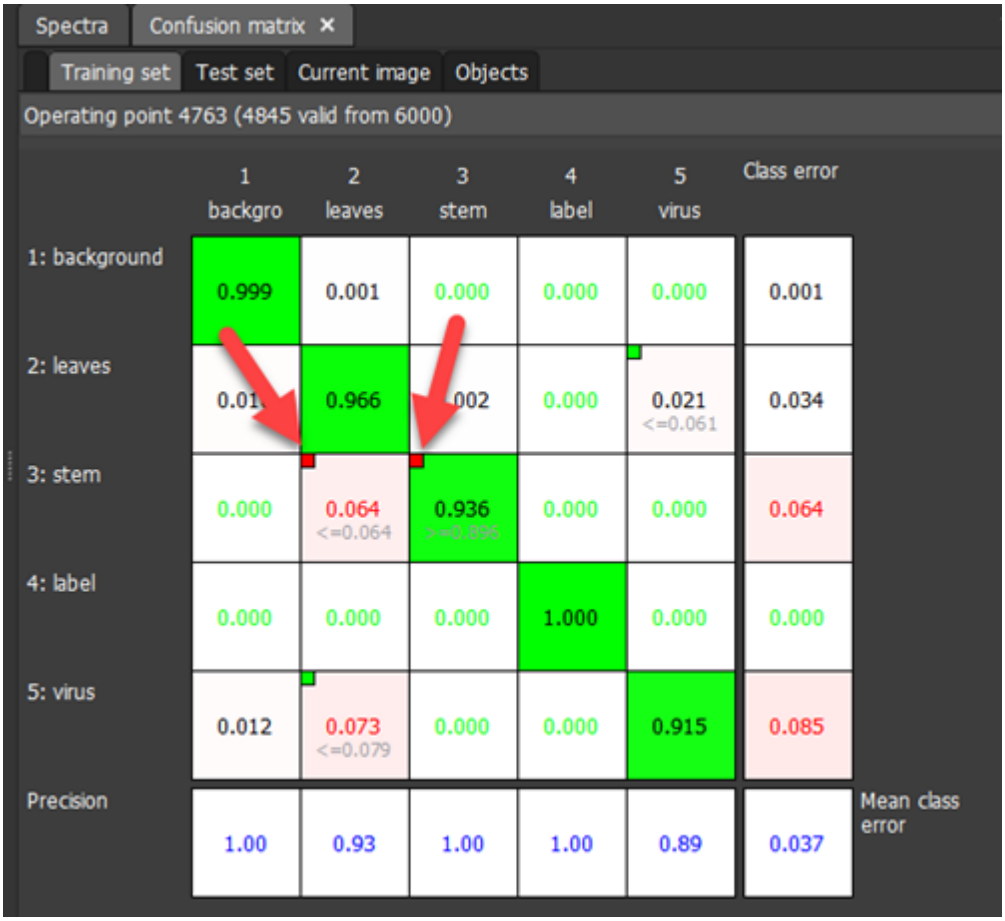


In order to tune the constrain, move mouse over the constrained field, hold Ctrl and use mouse wheel to adjust the constrain. The new solution is displayed.

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



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

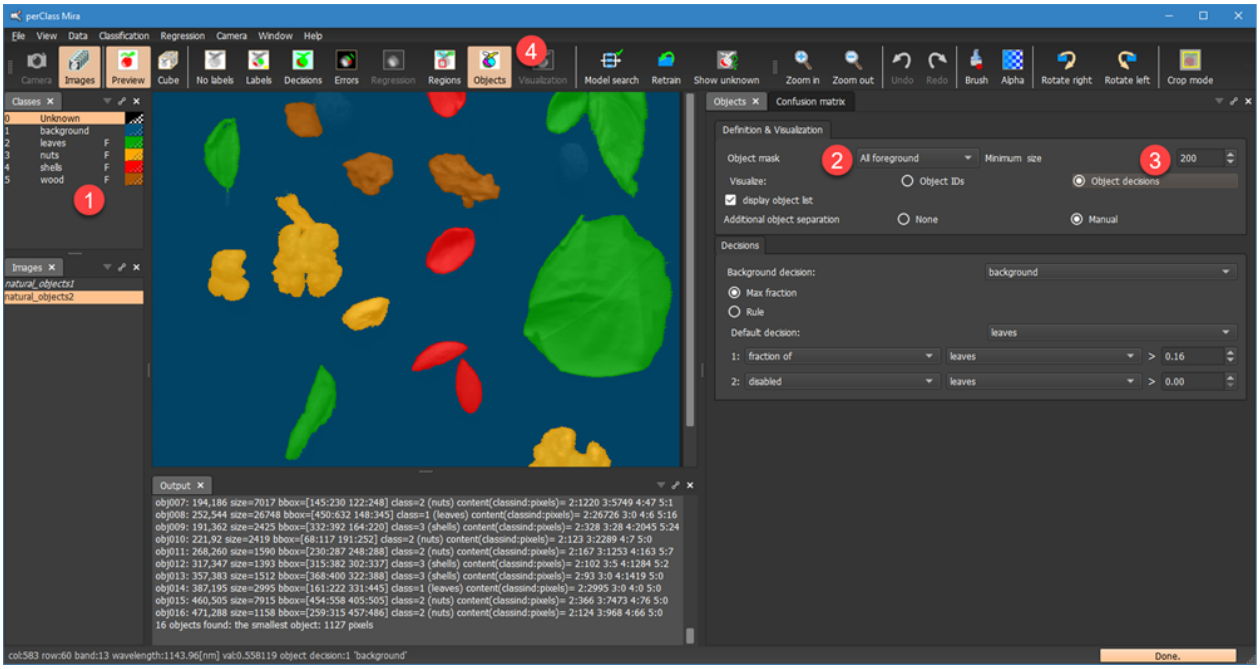


Object confusion matrix

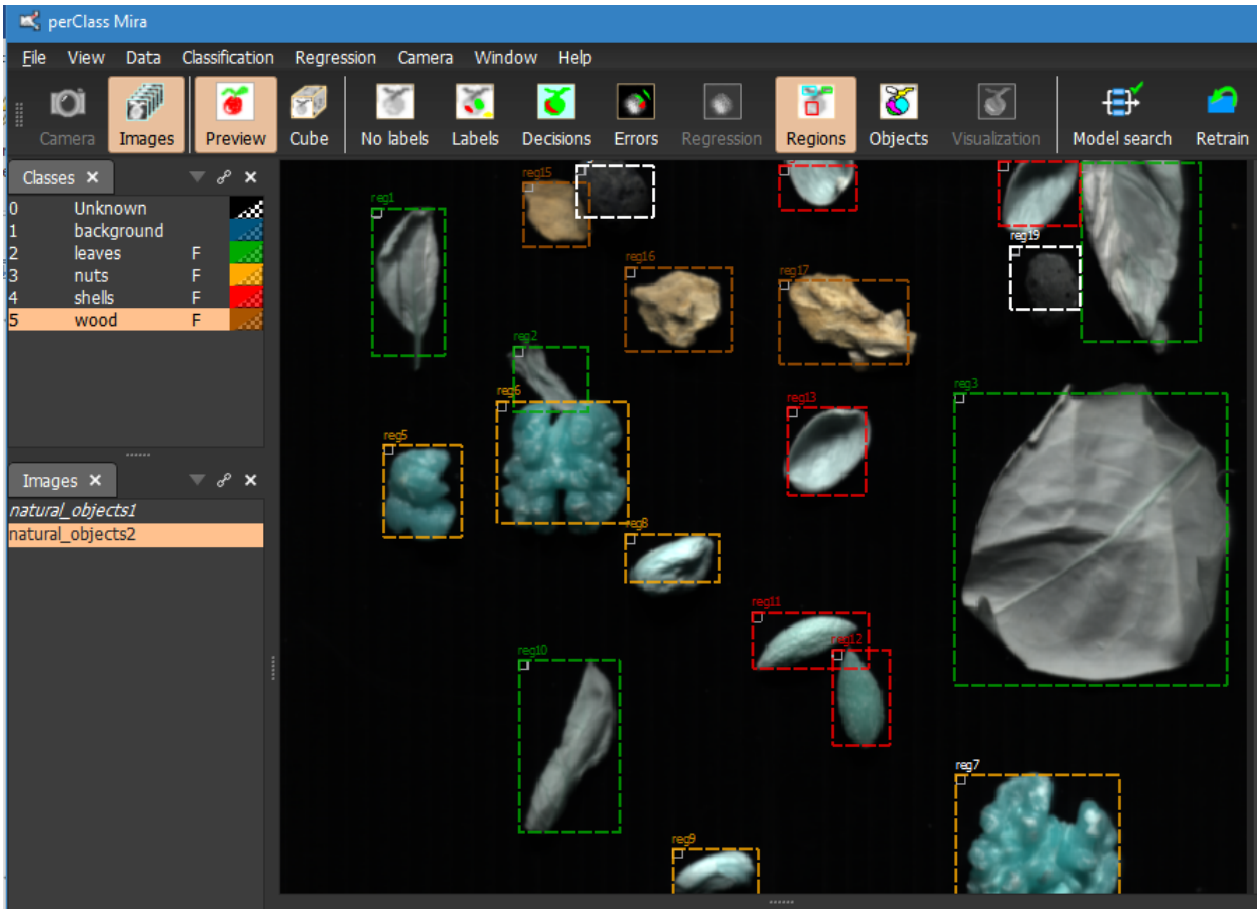
In the same way pixel confusion matrix allows us to understand performance of pixel classifier, perClass Mira provides object confusion matrix to characterize object classification performance.

In order to estimate object confusion matrix, we need
1. **object classifier** and
2. **object ground truth**, defined by image regions.

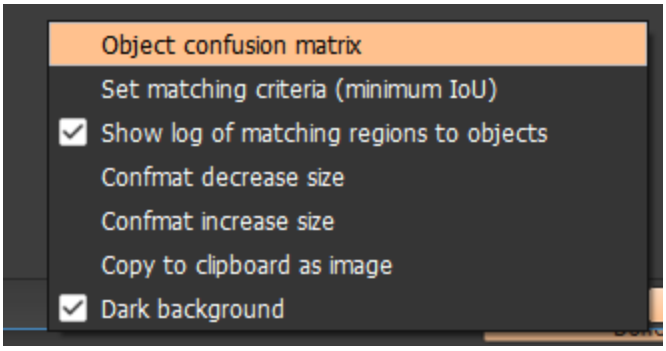
In our example, we built a pixel classifier and flagged several classes as foreground 1. We use the *All foreground* mode 2 and minimum size of 200 pixels 3 By clicking *Objects* toolbar button 4, we perform object segmentation followed by object classifier:



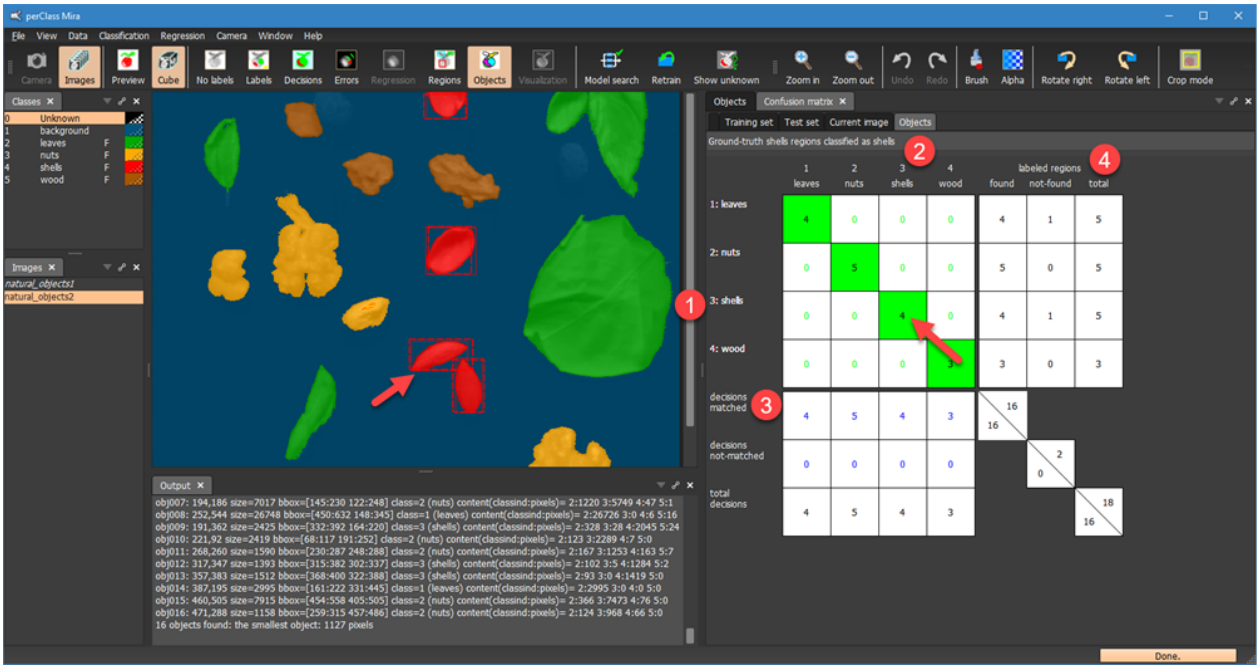
In order to define object ground truth, we use the *Regions* tool an annotate individual objects assigning them to their respective classes:



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 1 and classifier decisions in columns 2 .



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.

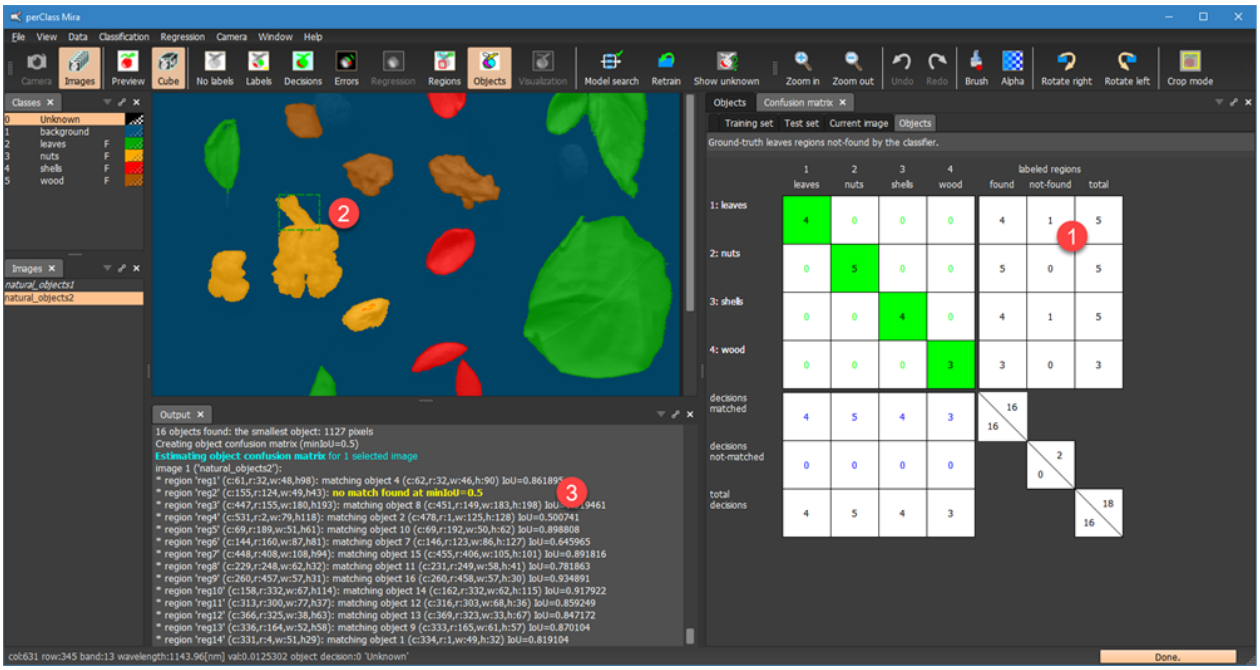
Detailed information on object matching

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** 3 and **all labeled regions** 4 .

The decisions section 3 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 4 provides extra insight on labeled regions that were not identified by the object classifier.

Example of interactive inspection

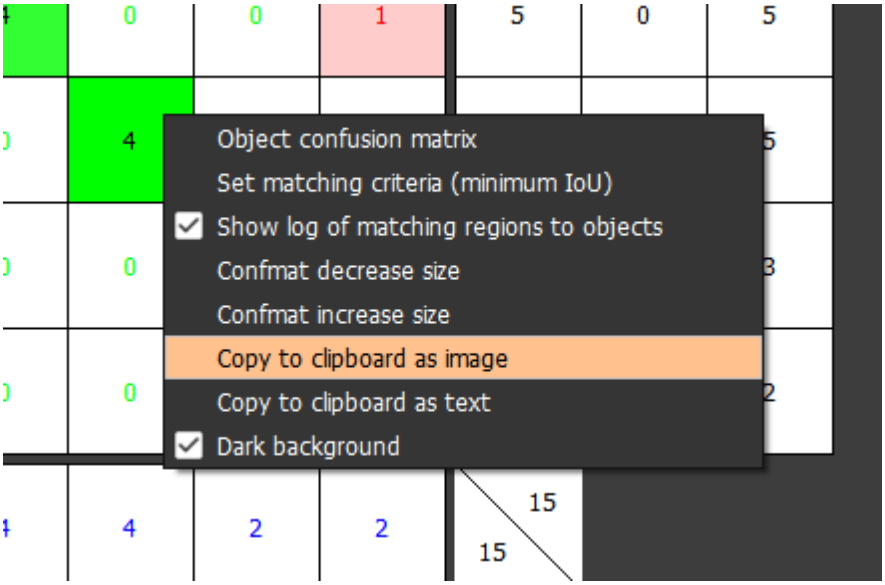
In the example below, we hover with the mouse pointer on the field 1. This highlights single region from *leaves* class that was not found 2. The *Output* window shows a log of all matches between the ground truth regions and object bounding boxes. We can see that the **reg2** region could not be matched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



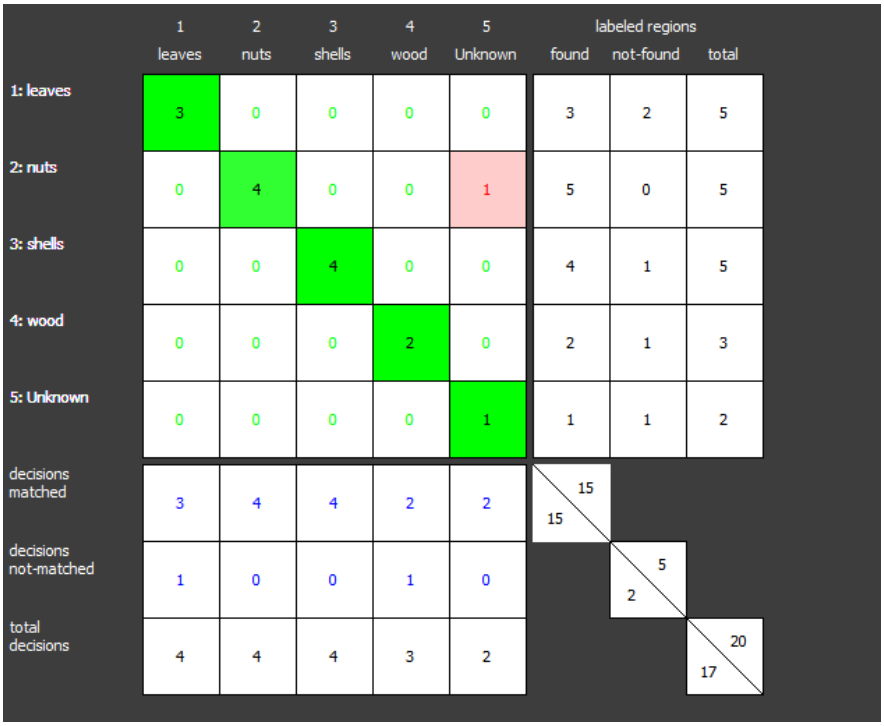
Copying confusion matrix

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:

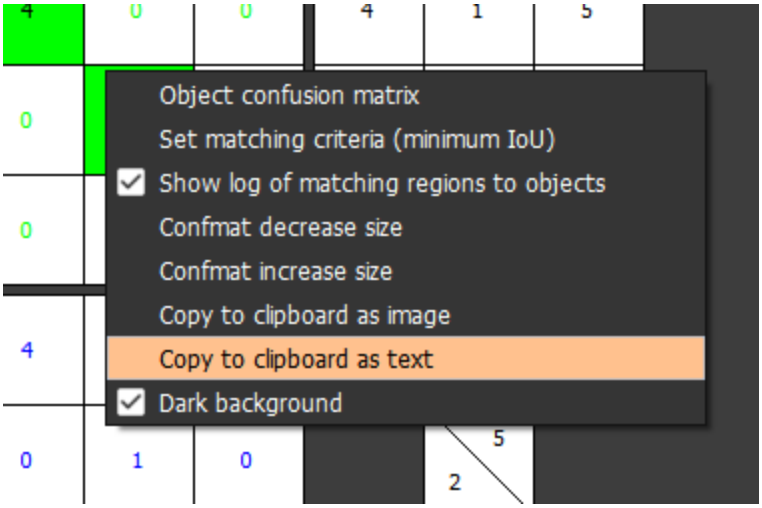


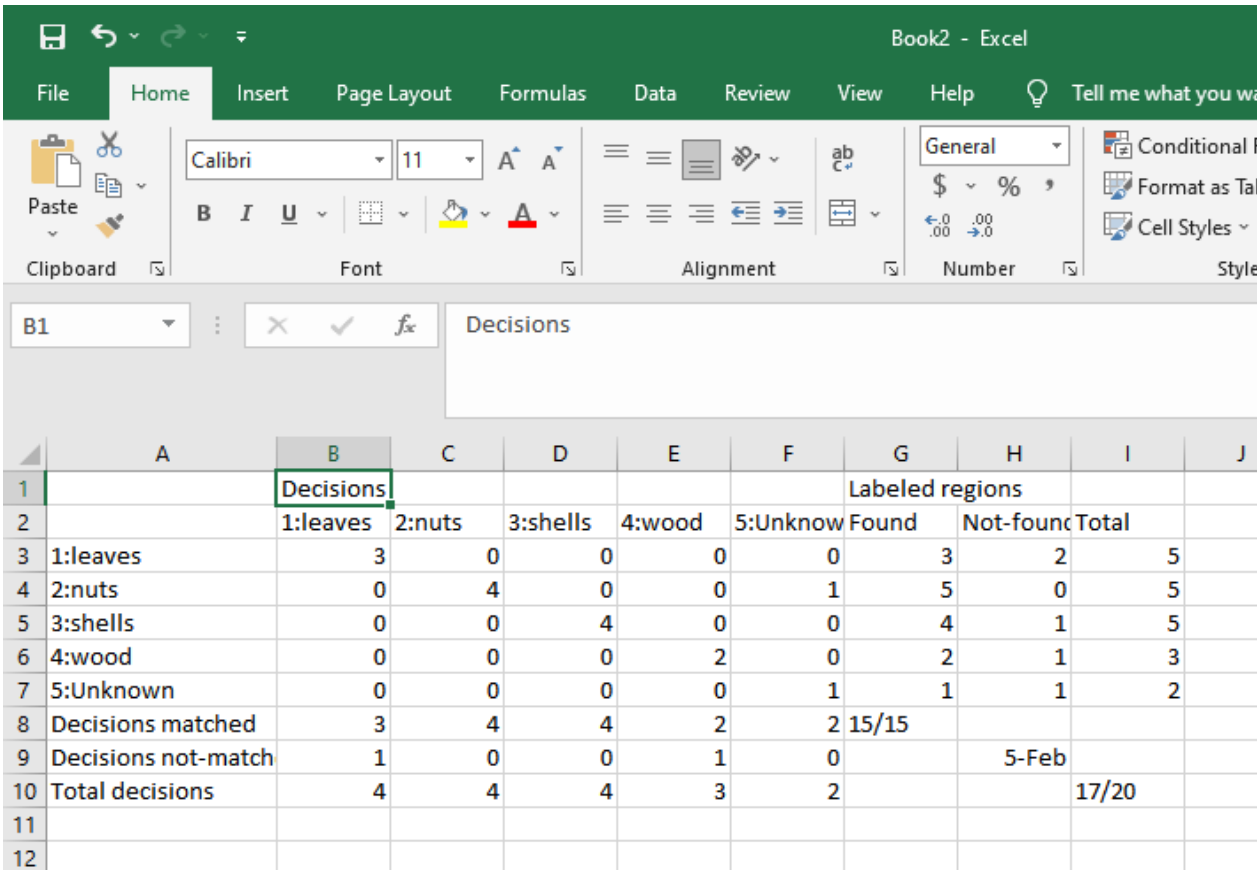
	1 leaves	2 nuts	3 shells	4 wood	5 Unknown	labeled regions found	not-found	total
1: leaves	3	0	0	0	0	3	2	5
2: nuts	0	4	0	0	1	5	0	5
3: shells	0	0	4	0	0	4	1	5
4: wood	0	0	0	2	0	2	1	3
5: Unknown	0	0	0	0	1	1	1	2
decisions matched	3	4	4	2	2	15		
decisions not-matched	1	0	0	1	0		5	
total decisions	4	4	4	3	2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

	1 leaves	2 nuts	3 shells	4 wood	5 Unknown	labeled regions 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			20
						15	2	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.





	A	B	C	D	E	F	G	H	I	J
1		Decisions					Labeled regions			
2		1:leaves	2:nuts	3:shells	4:wood	5:Unknow	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										

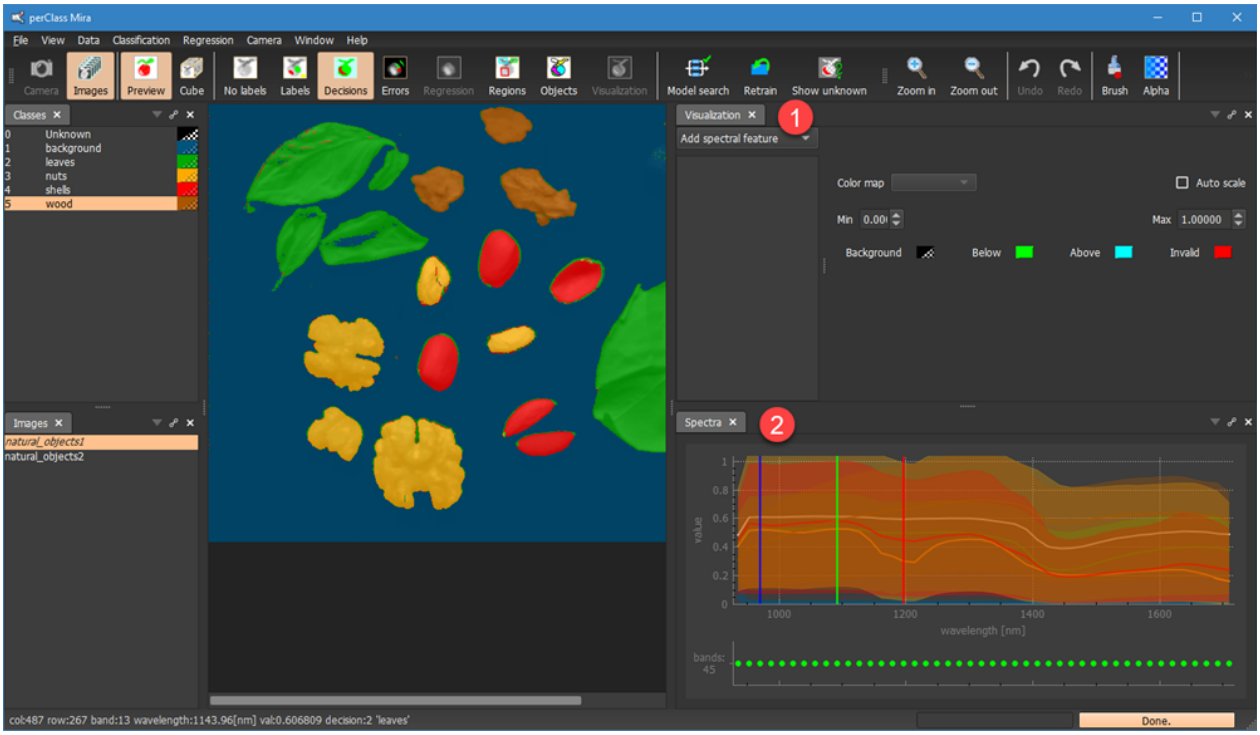
Visualization (spectral indices)

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.

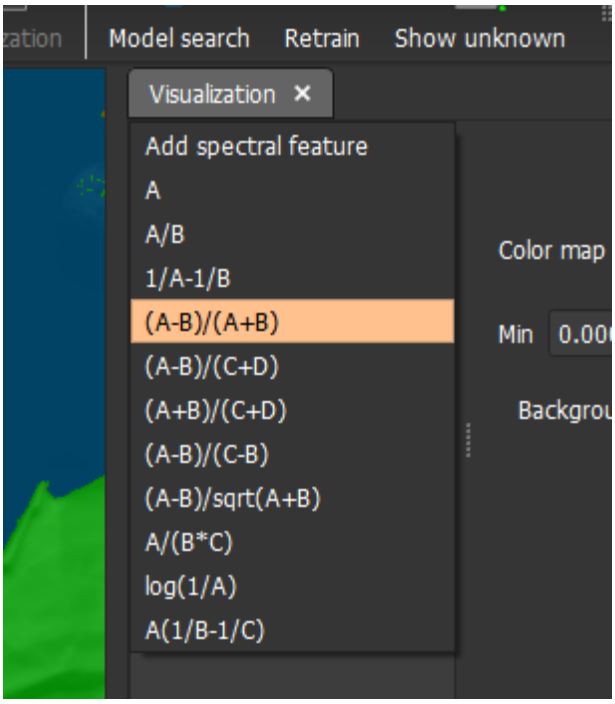
For example, one commonly-used spectral index is NDVI which, in broad terms, highlights a difference between visible and near-infrared reflectance. It is used in remote sensing in order to estimate vegetation coverage NDVI = (NIR - RED) / (NIR + RED). In the NDVI equation, the RED and NIR terms corresponds 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 (1) above the *Spectra* (2).

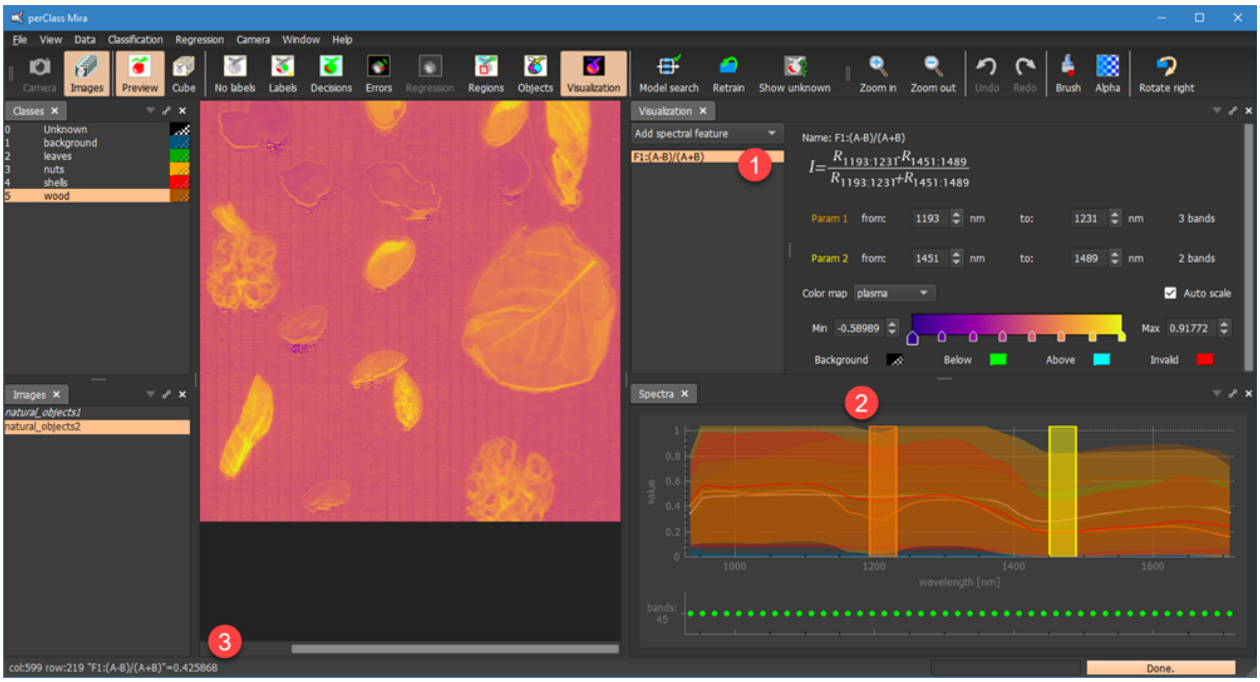


We may select the type of spectral feature (generic equation) in the combo box located in the *Visualization* panel.



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 (1) in the list box. The spectral index is then computed for every pixel fo 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 notaion) means integrated reflectance between 1451 and 1489 nm. Each of the spectral feature

parametes (in our case A and B) are also highlighted by colored bars in the spectral plot 2 . 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 3 .



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.

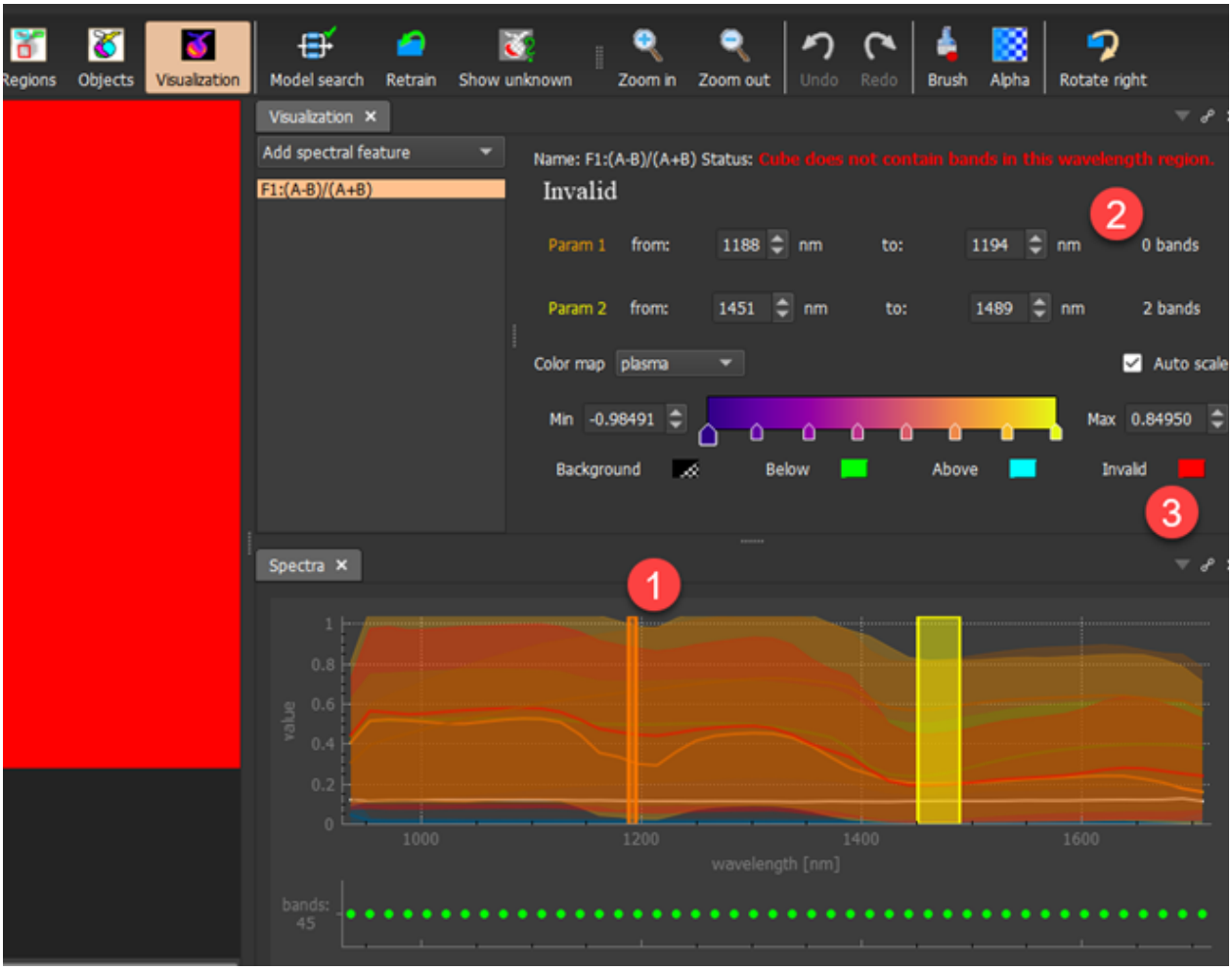
TIP: Hold Ctrl to always drag the bar. This is useful if the bar is narrow and simple click-and-drag would result in change of the bar boundaries

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 1 .

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

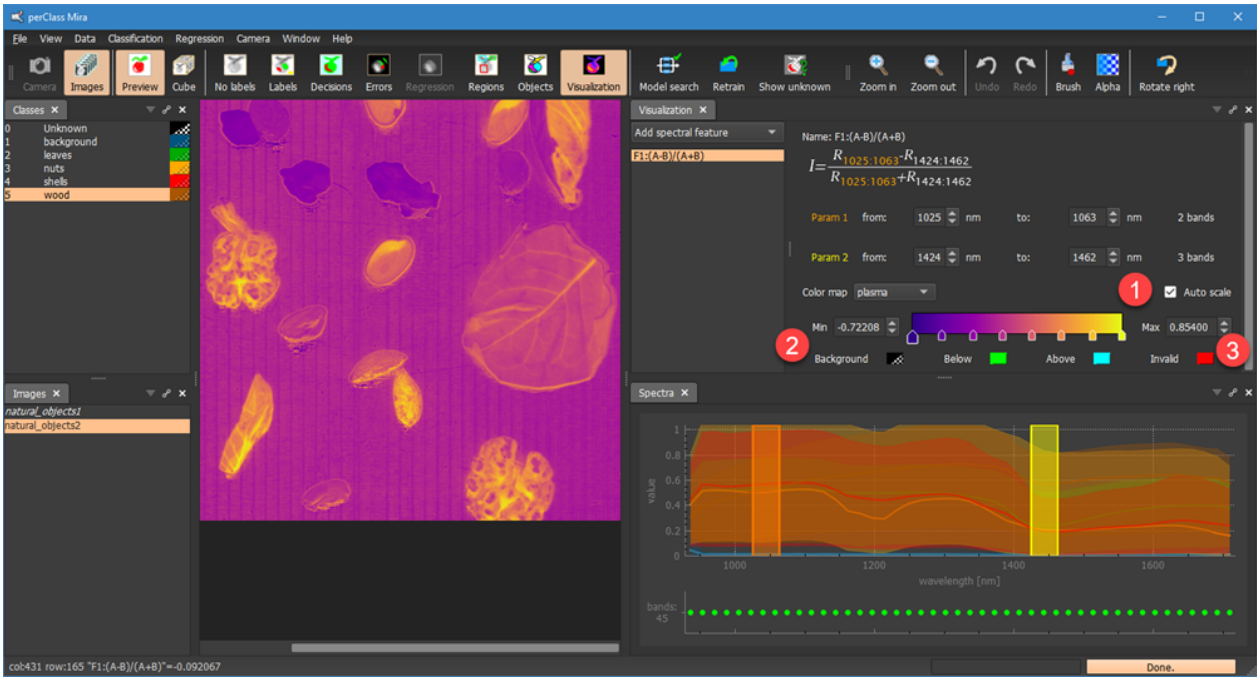
In the following example, our wavelength range for parmater A 1 does not cover any existing spectral bands in our image. This is indicated by zero bands 2 and a warning in red. The entire image is then also displayed in red - the default "invalid" color 3 .



Scaling spectral features

Auto-scaling

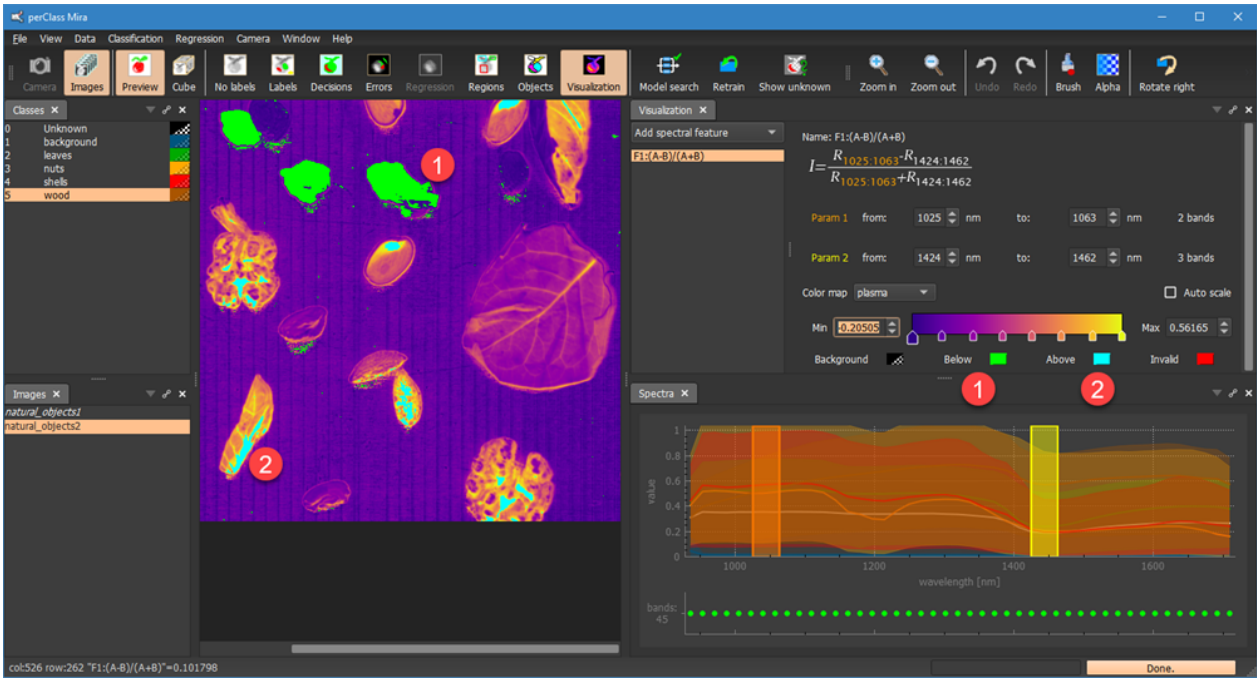
By default, visualization of spectral features is auto-scaled. This is indicated by the *Auto scale* checkbox 1



When switching between different images, the minimum 2 and maximum 3 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.

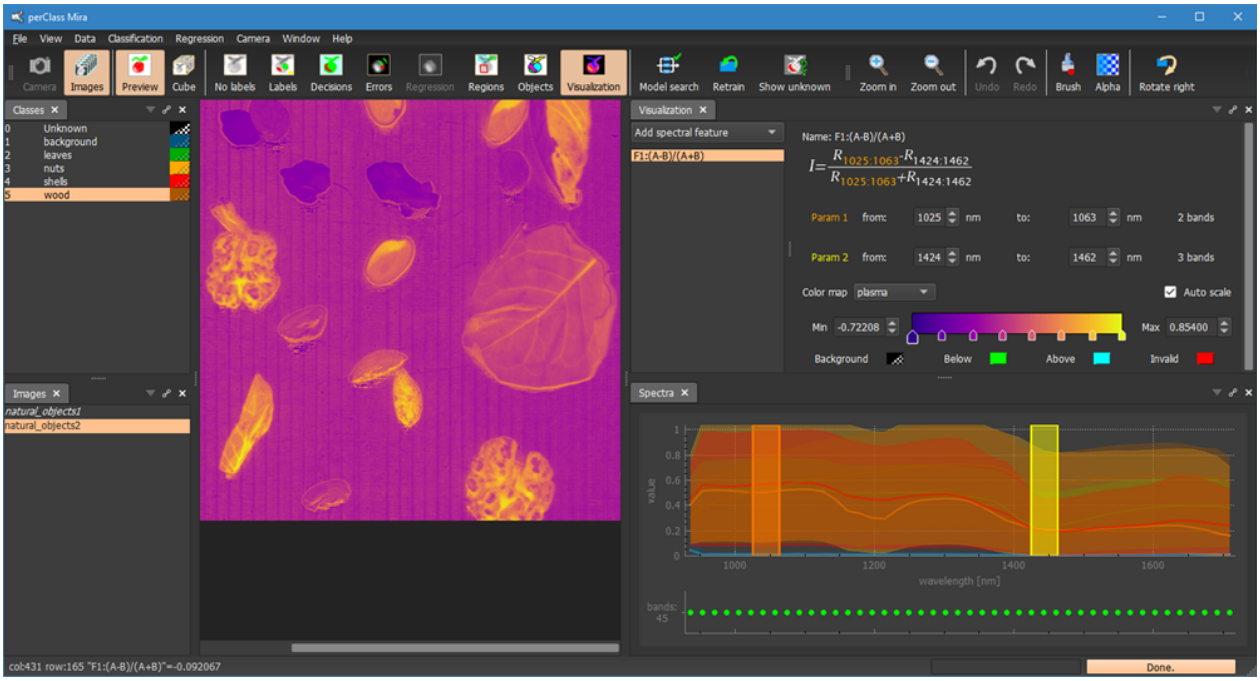
Manual scaling

When adjusting minimum and maximum range values manually, we may notice that some pixels are displayed in green 1 or cyan 2 . These correspond to pixels below or above the current visualization range. You may adjust the colors by the respective color swatches.

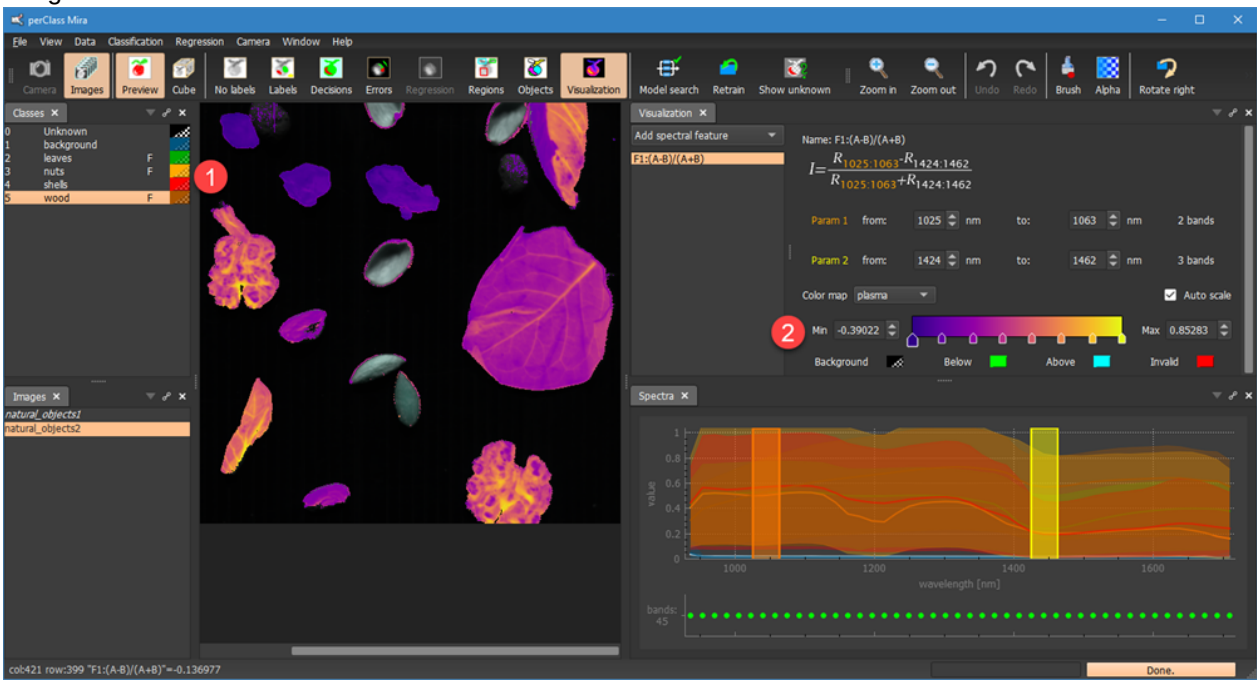


Applying feature extraction to foreground

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



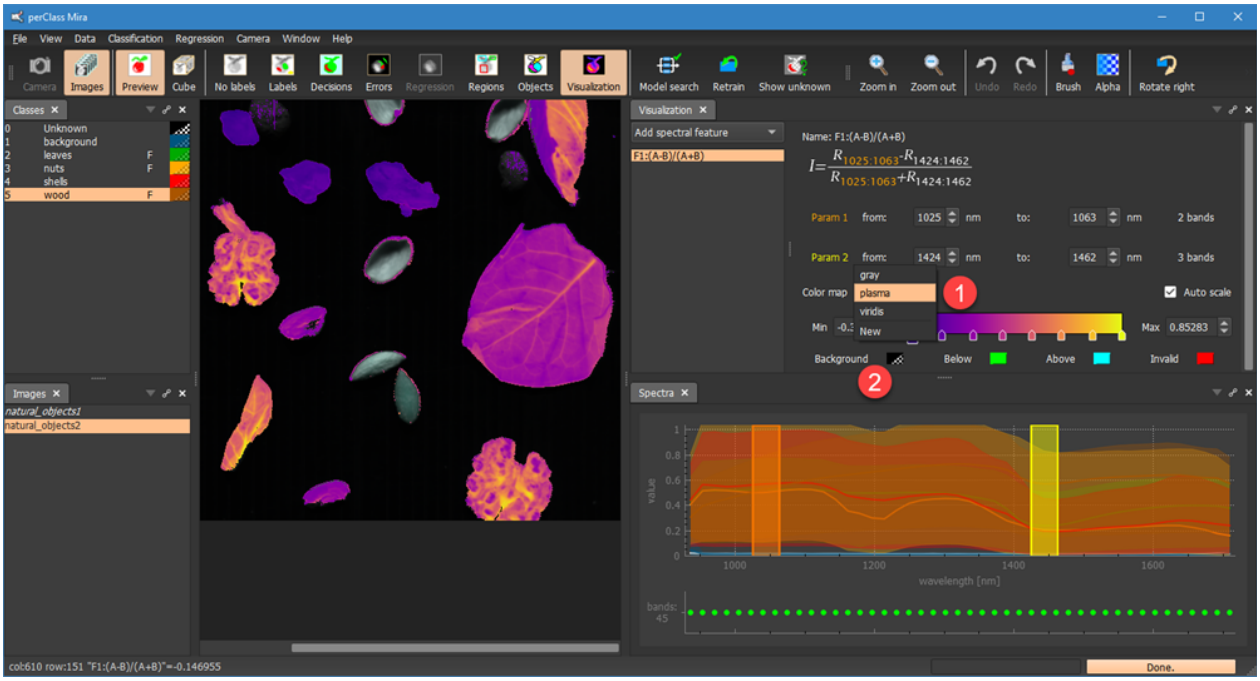
If a pixel classifier is defined, we may focus the analysis only to specific classes by selecting these as foreground (1):



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

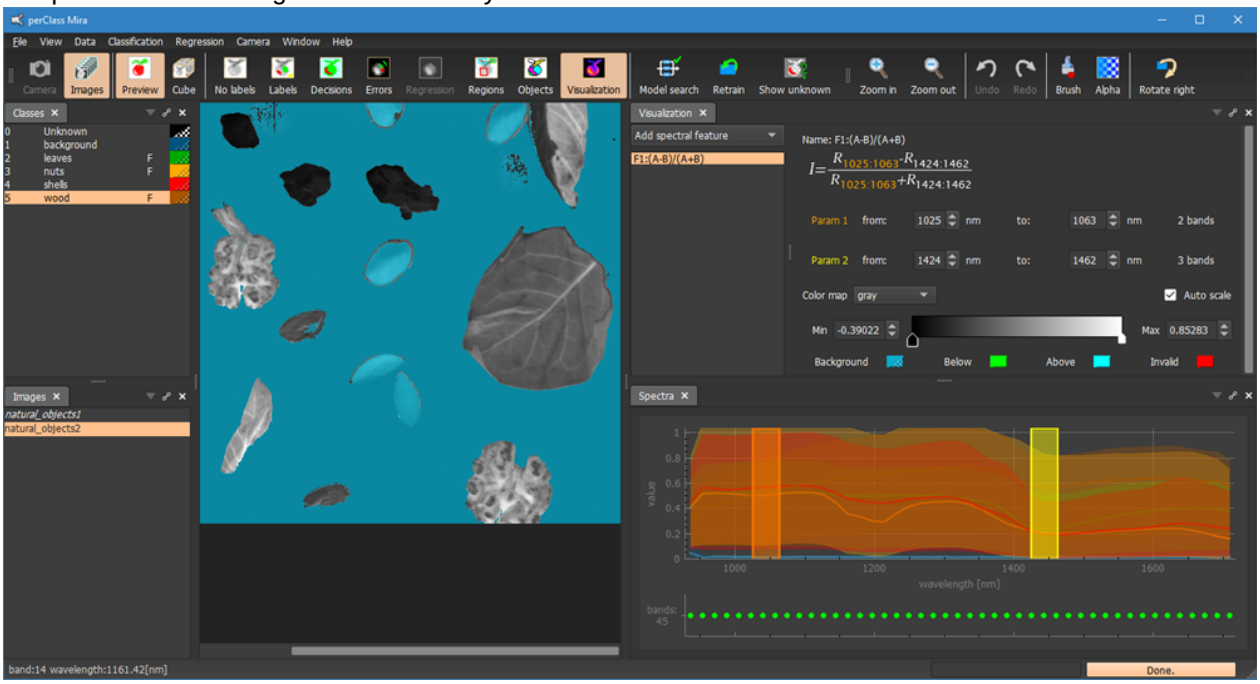
The color map of spectral feature visualization can be selected using the combo box (1)



We may choose from several predefined colormaps or create a custom colormap definition by selecting *New.*

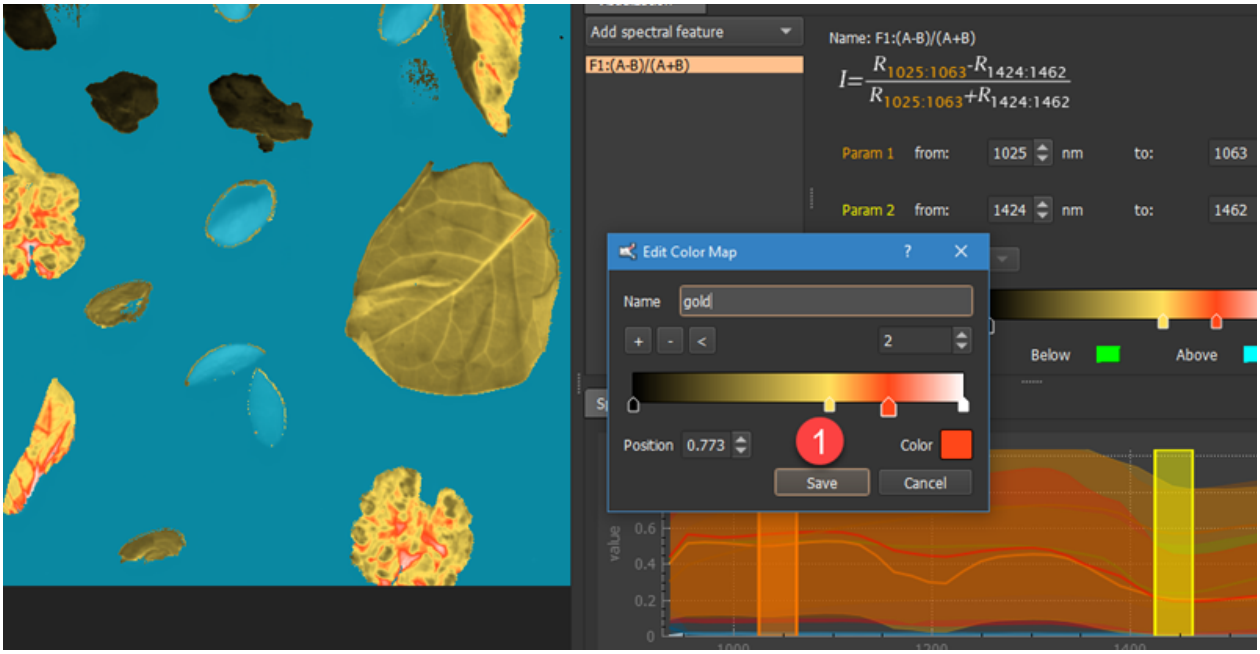
For some colormaps, it may be useful to set the background color to avoid confusion. We may do that using the *Background* color swatch.

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.

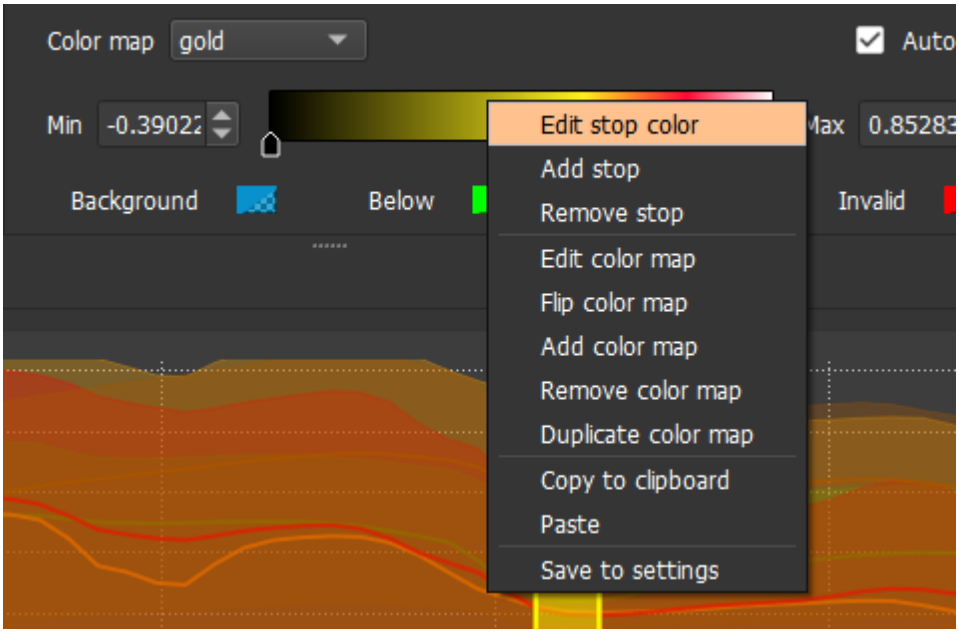


TIP: You may adjust transparency of the background color in the color dialog using the *Alpha channel* field. 255 denotes opaque and 0 fully transparent layer.

Example of custom color map definition: Select *New* in *Color map* combo box. Define the colormap steps and colors interactively 1 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 cane 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;"
```

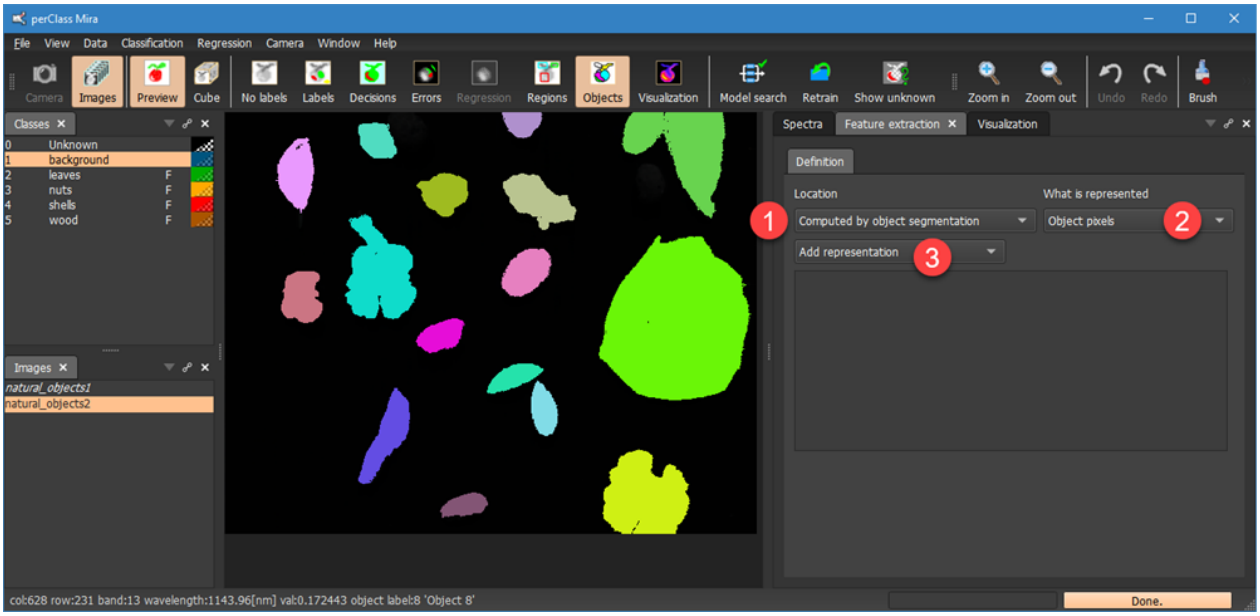
The content in double quotes specifies fully the color map. It is a list separating fields by semicolon. First the color map name is listed. Then, each of the color map steps is defined. Each step contains a 0.0 to 1.0 relative position followed by RGB definition of the color.

Feature extraction (exporting)

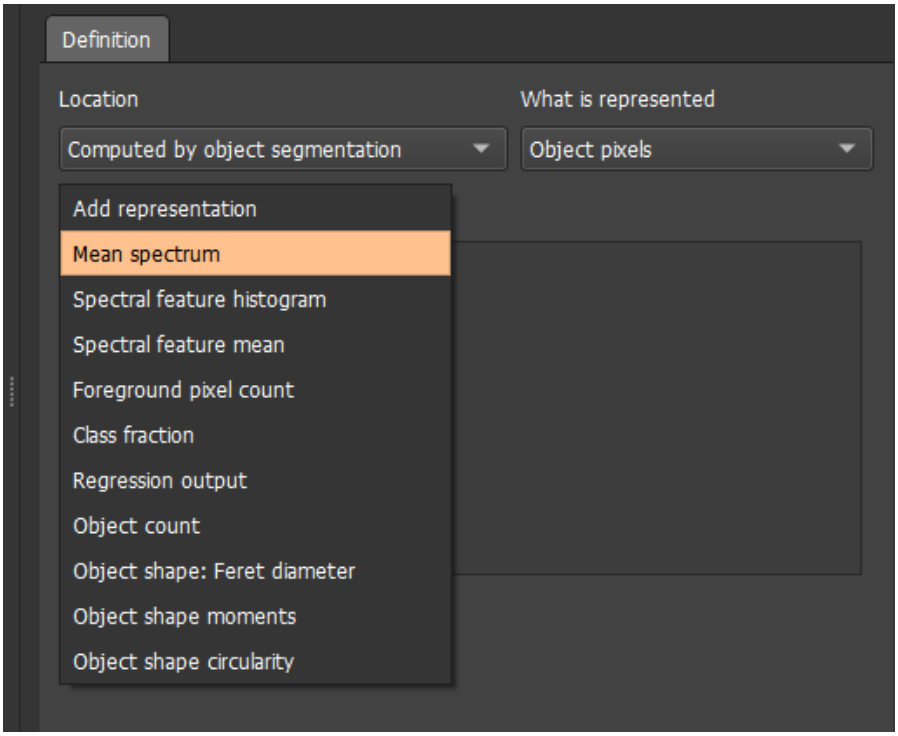
perClass Mira provides numerous ways how to extract and export information from a single or multiple images. The use case is to define classifier, segment objects or specify regions of interest and export user-defined features to external file (Excel .xls or XML). This data is the used for custom data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

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

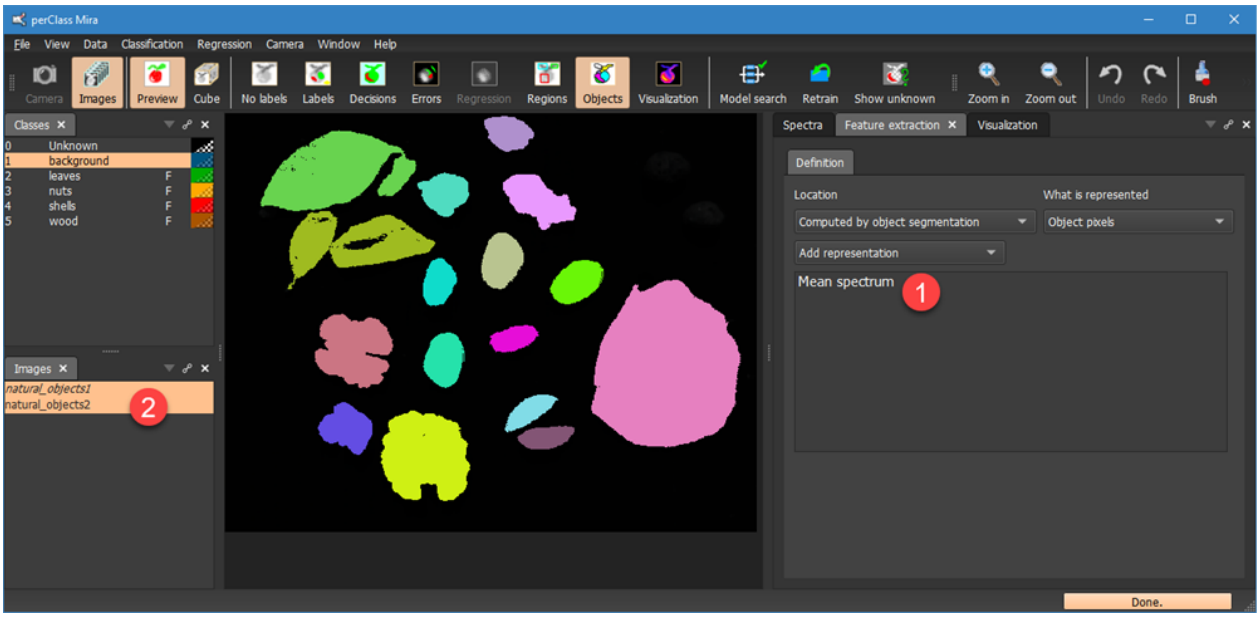
- Where the data is extracted from 1
- What pixels are included in the extraction 2
- What is being extracted 3



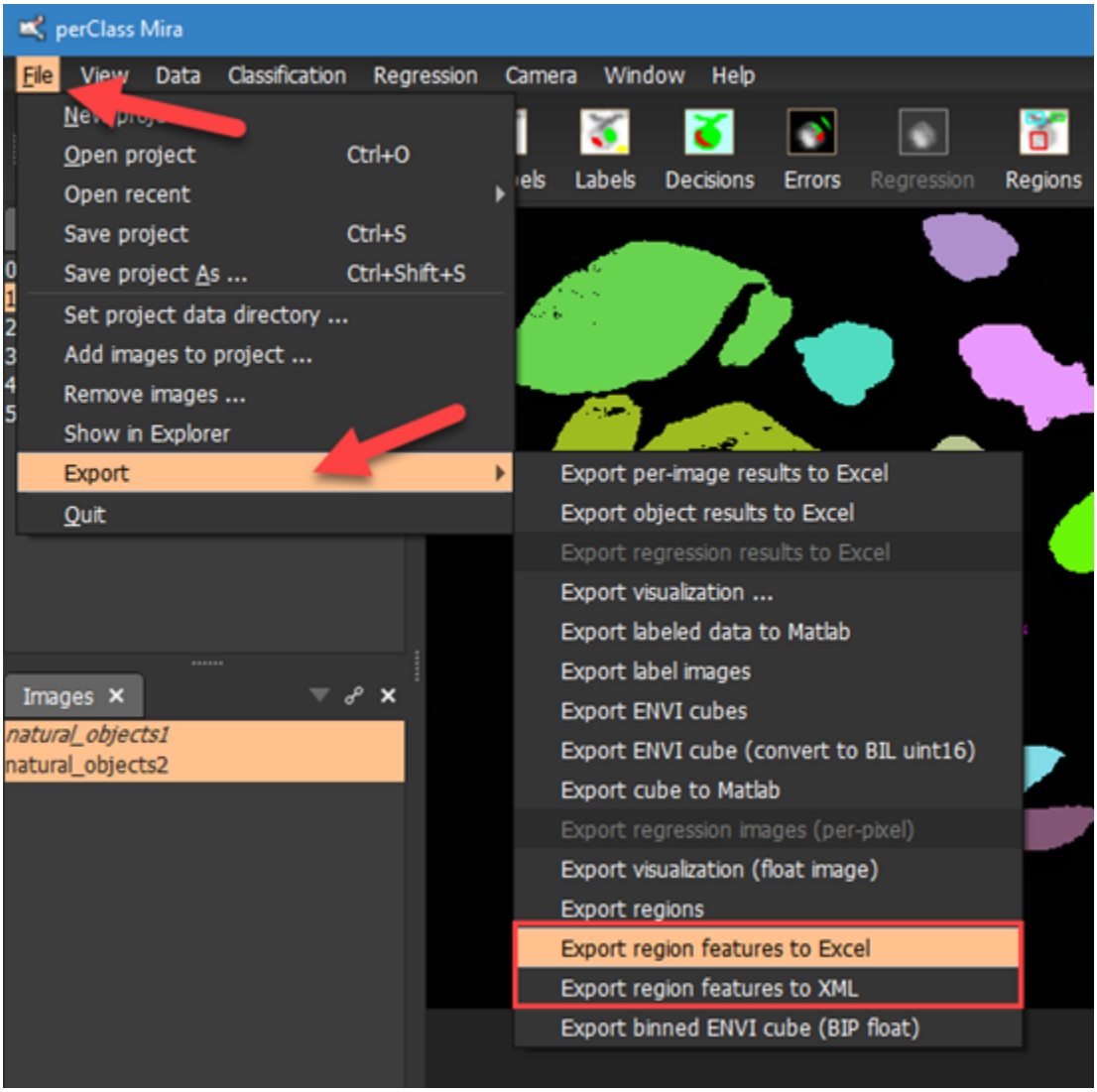
In the first example, we want to extract mean spectra from objects. Therefore, we select *Mean spectrum* from the *Add representation* combo box.



You may select multiple representations 1

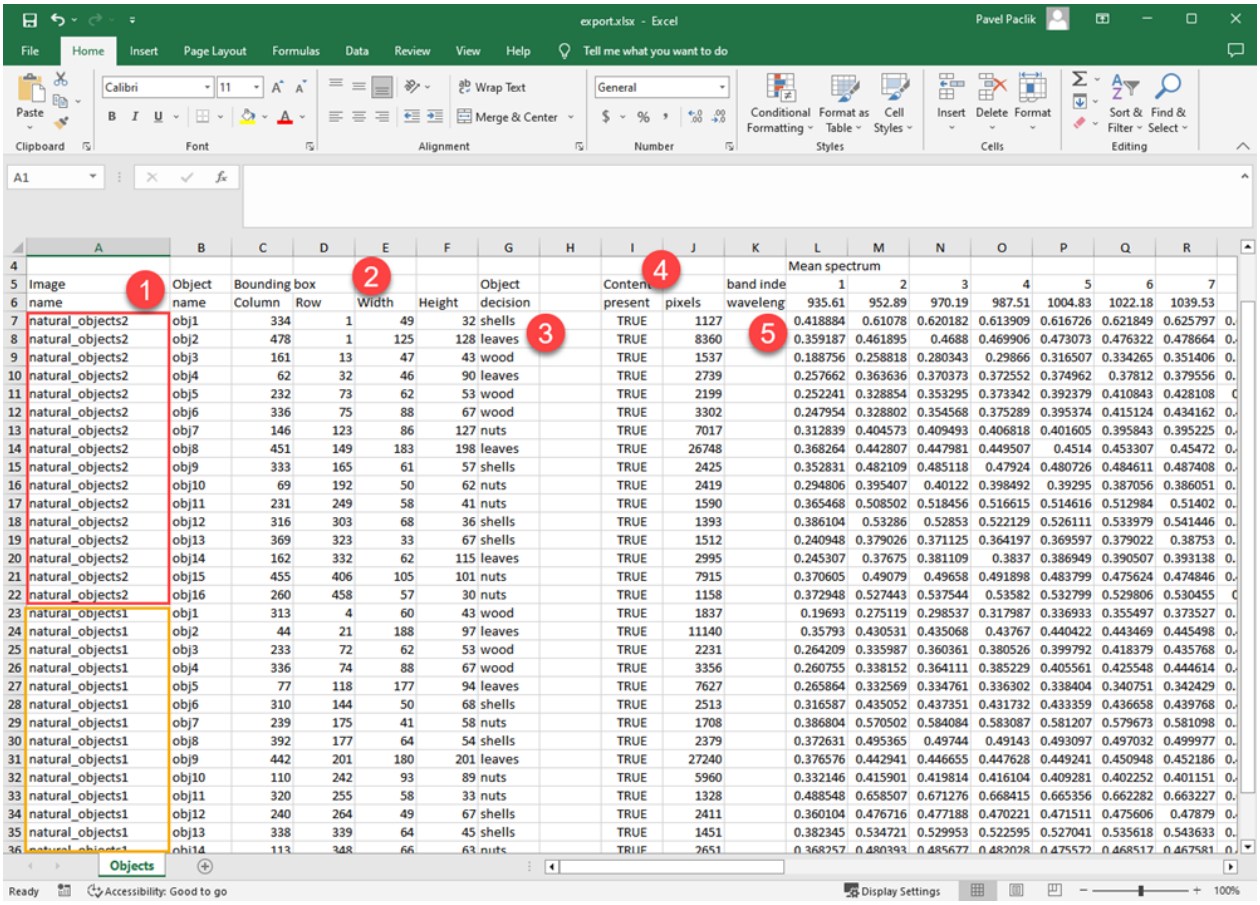


Select one of more images in *Images* list 2 and then *File* menu / *Export* and *Export region features to Excel.* Note you may also export the same data into XML. That option is more convenient if you wish to programmatically post-process data analysis.



You will 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:



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
4												Mean spectrum						
5	Image	Object	Bounding box				Object		Content		band inde	1	2	3	4	5	6	7
6	name	name	Column	Row	Width	Height	decision		present	pixels	waveleng	935.61	952.89	970.19	987.51	1004.83	1022.18	1039.53
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
8	natural_objects2	obj2	478	1	125	128	leaves		TRUE	8360		0.359187	0.461895	0.4688	0.469906	0.473073	0.476322	0.478664
9	natural_objects2	obj3	161	13	47	43	wood		TRUE	1537		0.188756	0.258818	0.280343	0.29866	0.316507	0.334265	0.351406
10	natural_objects2	obj4	62	32	46	90	leaves		TRUE	2739		0.257662	0.363636	0.370373	0.372552	0.374962	0.37812	0.379556
11	natural_objects2	obj5	232	73	62	53	wood		TRUE	2199		0.252241	0.328854	0.353295	0.373342	0.392379	0.410843	0.428108
12	natural_objects2	obj6	336	75	88	67	wood		TRUE	3302		0.247954	0.328802	0.354568	0.375289	0.395374	0.415124	0.434162
13	natural_objects2	obj7	146	123	86	127	nuts		TRUE	7017		0.312839	0.404573	0.409493	0.406818	0.401605	0.395843	0.395225
14	natural_objects2	obj8	451	149	183	198	leaves		TRUE	26748		0.368264	0.442807	0.447981	0.449507	0.4514	0.453307	0.45472
15	natural_objects2	obj9	333	165	61	57	shells		TRUE	2425		0.352831	0.482109	0.485118	0.47924	0.480726	0.484611	0.487408
16	natural_objects2	obj10	69	192	50	62	nuts		TRUE	2419		0.294806	0.395407	0.40122	0.398492	0.39295	0.387056	0.386051
17	natural_objects2	obj11	231	249	58	41	nuts		TRUE	1590		0.365468	0.508502	0.518456	0.516615	0.514616	0.512984	0.51402
18	natural_objects2	obj12	316	303	68	36	shells		TRUE	1393		0.386104	0.53286	0.52853	0.522129	0.526111	0.533979	0.541446
19	natural_objects2	obj13	369	323	33	67	shells		TRUE	1512		0.240948	0.379026	0.371125	0.364197	0.369597	0.379022	0.38753
20	natural_objects2	obj14	162	332	62	115	leaves		TRUE	2995		0.245307	0.37675	0.381109	0.3837	0.386949	0.390507	0.393138
21	natural_objects2	obj15	455	406	105	101	nuts		TRUE	7915		0.370605	0.49079	0.49658	0.491898	0.483799	0.475624	0.474846
22	natural_objects2	obj16	260	458	57	30	nuts		TRUE	1158		0.372948	0.527443	0.537544	0.53582	0.532799	0.529806	0.530455
23	natural_objects1	obj1	313	4	60	43	wood		TRUE	1837		0.19693	0.275119	0.298537	0.317987	0.336933	0.355497	0.373527
24	natural_objects1	obj2	44	21	188	97	leaves		TRUE	11140		0.35793	0.430531	0.435068	0.43767	0.440422	0.443469	0.445498
25	natural_objects1	obj3	233	72	62	53	wood		TRUE	2231		0.264209	0.335987	0.360361	0.380526	0.399792	0.418379	0.435768
26	natural_objects1	obj4	336	74	88	67	wood		TRUE	3356		0.260755	0.338152	0.364111	0.385229	0.405561	0.425548	0.444614
27	natural_objects1	obj5	77	118	177	94	leaves		TRUE	7627		0.265864	0.332569	0.334761	0.336302	0.338404	0.340751	0.342429
28	natural_objects1	obj6	310	144	50	68	shells		TRUE	2513		0.316587	0.435052	0.437351	0.431732	0.433359	0.436658	0.439768
29	natural_objects1	obj7	239	175	41	58	nuts		TRUE	1708		0.386804	0.570502	0.584084	0.583087	0.581207	0.579673	0.581098
30	natural_objects1	obj8	392	177	64	54	shells		TRUE	2379		0.372631	0.495365	0.49744	0.49143	0.493097	0.497032	0.499977
31	natural_objects1	obj9	442	201	180	201	leaves		TRUE	27240		0.376576	0.442941	0.446655	0.447628	0.449241	0.450948	0.452186
32	natural_objects1	obj10	110	242	93	89	nuts		TRUE	5960		0.332146	0.415901	0.419814	0.416104	0.409281	0.402252	0.401151
33	natural_objects1	obj11	320	255	58	33	nuts		TRUE	1328		0.488548	0.658507	0.671276	0.668415	0.665356	0.662282	0.663227
34	natural_objects1	obj12	240	264	49	67	shells		TRUE	2411		0.360104	0.476716	0.477188	0.470221	0.471511	0.475606	0.47879
35	natural_objects1	obj13	338	339	64	45	shells		TRUE	1451		0.382345	0.534721	0.529953	0.522595	0.527041	0.535618	0.543633
36	natural_objects1	obj14	113	348	66	63	nuts		TRUE	2651		0.368257	0.480393	0.485677	0.482028	0.475572	0.468517	0.467581

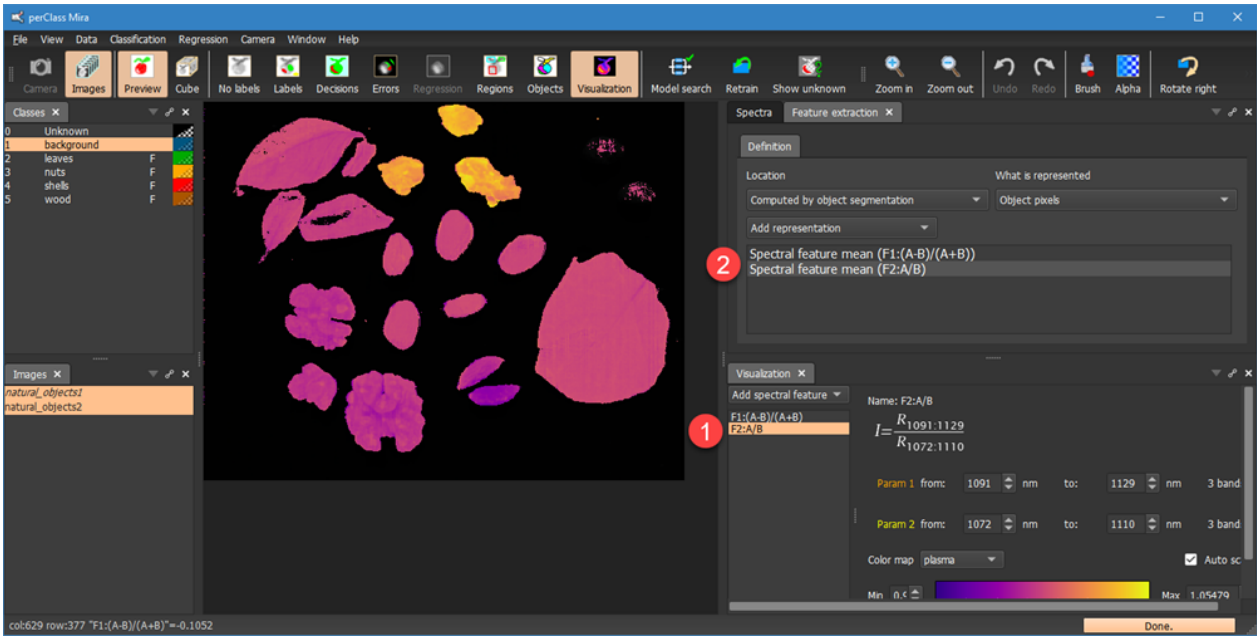
Objects of each selected image are described by rows. For each object, we can see the scan name (1) followed by an object name. When exporting object segmentation, the object name is automatically assigned in the segmentation procedure. When exporting content of user-defined regions, the region name is used. This can be user-assigned. For each object, its bounding box (2) and per-object decision (3) is provided. The *Content present* column shows (4) 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 (5) contains the exported data. In our case, the columne 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.

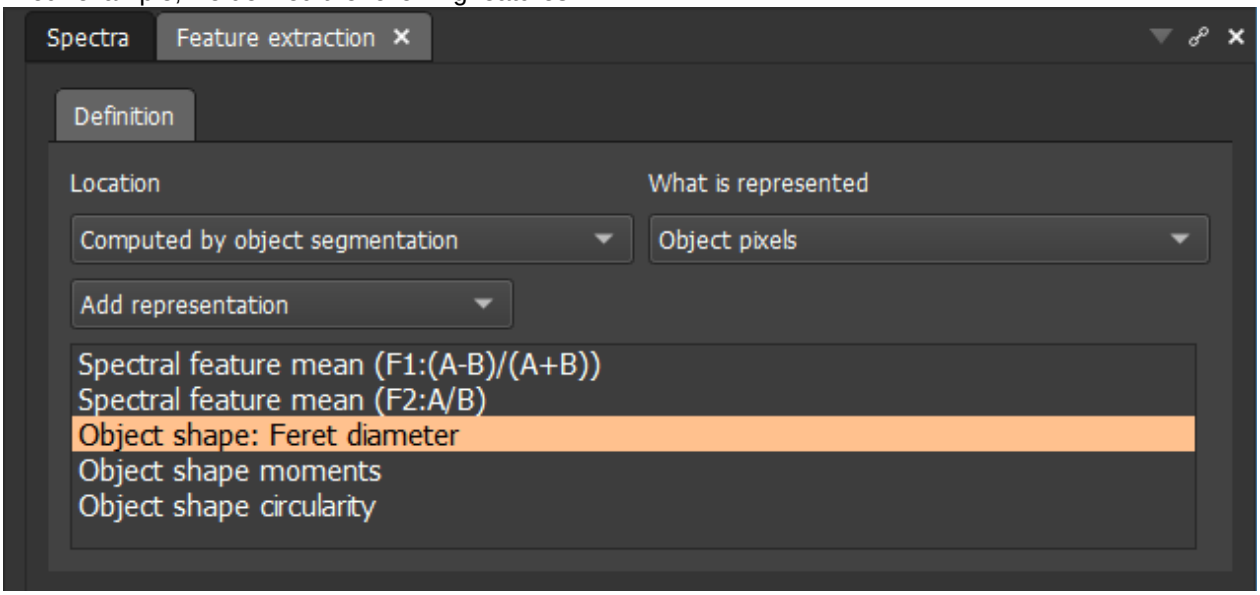
Say, we wish to extract two spectral indices and some shape representation of each object. We have defined two spectral indices. We need to select a specific index (1) and then choose the desired reprensetation in the *Add representation* combo to include it in the list (2).



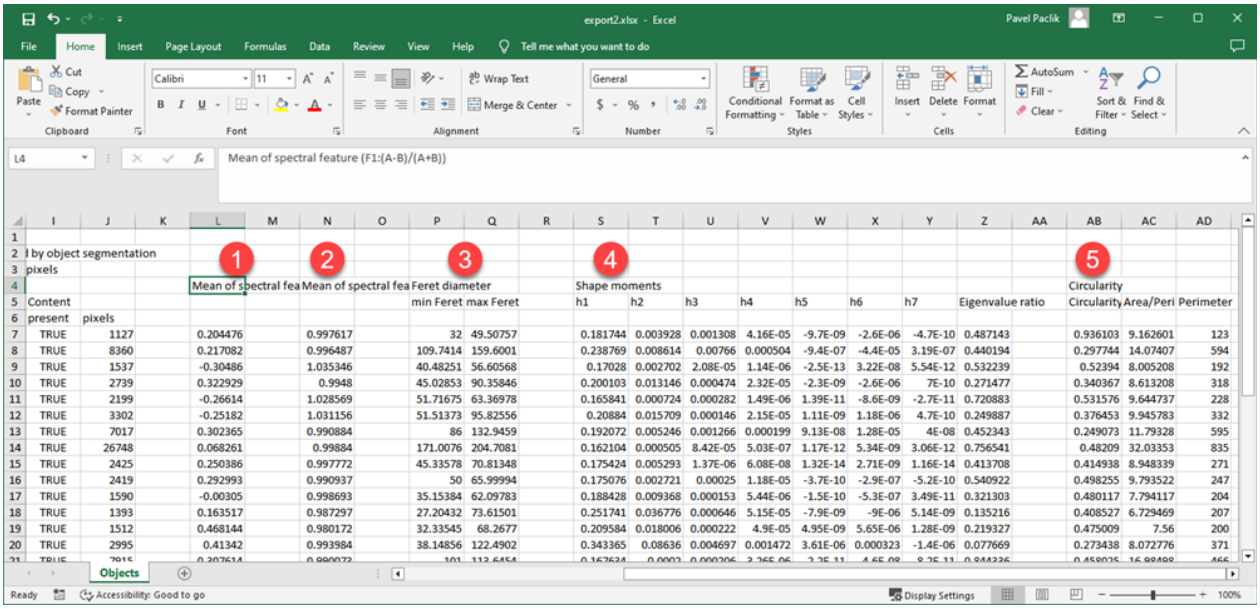
Available feature types

- **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 minmum 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 cirle 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:

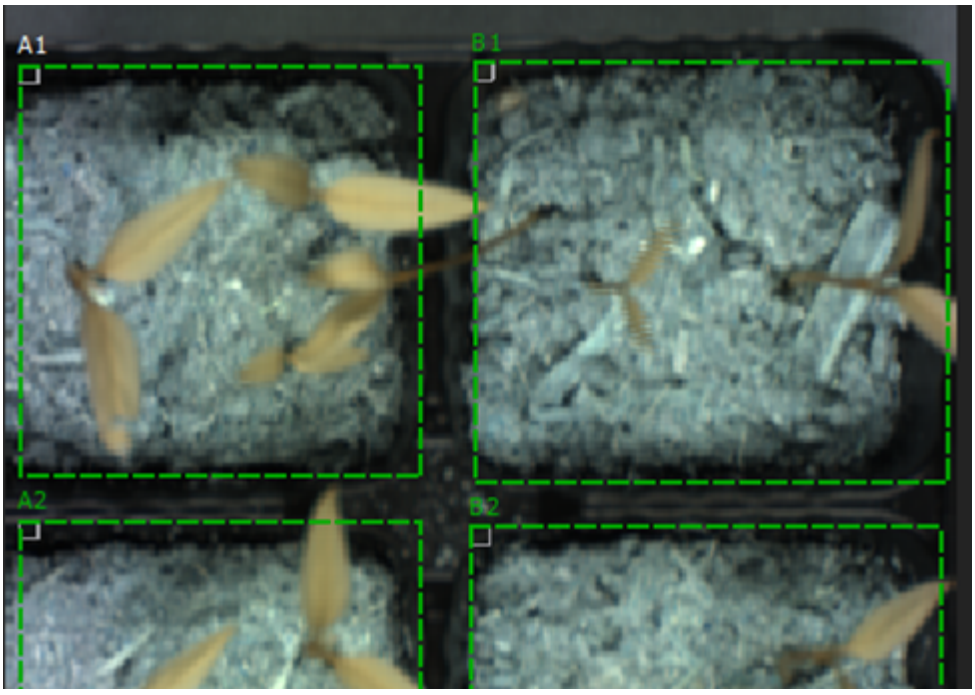


Below screenshot of the exported Excel file with indicated 5 feature groups.

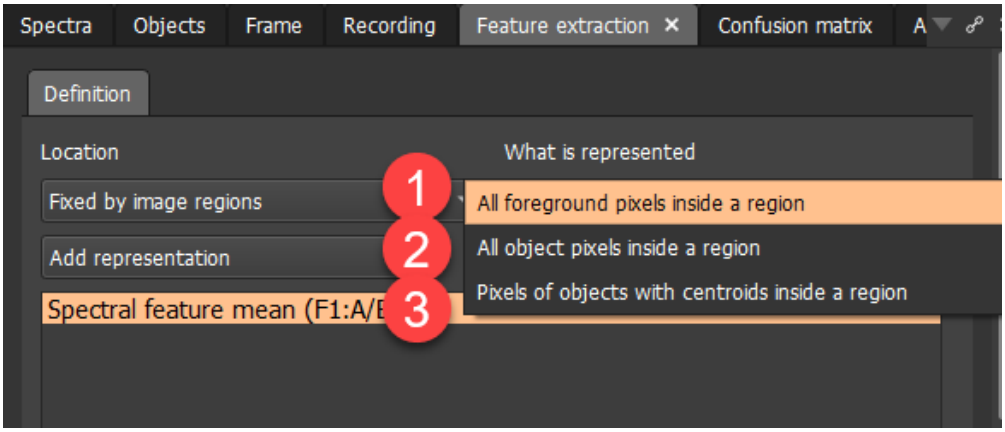


Extracting from region grid

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 featue extraction from regions is, that we may detect absence of data in a cell (for example, when the seed did not germinate).



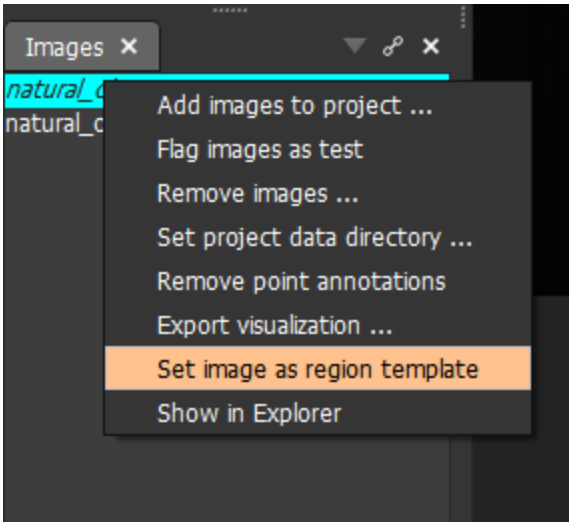
When using regions for feature extraction, we have few options how to define what pixels are included in the processing:



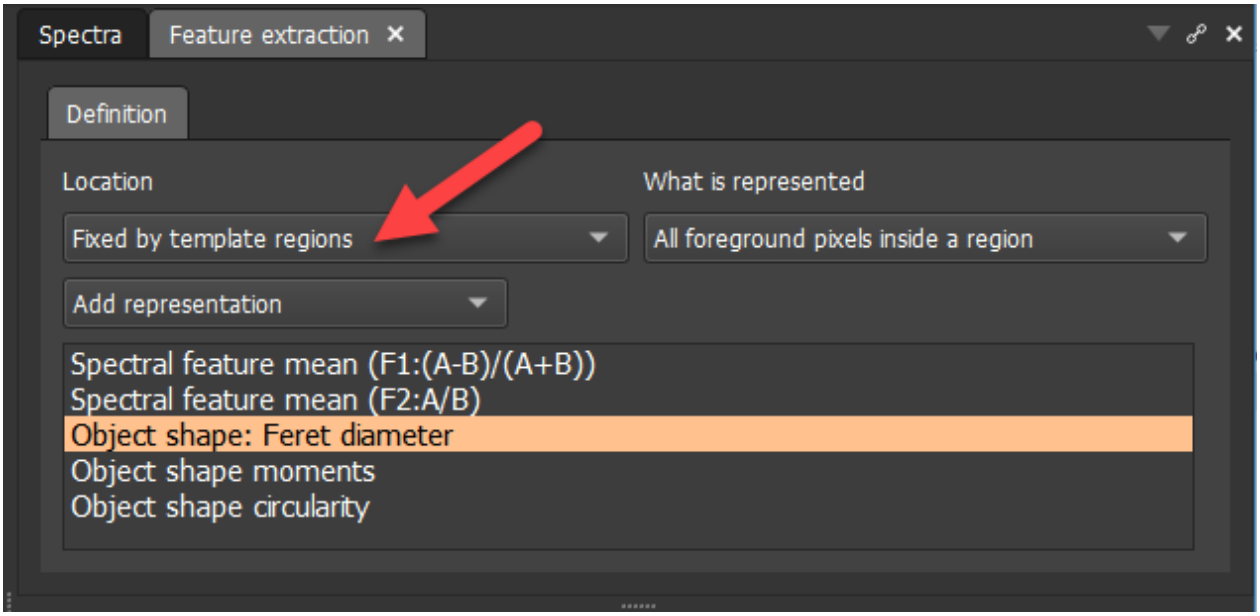
The option 1 simply includes all foreground pixels within the region. The option 2 only considers the object pixels. Therefore, pixels of small objects (with size smaller than the defined minimum object size) are not included. Finally, the option 3 includes only pixels of objects with cetroids within 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:

```
<?xml version="1.0" encoding="UTF-8"?>
<object_representation xmlns="http://perclass.com/mira" version="perClass Mira 4.2 (28-feb-2023)">
    <location>Computed by object segmentation</location>
    <what_is_represented>All object pixels</what_is_represented>
    <representations>
        <representation index="1" name="Spectral feature mean (F1:(A-B)/(A+B))" type="spectral_feature_mean"/>
        <representation index="2" name="Spectral feature mean (F2:A/B)" type="spectral_feature_mean"/>
        <representation index="3" name="Object shape: Feret diameter"/>
        <representation index="4" name="Object shape moments"/>
        <representation index="5" name="Object shape circularity"/>
    </representations>
    <images count="2">
        <image index="2" name="natural_objects2">
            <objects count="16">
                <object index="1" type="computed" name="obj1" bbox="334 1 49 32">
                    <representations>
                        <representation index="1" name="Spectral feature mean (F1:(A-B)/(A+B))" present="true" pixels="1127">0.204476</representation>
                        <representation index="2" name="Spectral feature mean (F2:A/B)" present="true" pixels="1127">0.997617</representation>
                        <representation index="3" name="Object shape: Feret diameter"/>
                        <representation index="4" name="Object shape moments"/>
                        <representation index="5" name="Object shape circularity"/>
                    </representations>
                </object>
                <object index="2" type="computed" name="obj2" bbox="478 1 125 128">
                    <representations>
                        <representation index="1" name="Spectral feature mean (F1:(A-B)/(A+B))" present="true" pixels="8360">0.217082</representation>
                        <representation index="2" name="Spectral feature mean (F2:A/B)" present="true" pixels="8360">0.996487</representation>
                        <representation index="3" name="Object shape: Feret diameter"/>
                        <representation index="4" name="Object shape moments"/>
                        <representation index="5" name="Object shape circularity"/>
                    </representations>
                </object>
                <object index="3" type="computed" name="obj3" bbox="161 13 47 43">
                    <representations>
                        <representation index="1" name="Spectral feature mean (F1:(A-B)/(A+B))" present="true" pixels="1537">-0.304859</representation>
                        <representation index="2" name="Spectral feature mean (F2:A/B)" present="true" pixels="1537">1.03535</representation>
```

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

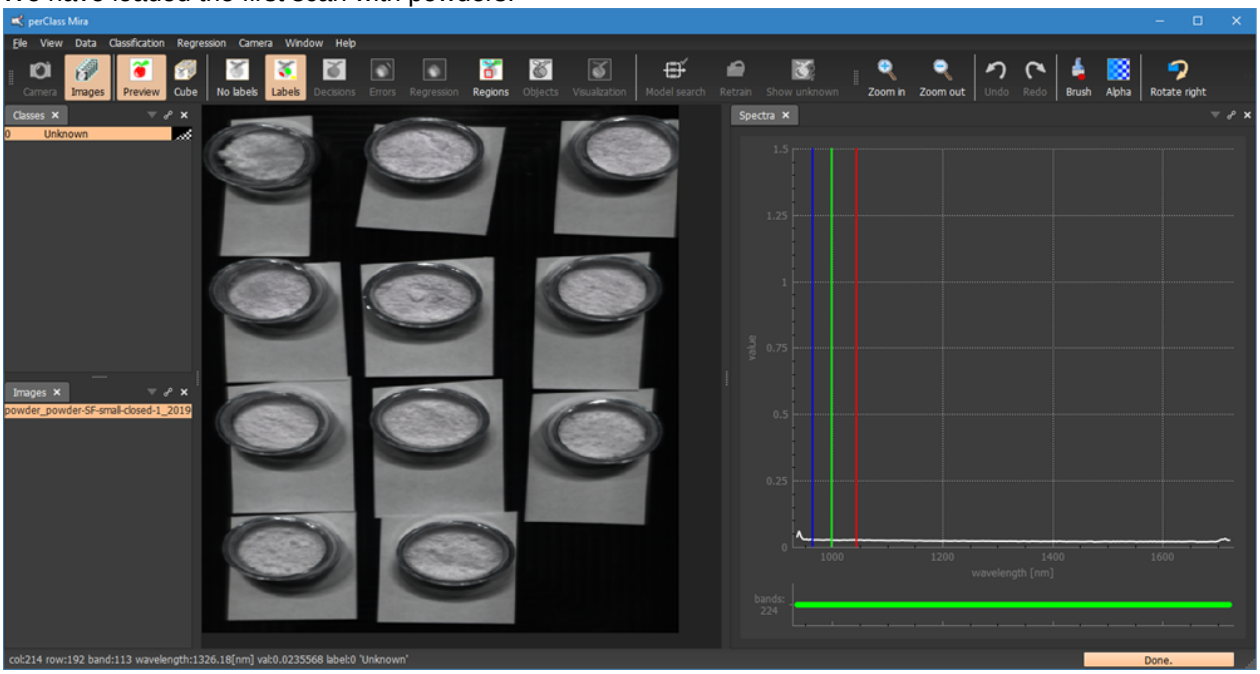
Regression

Regression modeing 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.

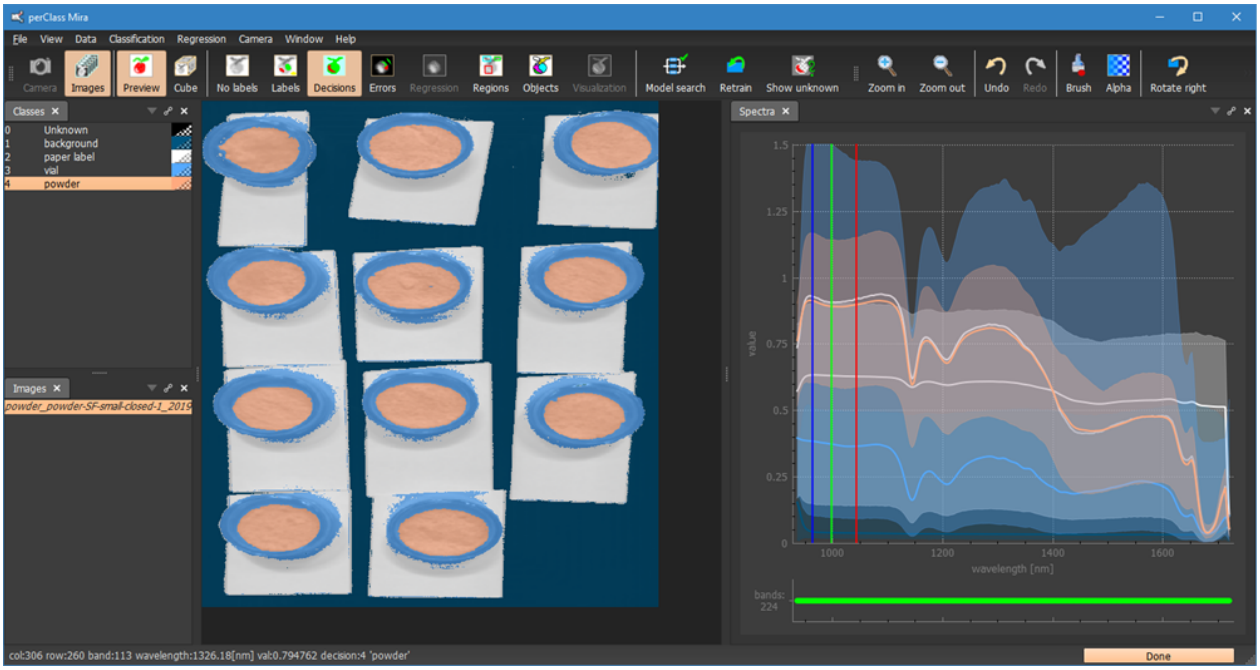
In this example, we use the powder data set with vials containing mixtures of two powders, namely flower and soda. Our goal is to train a model that will be able to estimate the mixing propotion for a new powder mix.

We have loaded the first scan with powders:



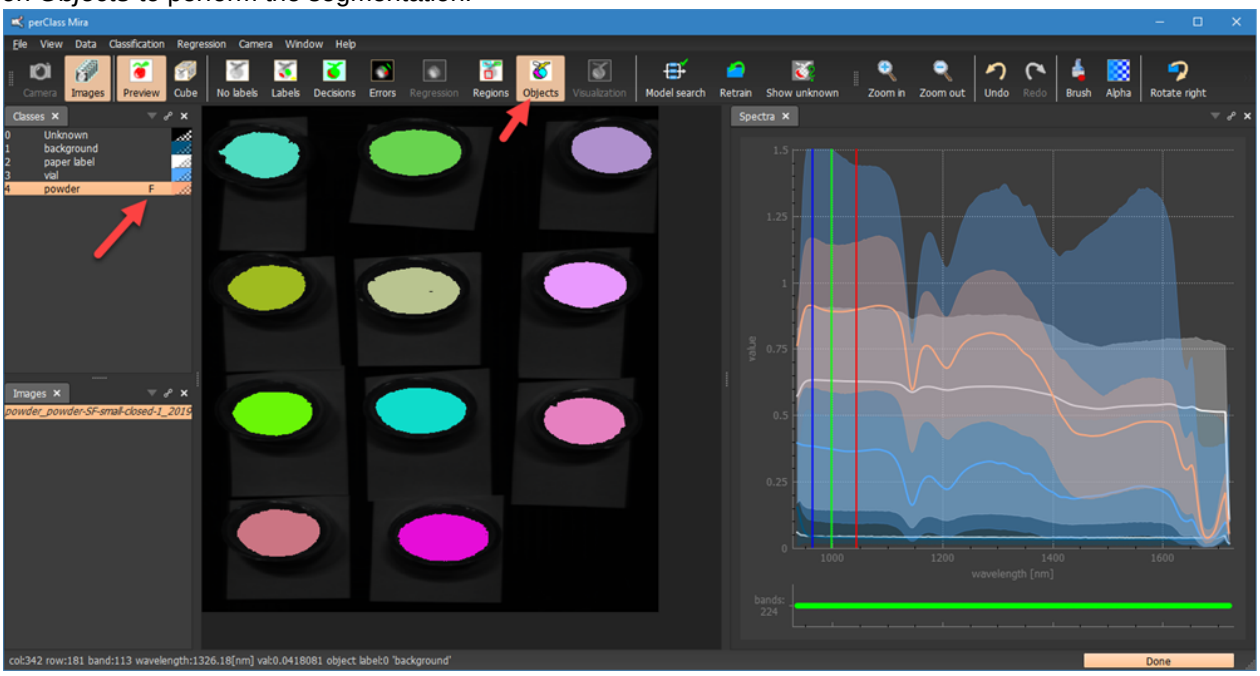
Step 1: Pixel classification

In the first step, we create a pixel classifier. We only care about good wuality 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.



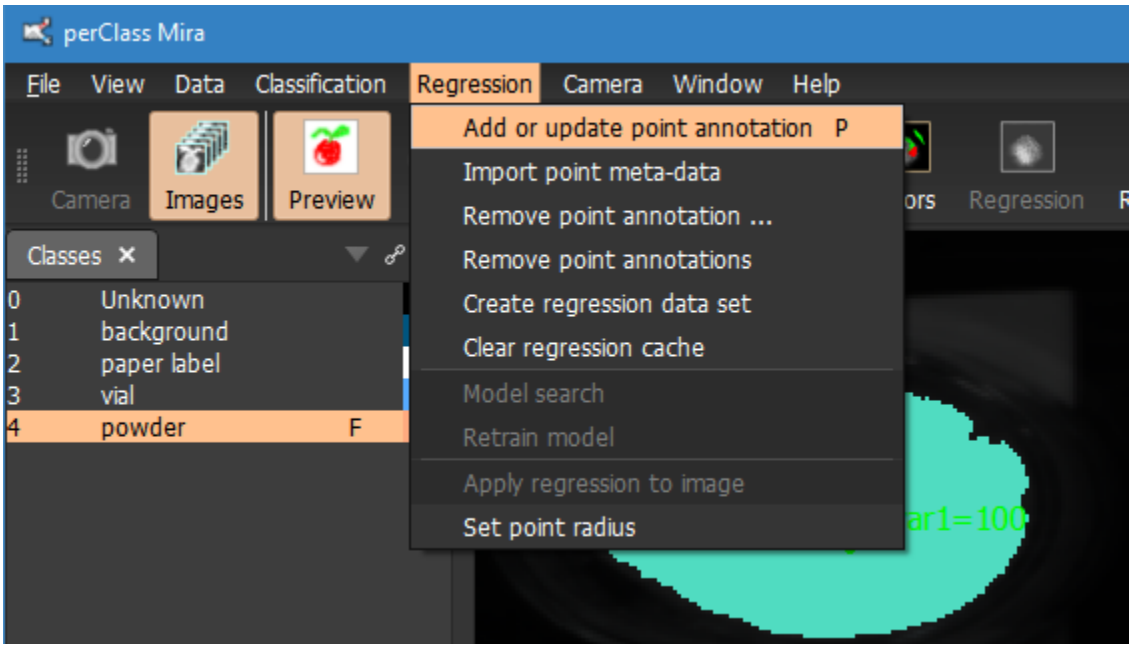
You may use more then one foreground class in regression. For example, you may build a model with separate pixel classes for white and dark grapes, flag both as foreground and use both in regression modeling.

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:

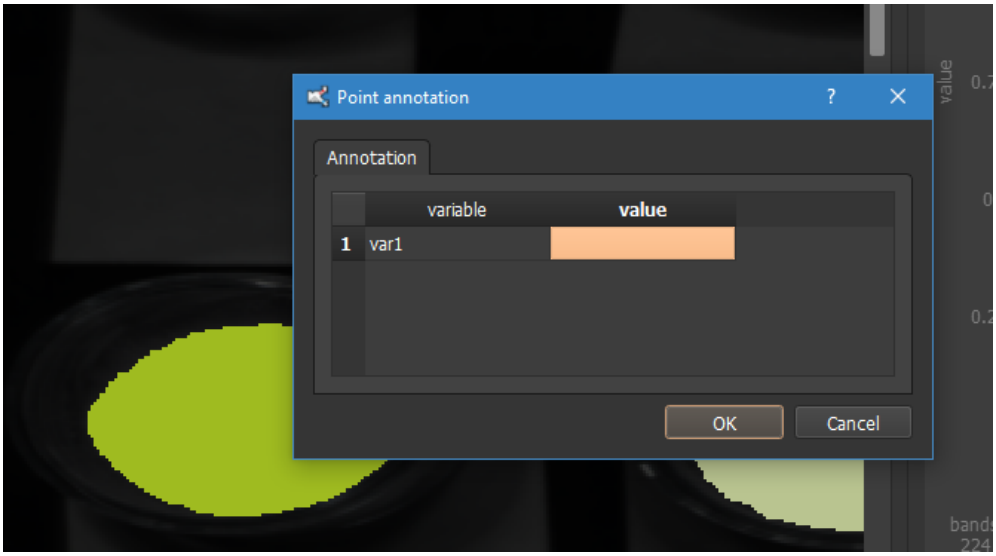
For each powder object, we add a single number that corresponds to the percentage of soda in the sample.

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

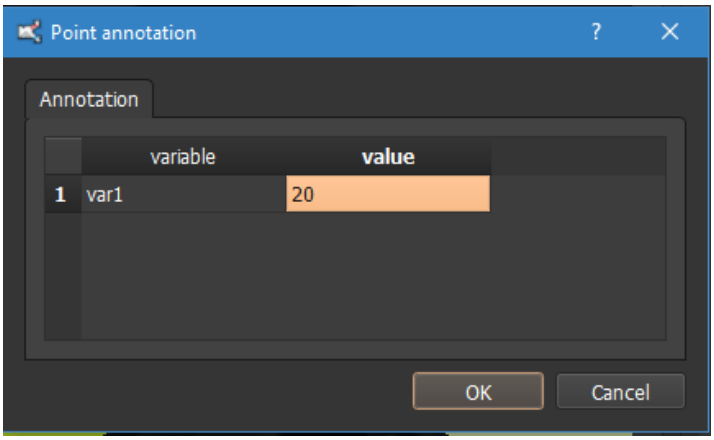


More convenient is to use the keyboard shortcut. We position the mouse pointer on top of the desired

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



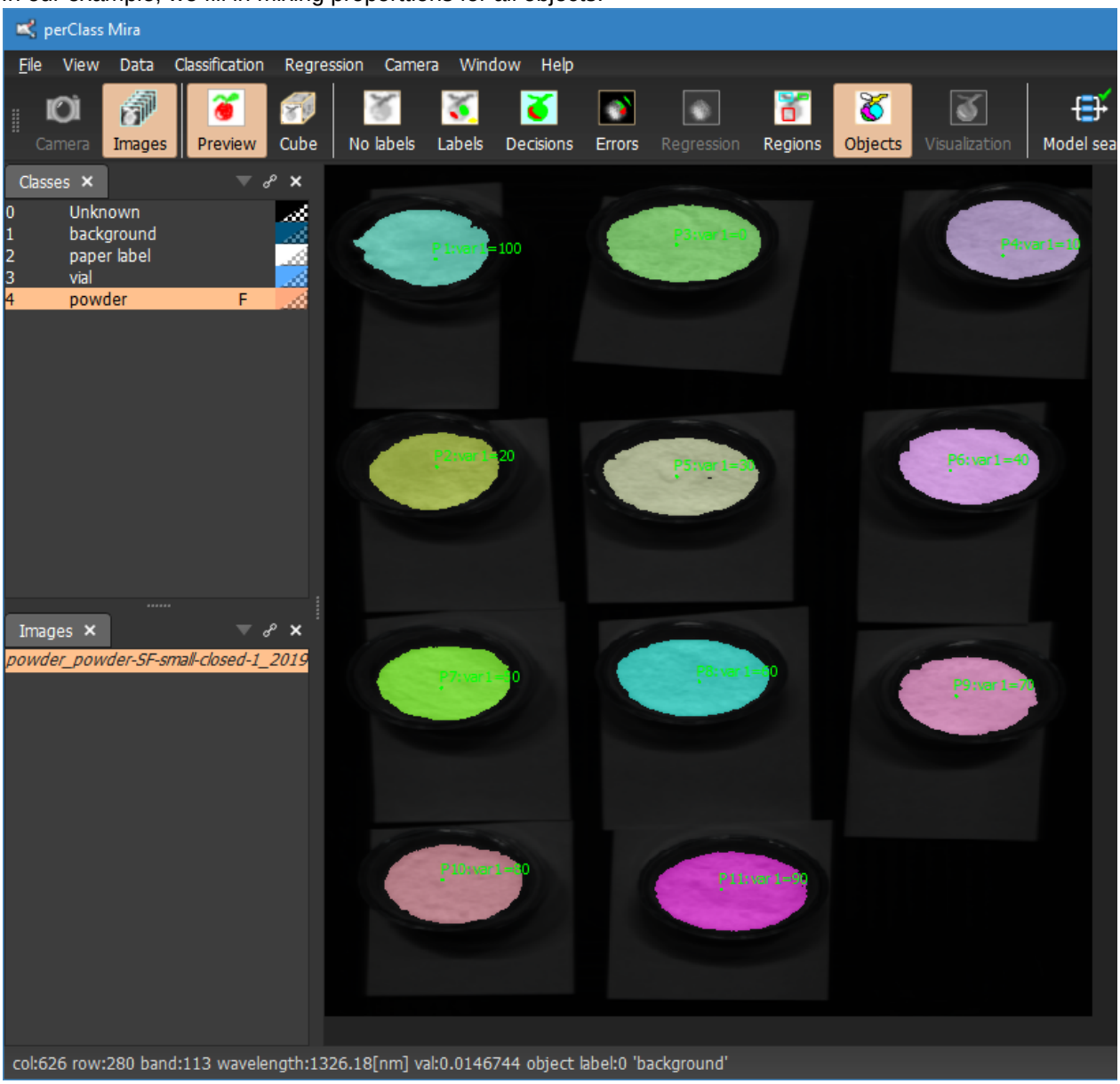
We may directly type in the numerical ground truth value, in our case 20 and press Enter twice (the first time to confirm entering the value, the second time to confirm the dialog):



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

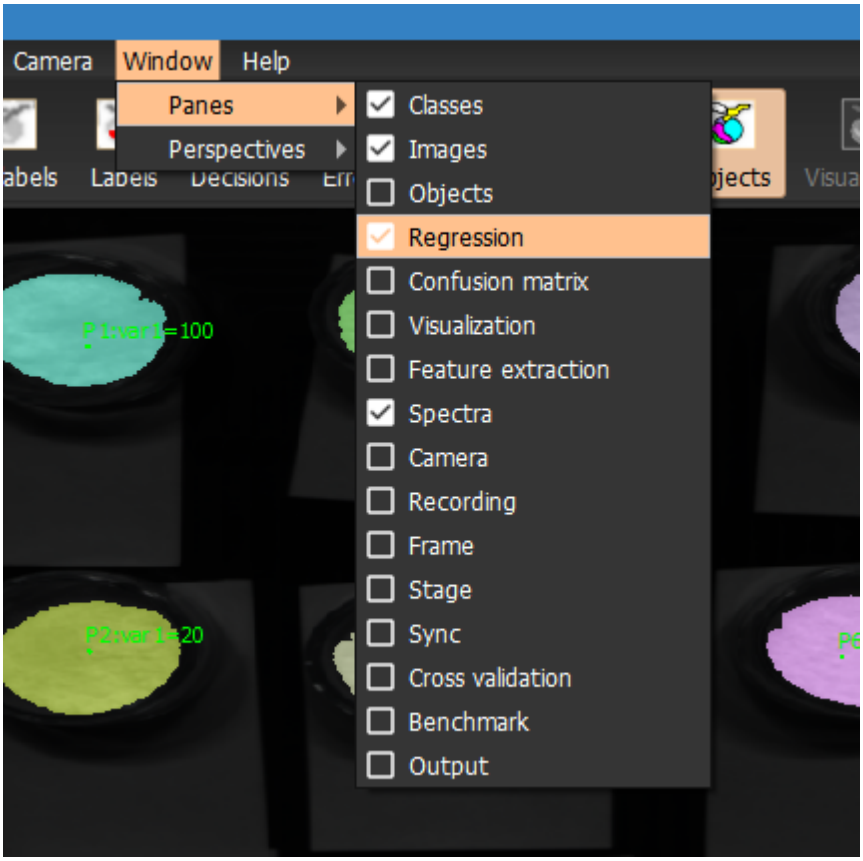
Each point has a unique number in the project which is assigned automatically. We may move the point around. By doible clicking the point, we may edit the attached values.

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

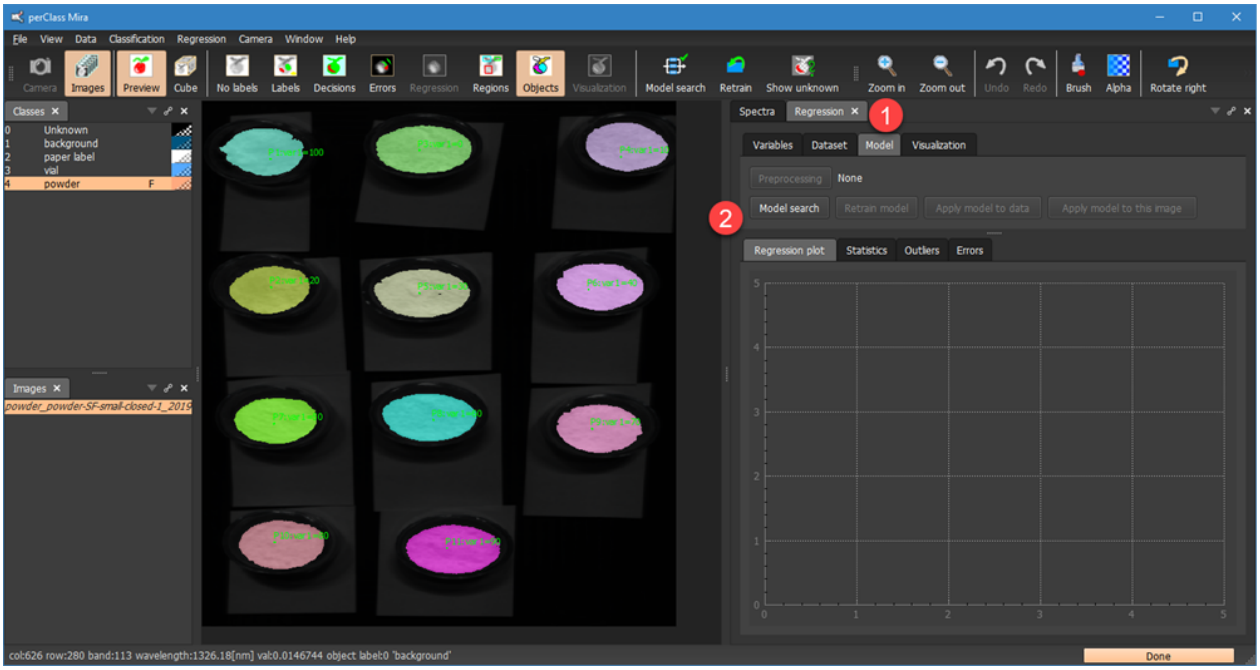


Step 4: Regression modeling

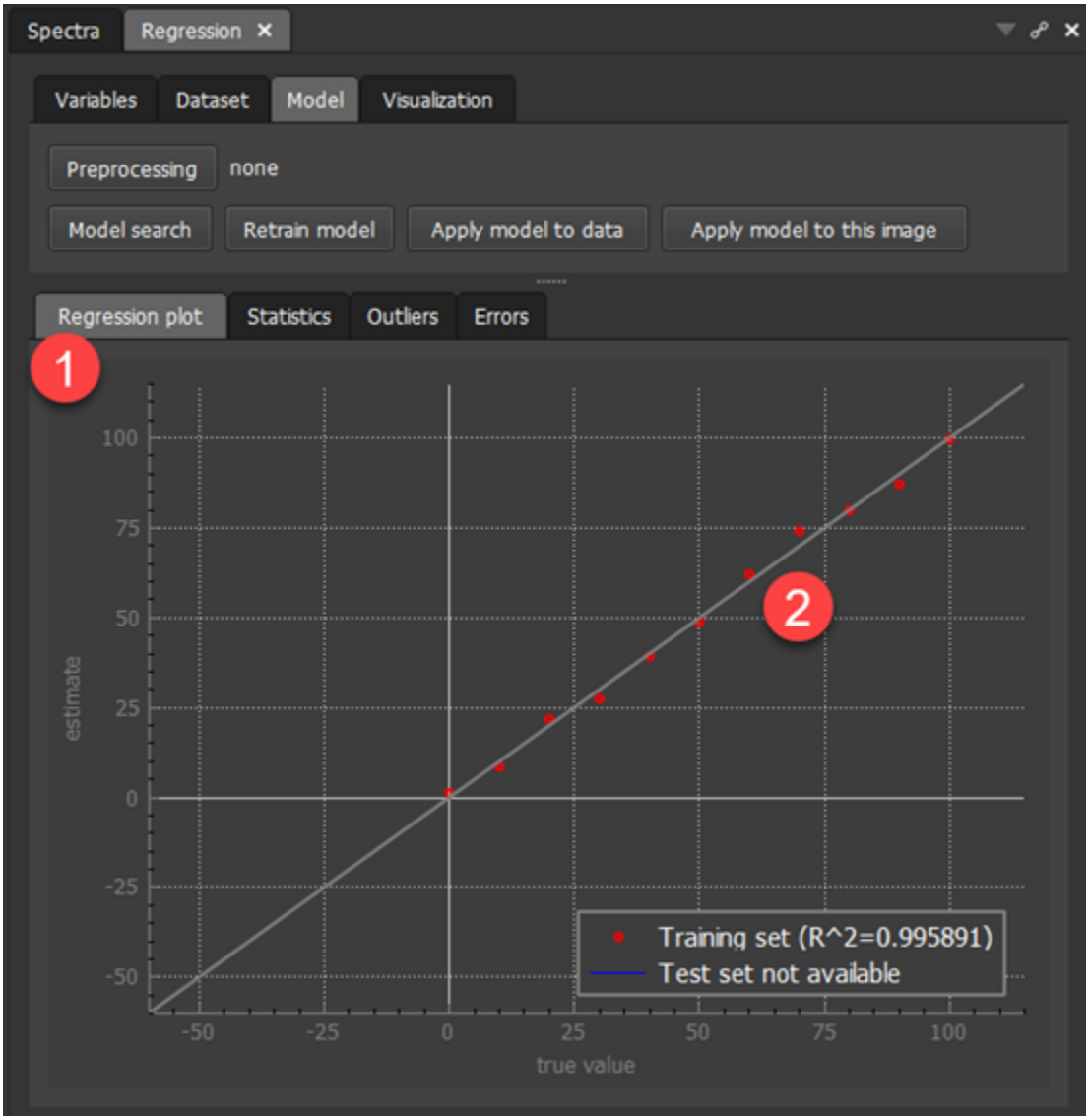
In the fourth step, we will build a regression model. We open the *Regression* panel (if not visible, enable it in the *Window* / *Panes* menu)



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:

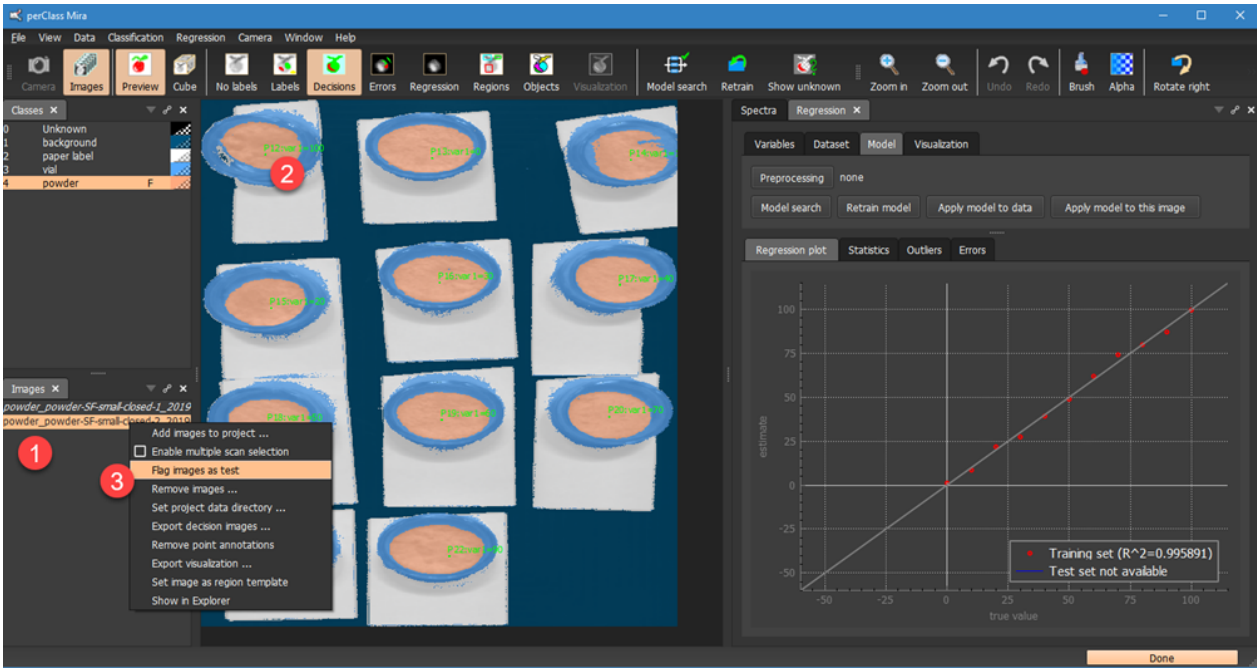


Specifically, the regression plot 1 will show number of red points 2 . Each point corresponds to one object. The red color denotes training objects.

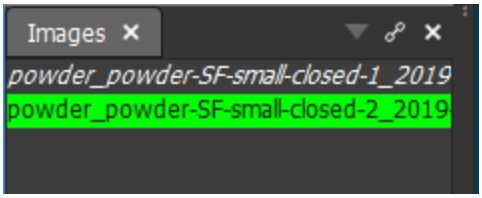
Step 5: Defining test data set

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.

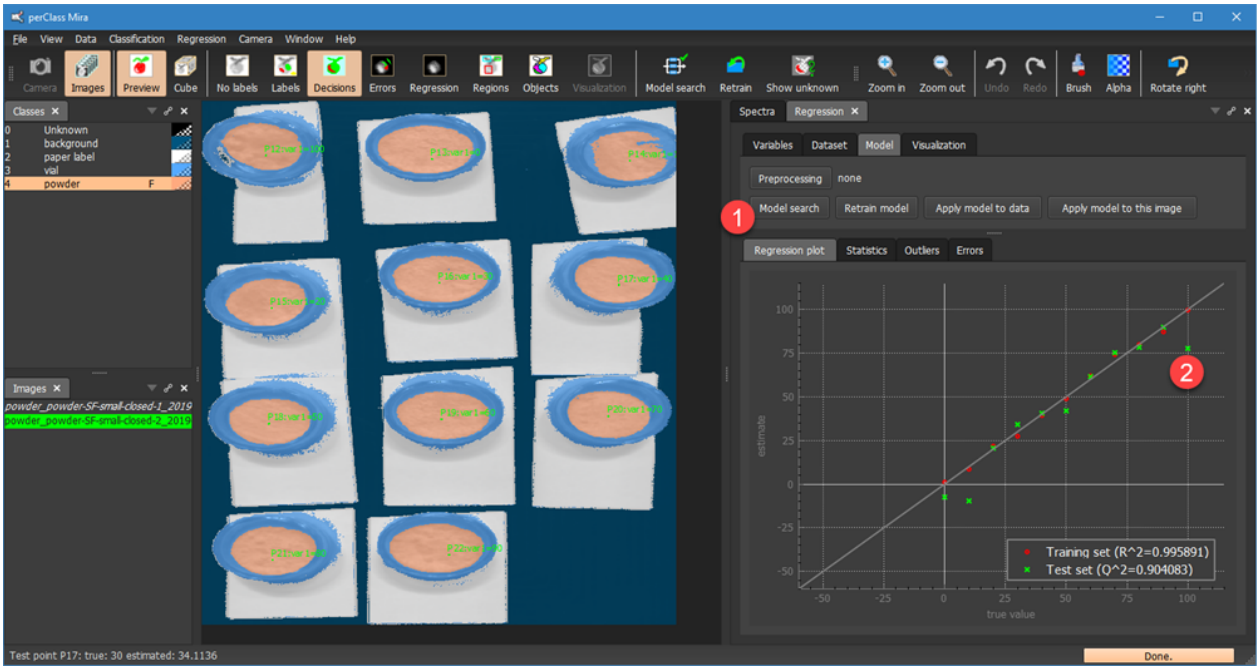
In perClass Mira, test flag applies to entire images. Therefore, we need to include at least one more image and annotate its objects with the numerical gound truth.



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 2 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 regresion 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

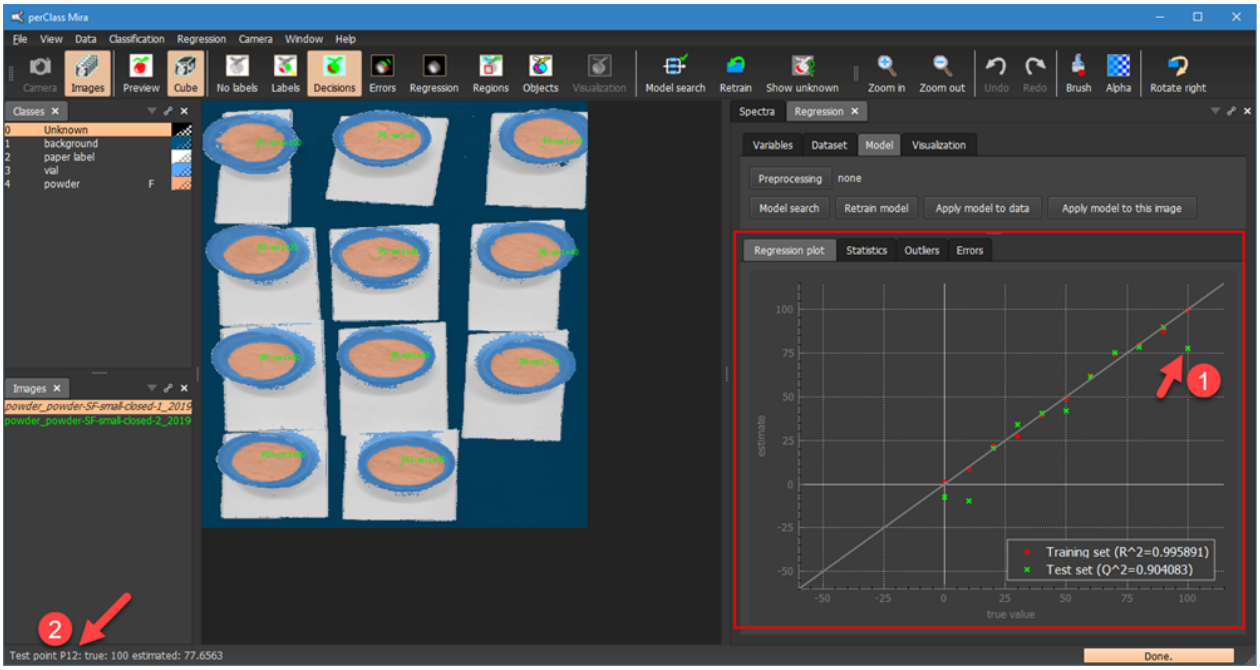
We may improve the regression model by

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

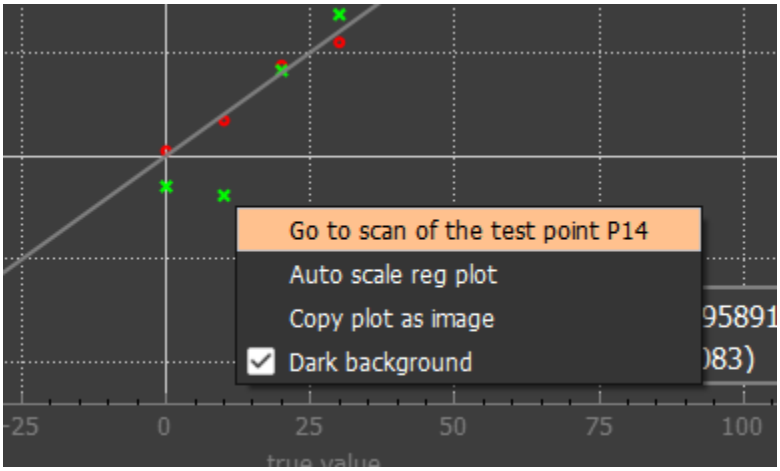
Regression plot

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 1 are displayed in the status bar 2 .

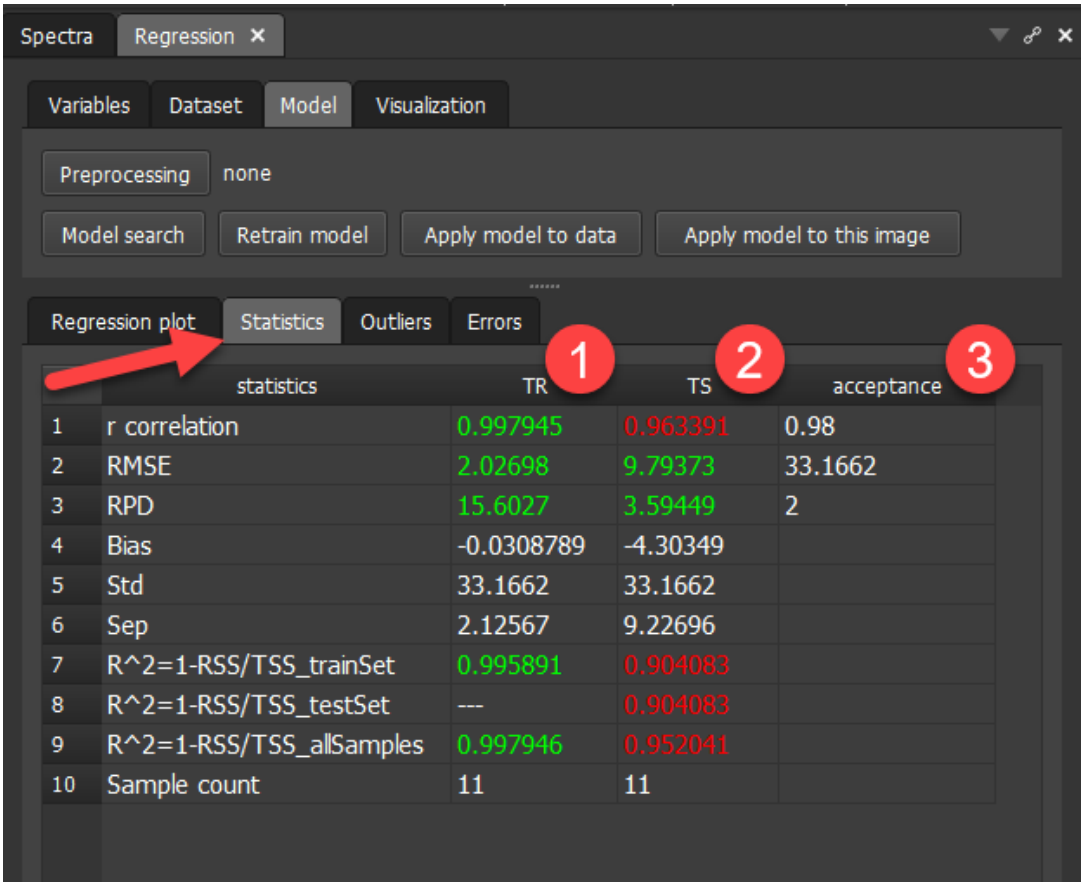


We may jump to the scan of the nearest point using the context menu and *Got to the scan of the point* command:



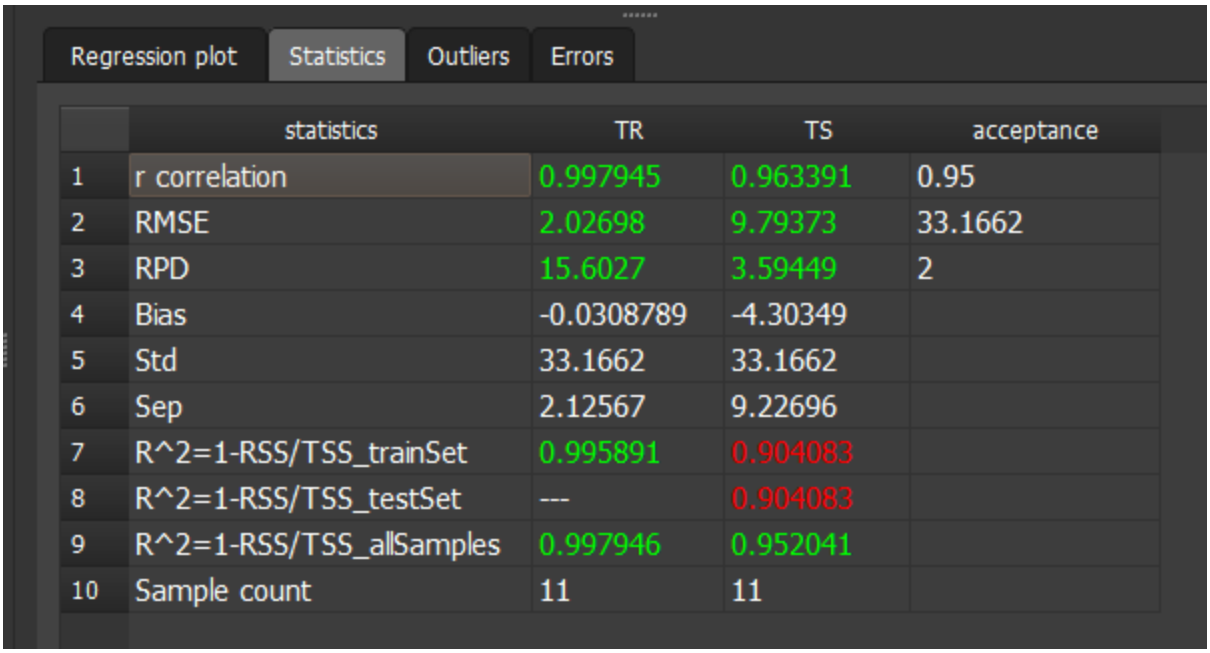
Performance statistics

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



	statistics	TR	TS	acceptance
1	r correlation	0.997945	0.963391	0.98
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^2=1-RSS/TSS_trainSet	0.995891	0.904083	
8	R^2=1-RSS/TSS_testSet	---	0.904083	
9	R^2=1-RSS/TSS_allSamples	0.997946	0.952041	
10	Sample count	11	11	

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^2=1-RSS/TSS_trainSet	0.995891	0.904083	
8	R^2=1-RSS/TSS_testSet	---	0.904083	
9	R^2=1-RSS/TSS_allSamples	0.997946	0.952041	
10	Sample count	11	11	

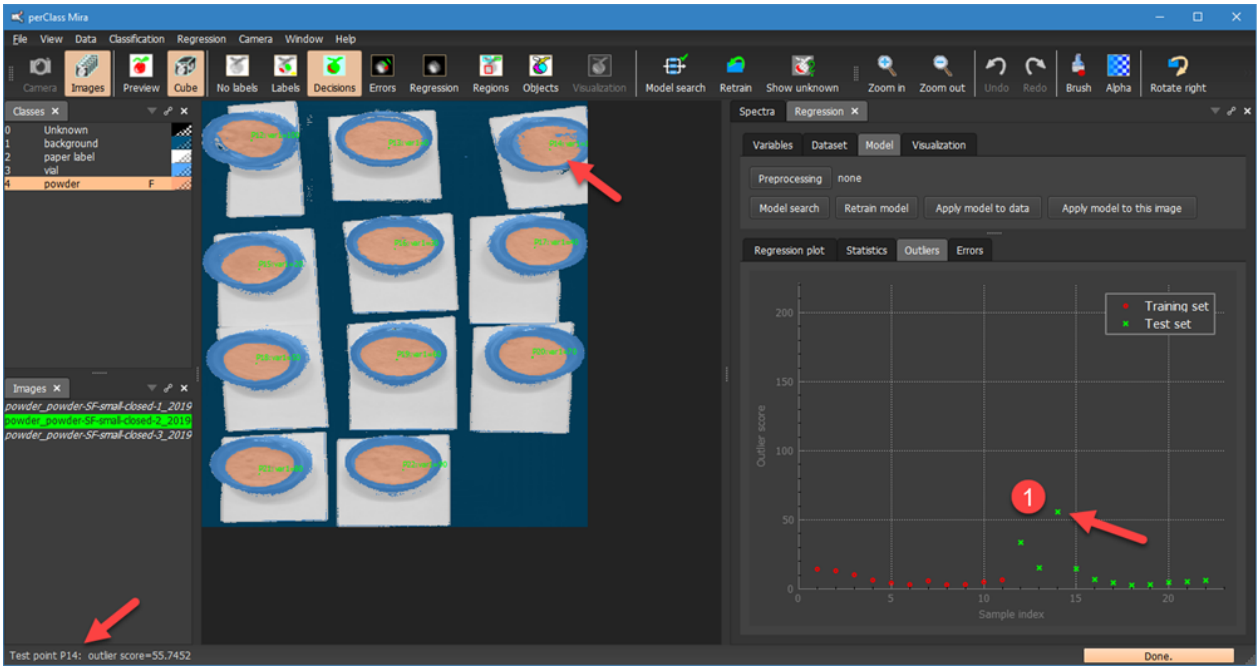
Performance statistics may be copied out as text by selecting specific cells in the table and using *Copy statistics as text* command. These values can be directly pasted into Excel table.

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 1 is the P14 indicated in the status bar and in the image by arrows.

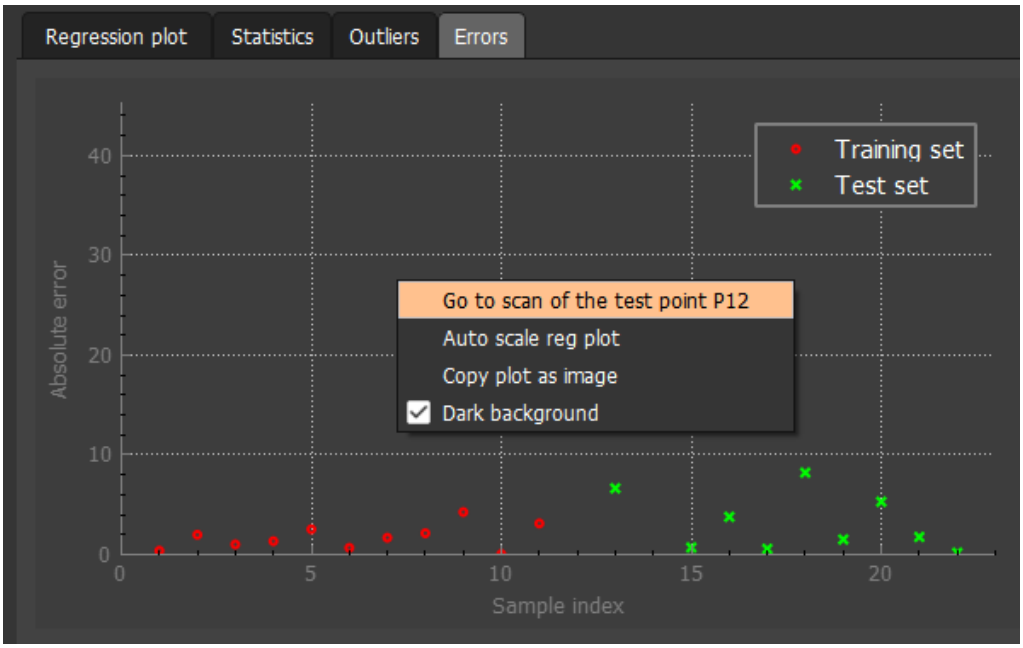


Note, that ground truth information is not needed when computing the outlier score. Therefore, it may help us to understand any new observations.

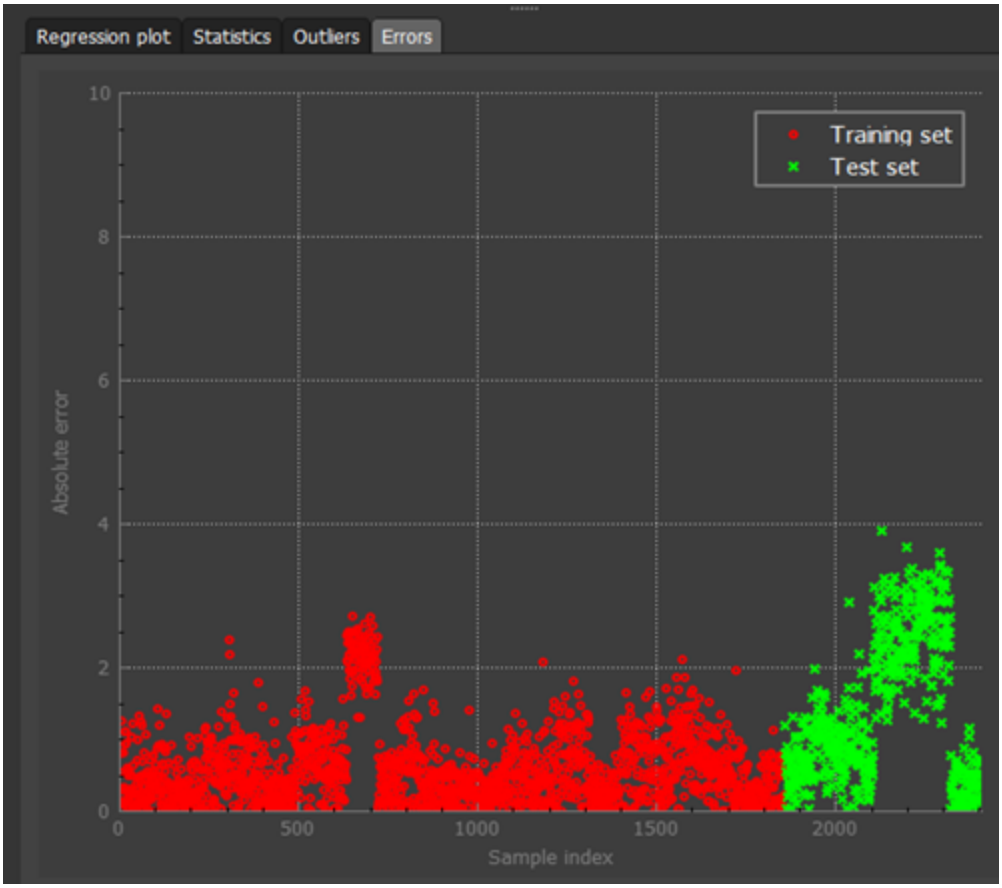
Error plot

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.



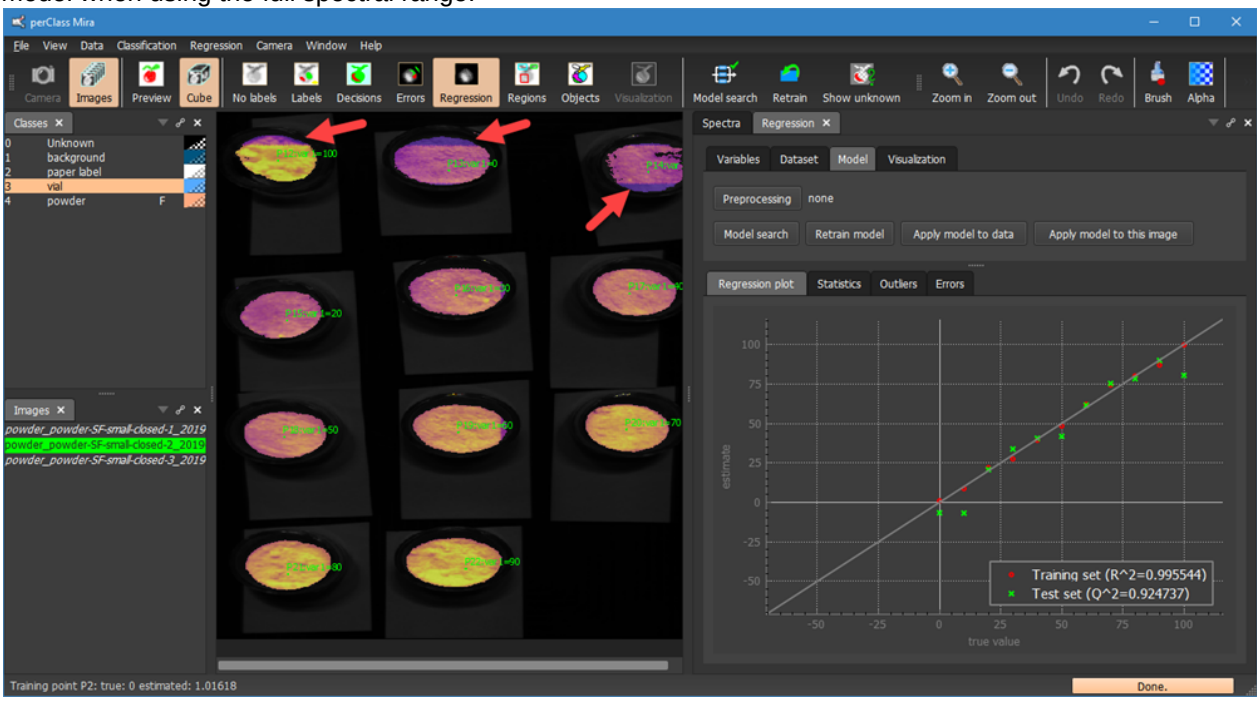
The example below originates from a different real-world project. It shows that we may easily spot systematic errors do to strong grouping information presented in the error (or outlier) plots. The groups with higher error may refer to different fruit varieties, producers, processing setting and similar.



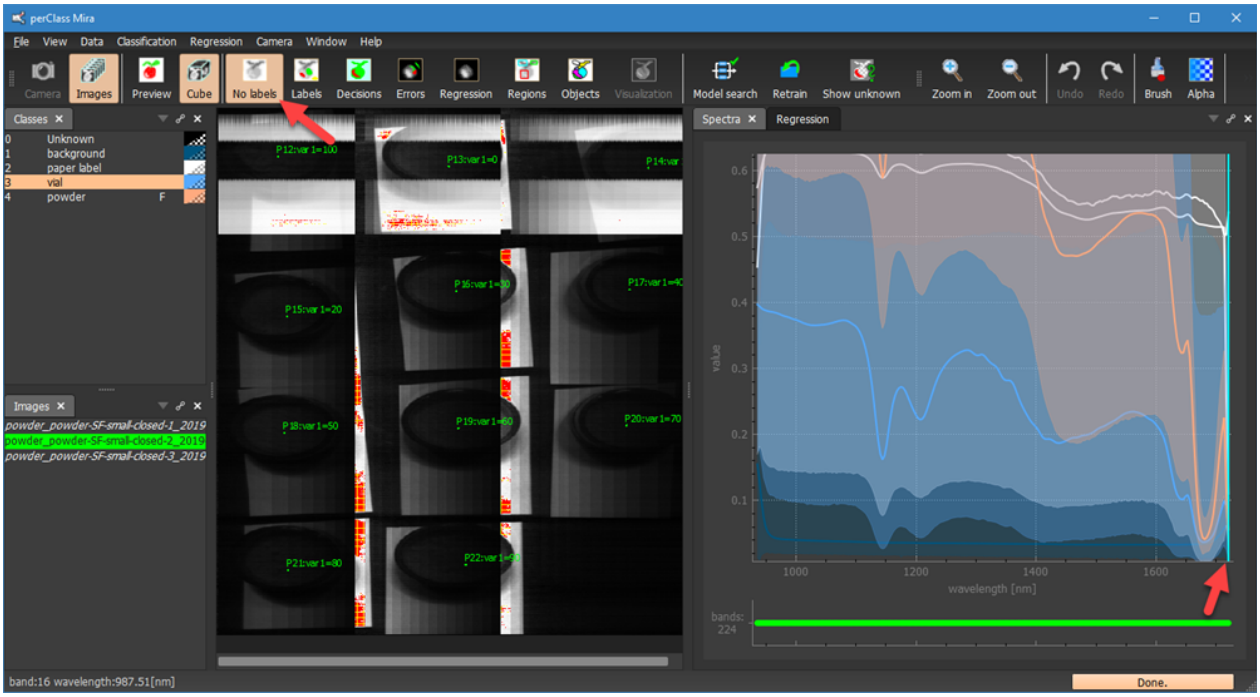
Regression using subset of bands

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

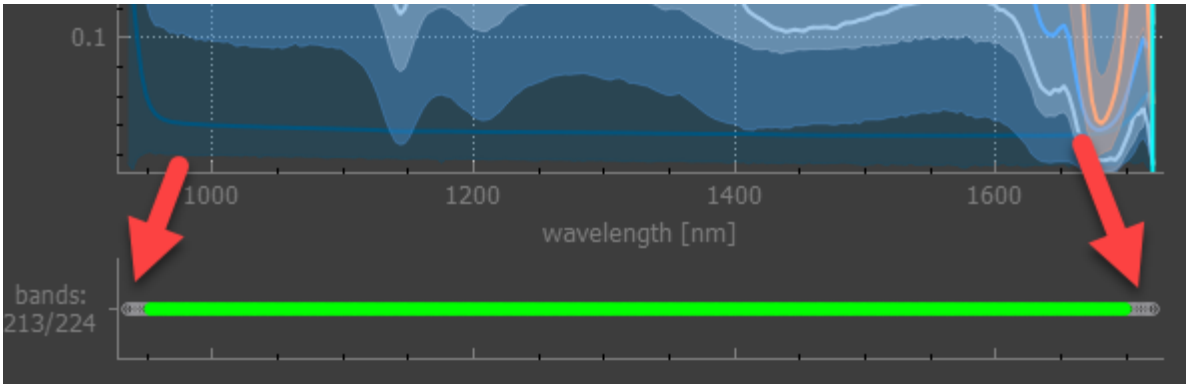
In this example, we use Pixel visualization of the regression output to show that there is some issue with our model when using the full spectral range:



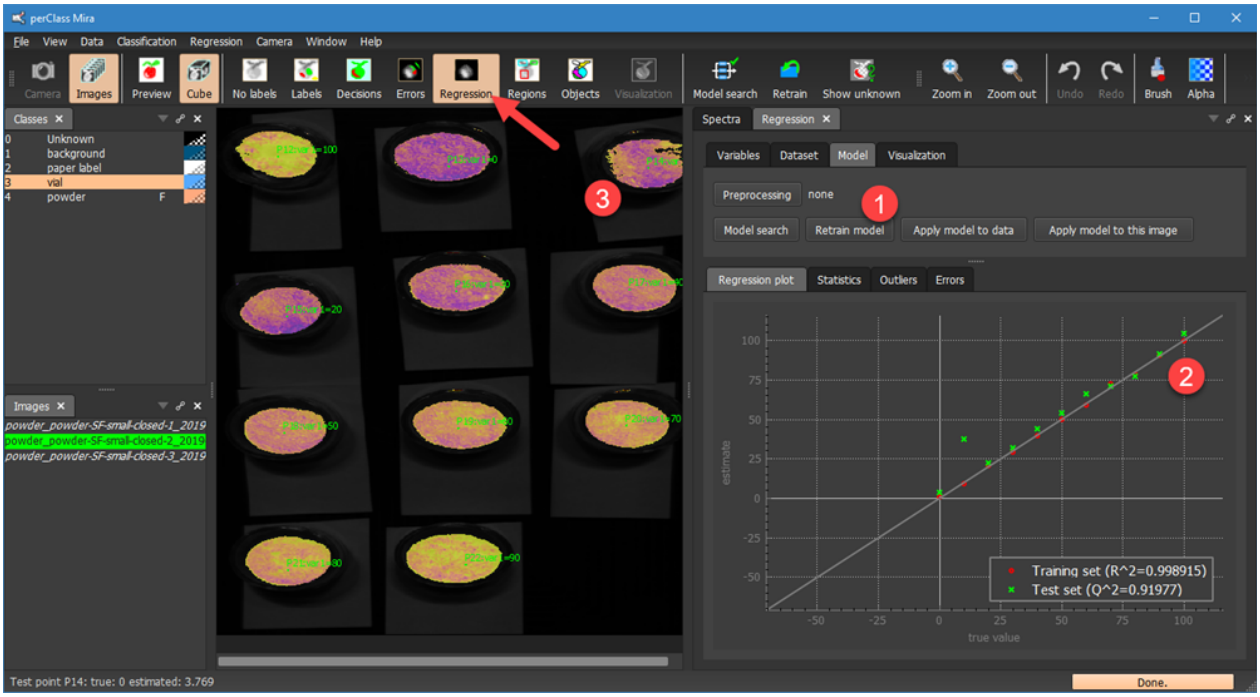
When we explore individual spectral bands, we can see that several first bands is highly noisy (not shown here) and the few last bands contain strong image artifacts:



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



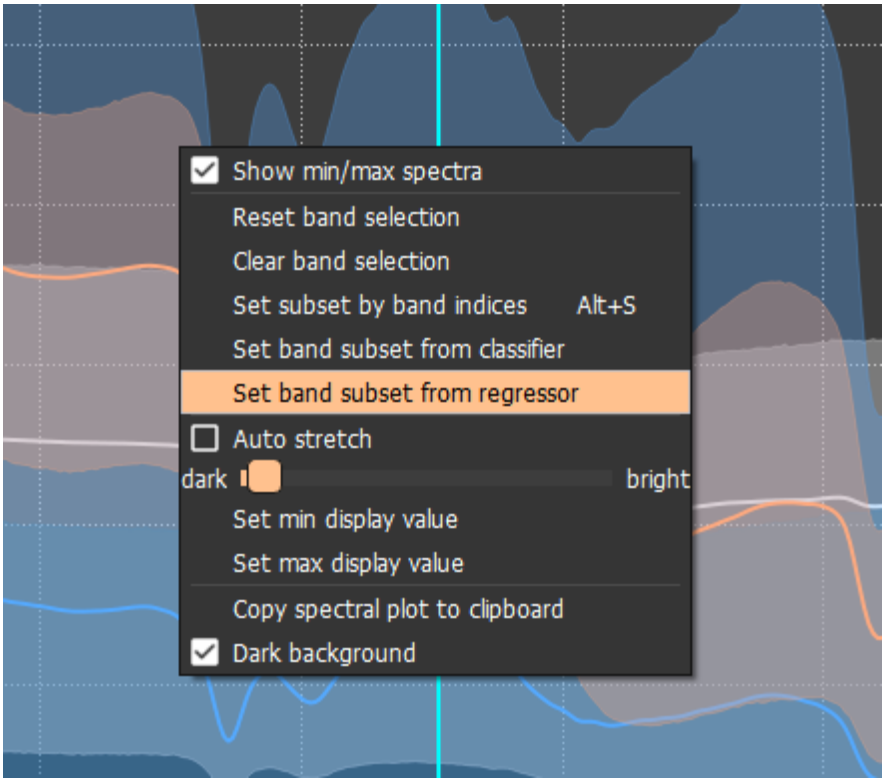
and retrain the regression model using the *Retrain* button (1). We may observe that the *Regression plot* shows improved performance (2) and pixel regression output (3) lacks any artifacts.



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.



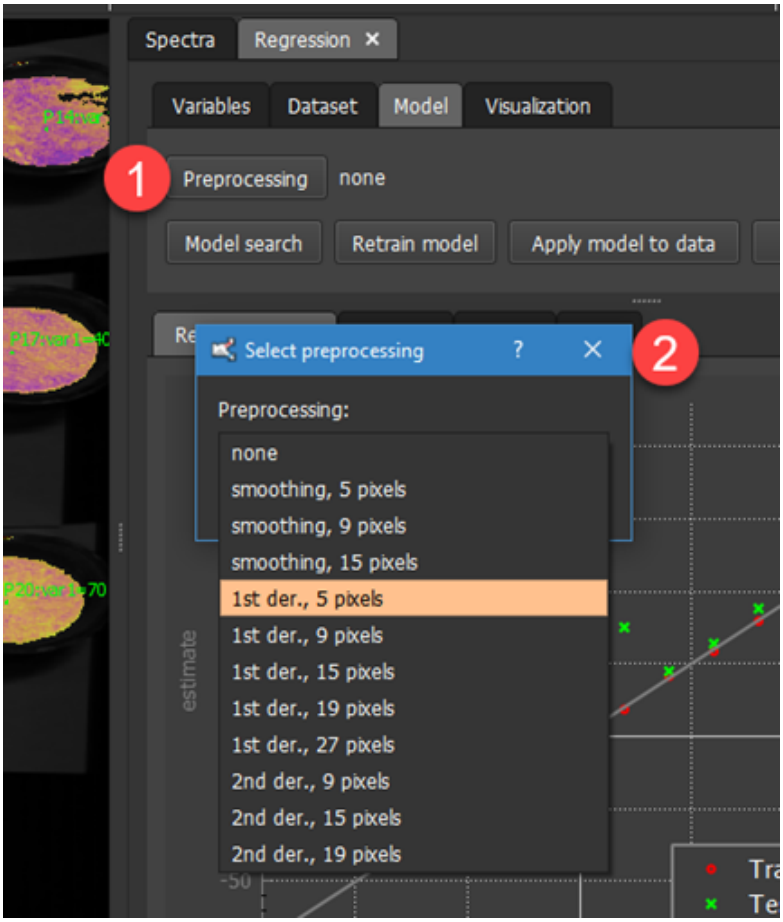
Preprocessing

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:

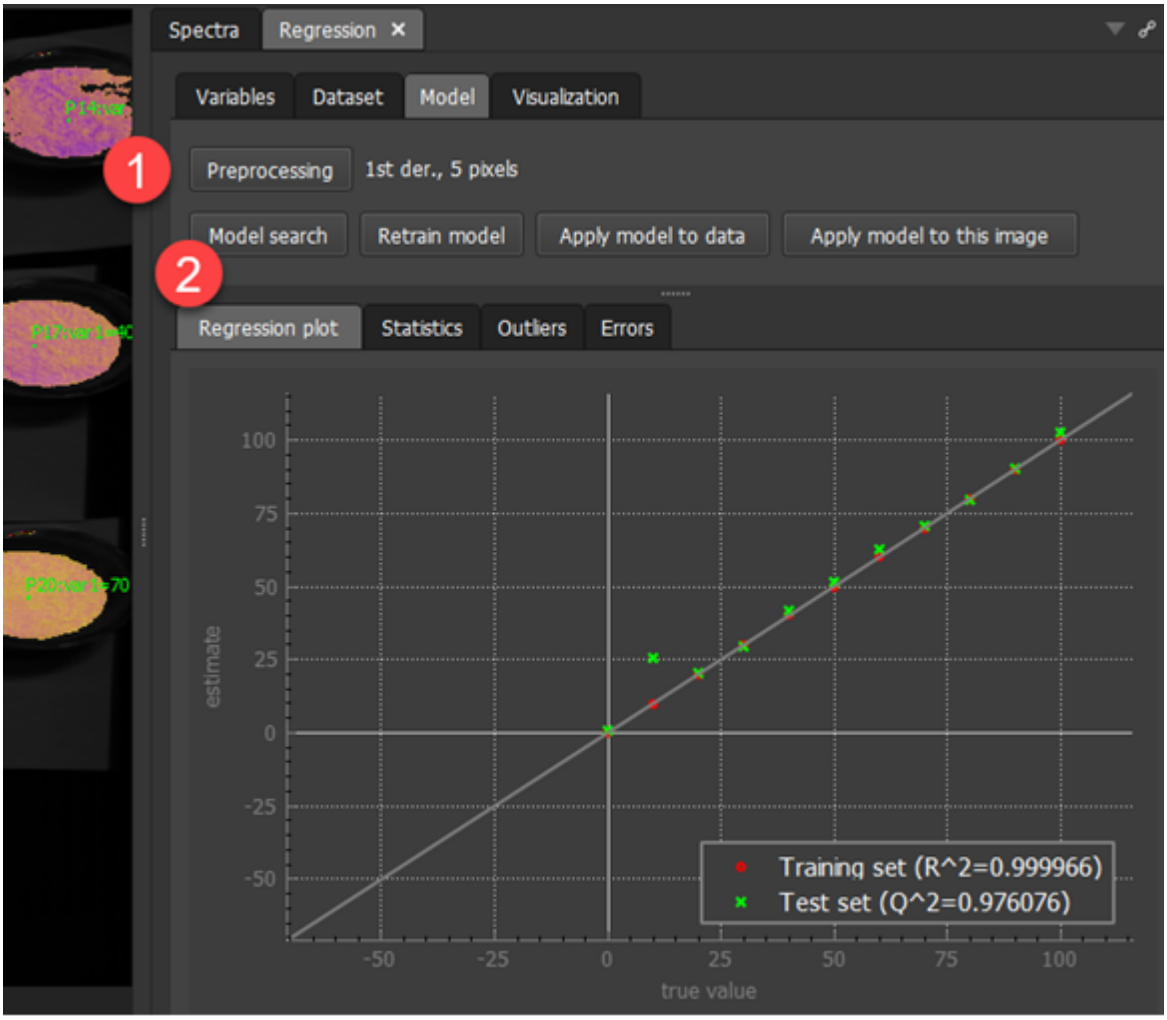
- Smoothing
- 1st derivative
- 2nd derivative

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

Preprocessing is set using the *Preprocessing* button 1 in the *Model* tab. When selected a dialog 2 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** 2. 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.

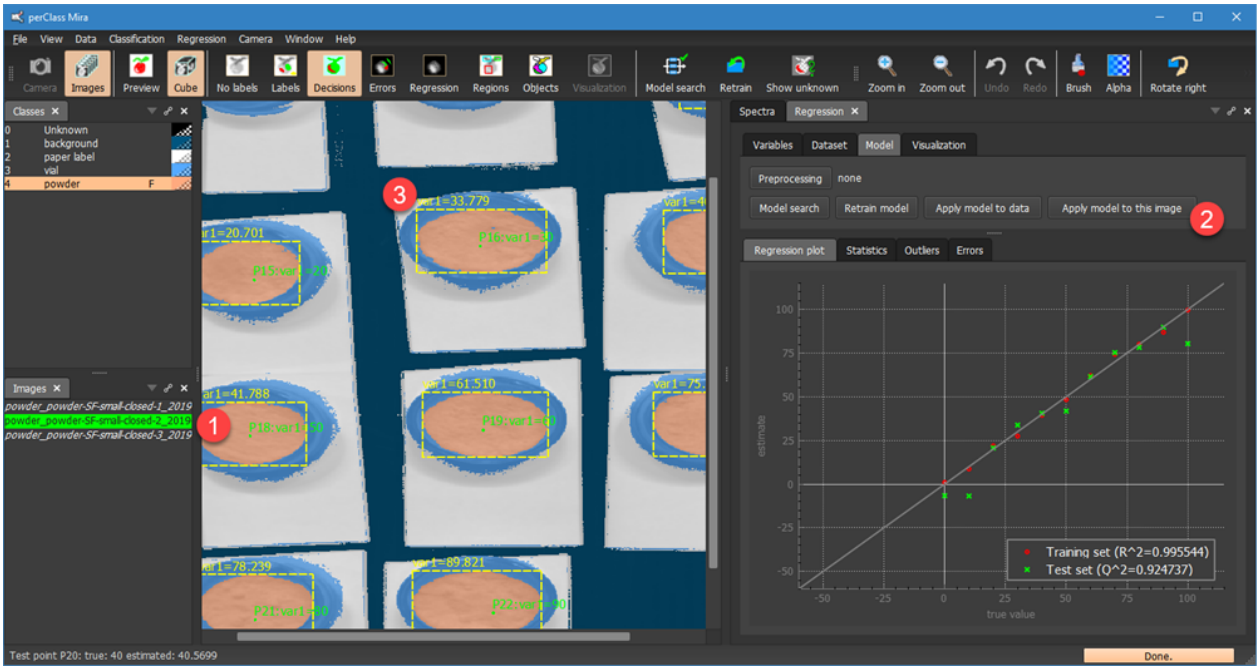
The *Model search* employs search for optimal complexity followed by model trainig. The Retrain command only retrains the model using the last selected modeling scheme.

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

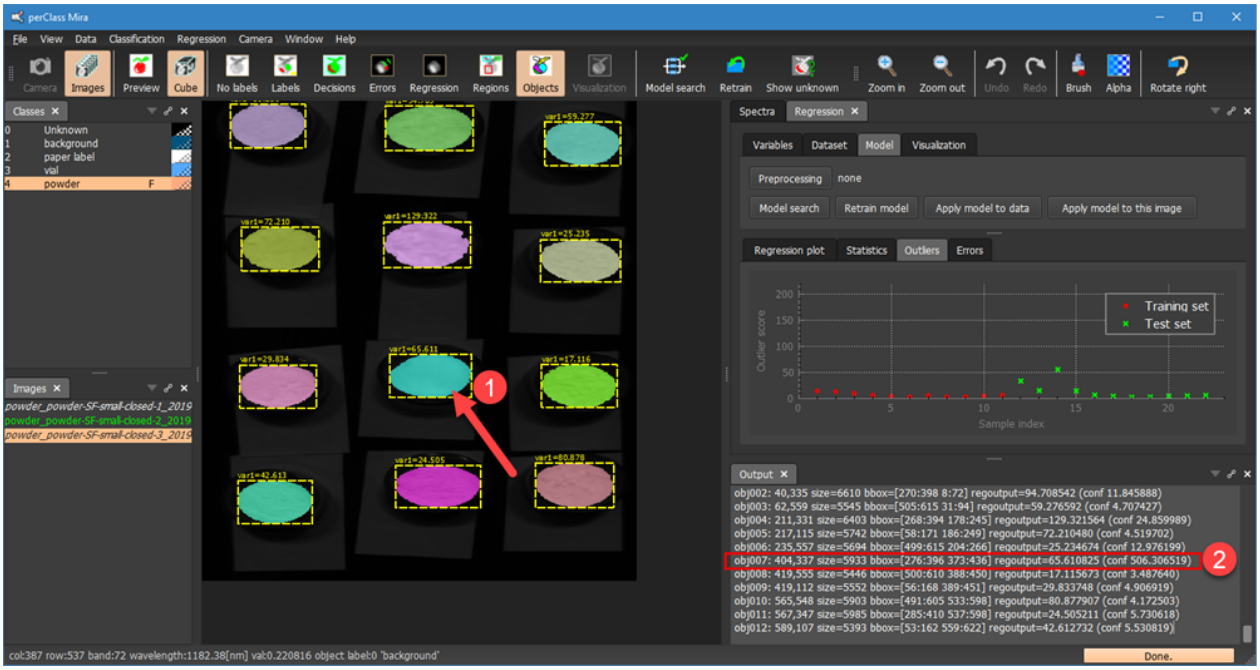
The regression model can be applied to any scan (with or without ground truth) by selecting the scan 1 and pressing the *Apply model to this image* button 2 in the *Regression* / *Model* tab. The entire processing pipeline is applied to the scan. This means, that the pixel classifier produces decisions, objects

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



The *Output* panel shows detailed information on regression output. For each object a cetroid, size, bounding box and the regression output 1 are given. In addition, an outlier score 2 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 teh *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 1 :

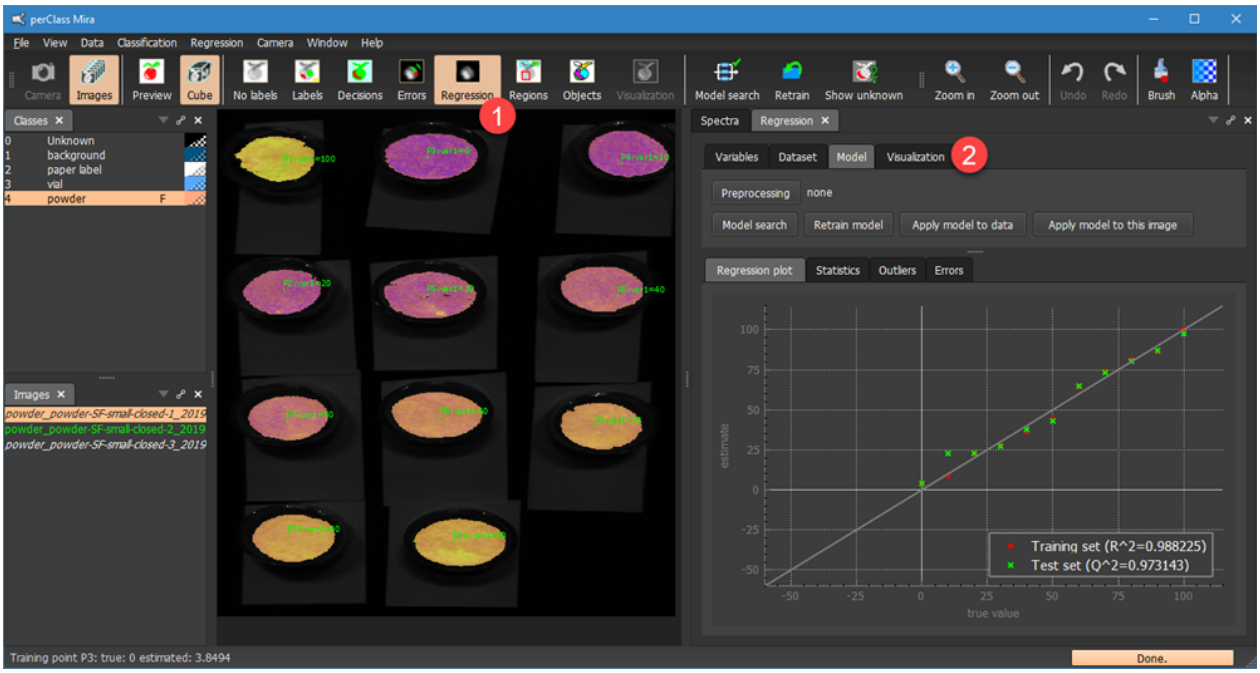


The corresponding object 007 information is ightlighted with 2 shows that the outlier score is >500.0 while the other samples show the range of 3.0-12.0.

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

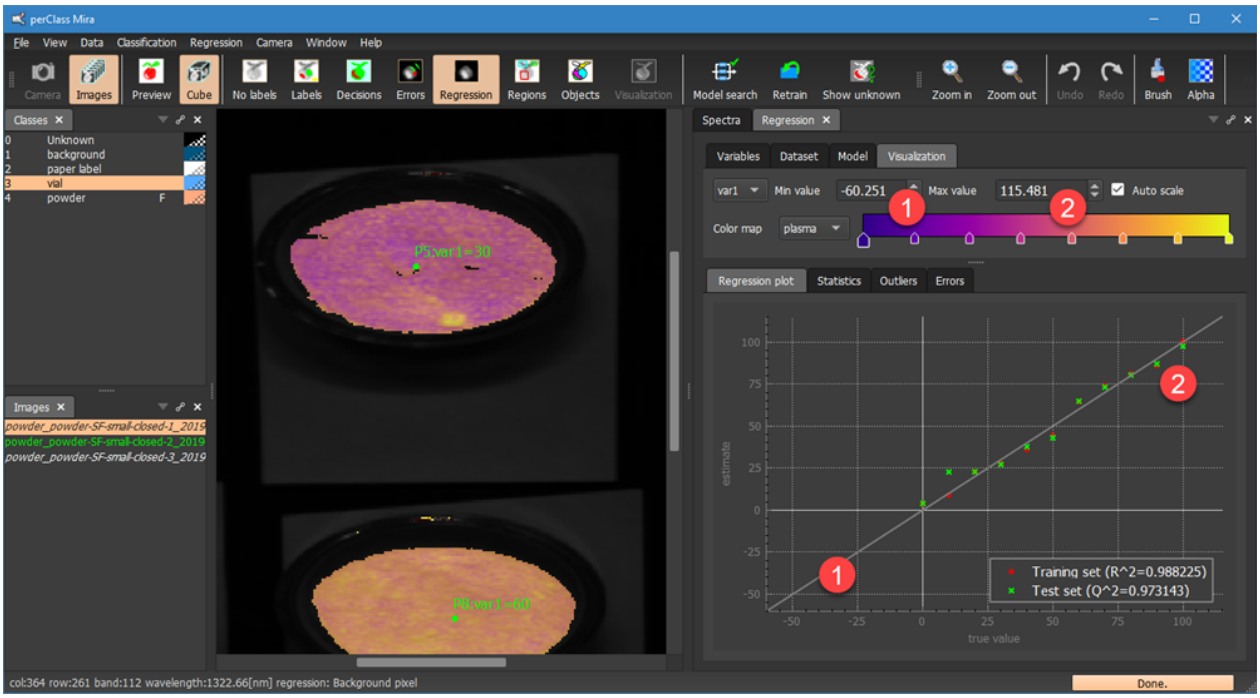
Regression toolbar button 1
The pixel visualization of regression output allows us to understand inhomogeneities within the objects and spatial distribution of the modelled phenomena.



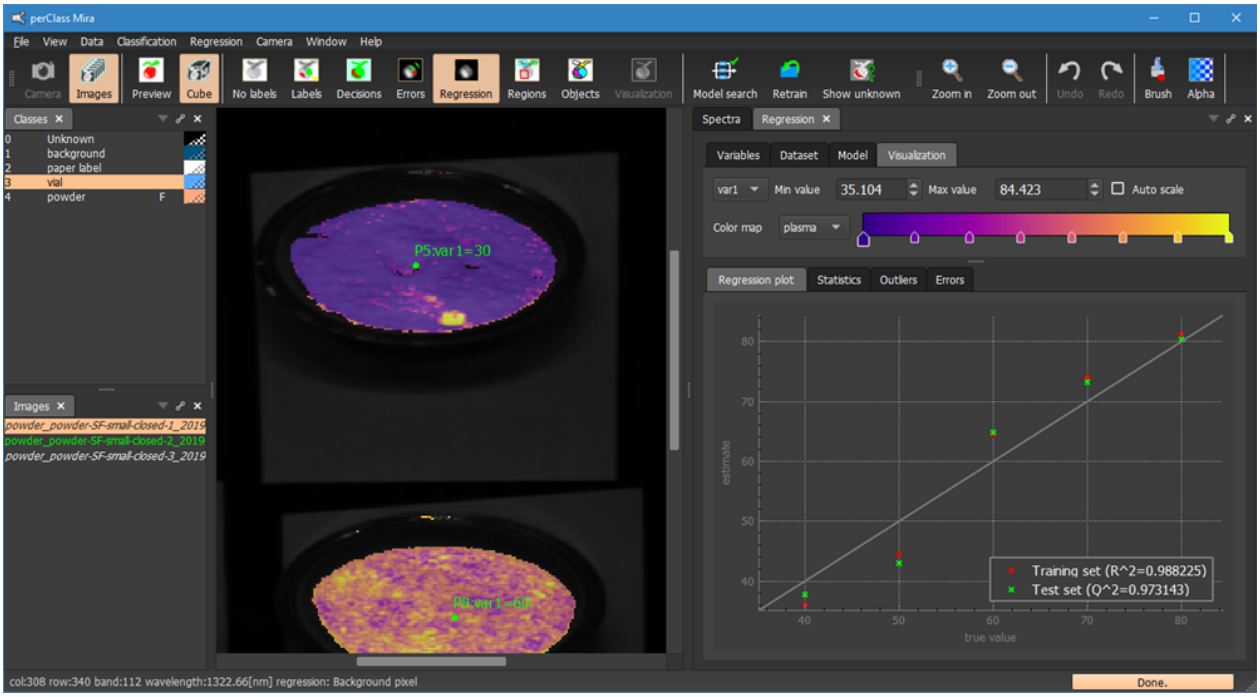
The display shows only the foreground pixels. The *Visualization* tab 2 provides number of options to fine-tune the rendering of the estimated output.

In the following example, we stretch min (1) and max (2) 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, respectivelly:

Before stretch:



After stretch:



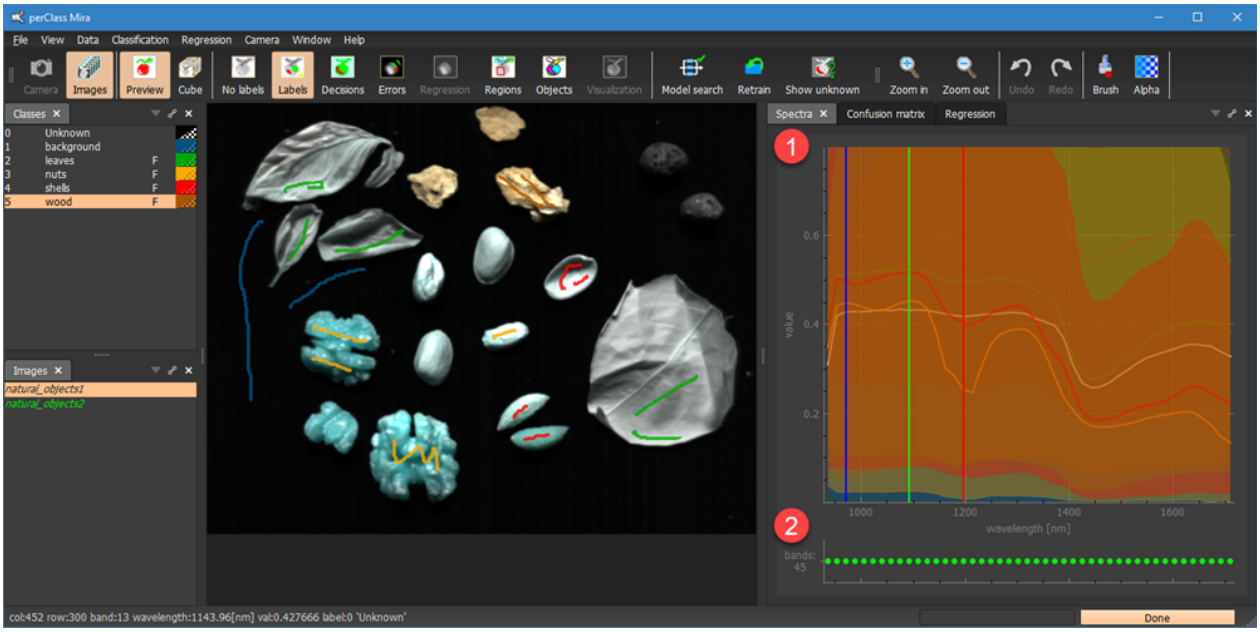
Note, that the floating point regression output does not change, we only affect its rendering via the specified colormap and visualization settings.

For more information on colormap control, see the description in the visualization section.

Spectral plot

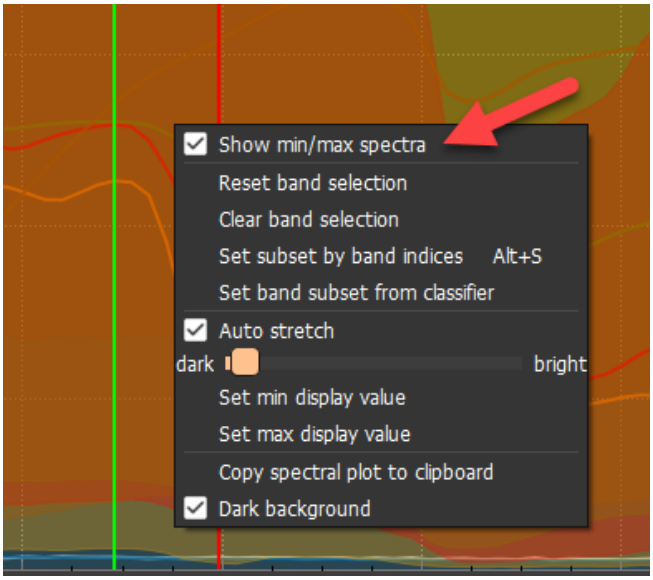
Spectral plot (1) displays information on labeled data across all spectral bands. The horizontal axis shows wavelength in nm. The vertical axis the pixel value. For the sake of image interpretation it is recommended to use data converted into reflectance. In perClass Mira, specific project types correct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

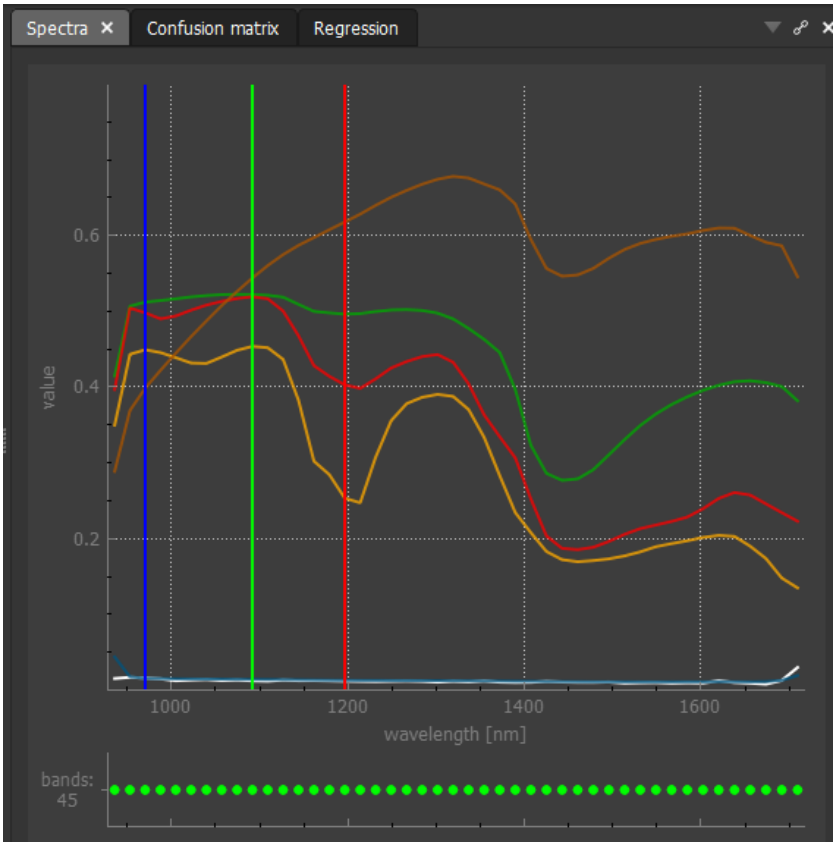
The band widget (2) under spectral plot shows individual spectral bands available in all images currently loaded in project. perClass Mira requires that all images use the same spectral bands/wavelength definition.



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:





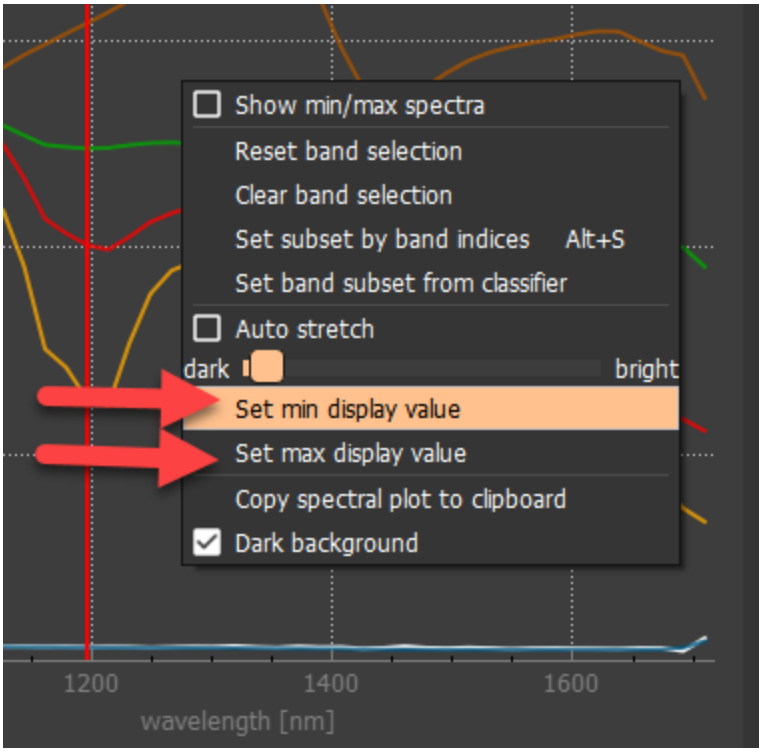
Display range and scaling

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.

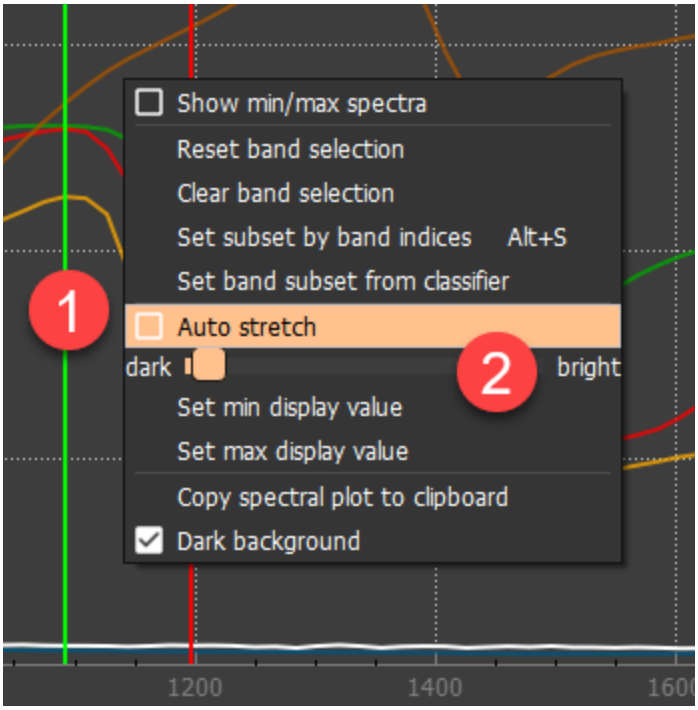
We may interactively adjust the bounds of the display value. This can be done either via the mouse wheel when the mouse pointer is close to top/bottom of the spectral plot or by a click and drag operation.

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



Alterrnatively, 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" meaning ful content in each image irrespective of its overall brightness or darkness. However, it is important to keep in mind that a the same gray value or color in two images represents different raw spectral or refelctance values.

The *Auto stretch* mode is enabled by the checkbox 1 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 2 in the context menu.

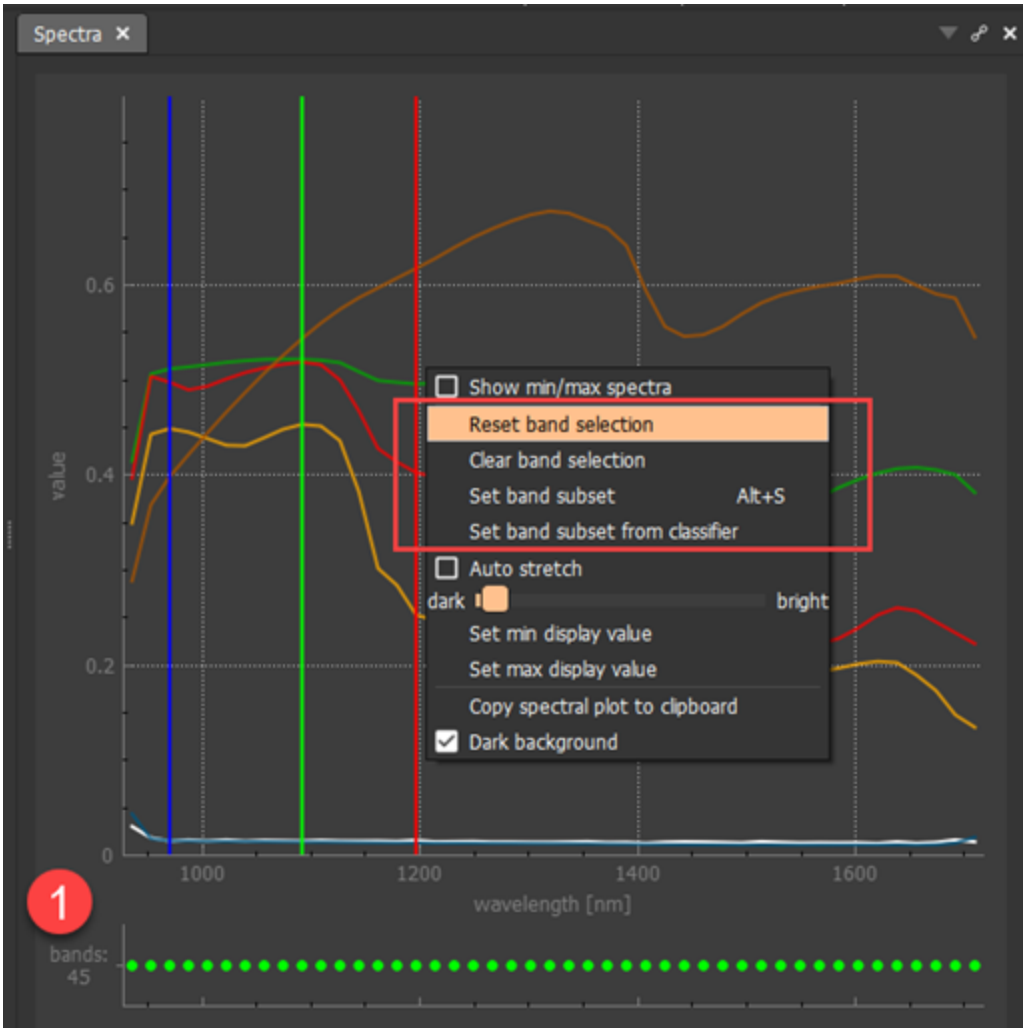


Adjusting display manually disables the *Auto stretch* function.

Band selection

Under the spectral plot, you may find the band widget 1 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.

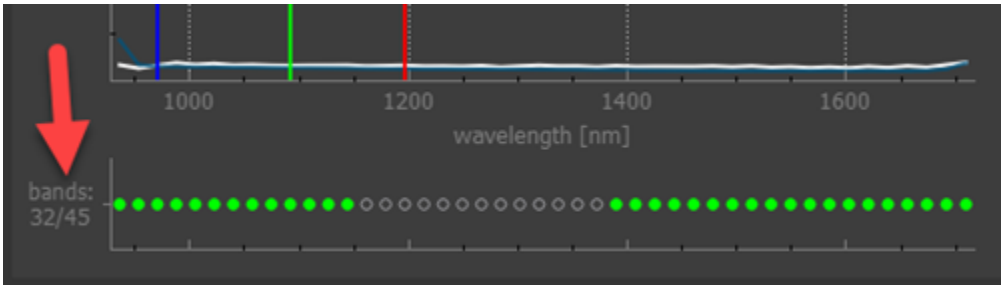
Spectral plot provide several commands allowing us to effective define and work with band subsets. Some of these are highlighted by the red rectangle in the context menu screenshot:



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.

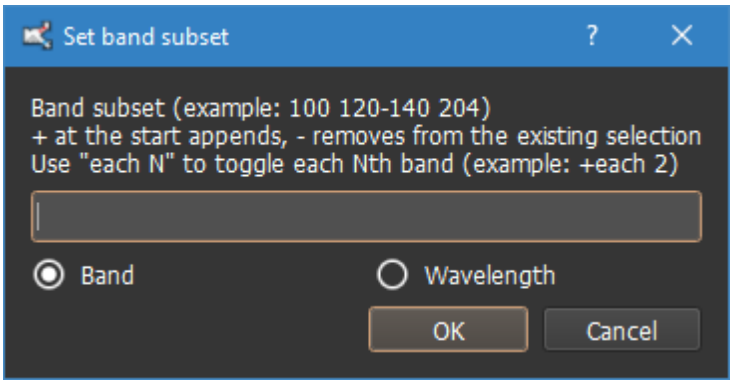
Note, that the number of selected bands and the total number of bands are provided to the left of the band widget:



Band selection by commands

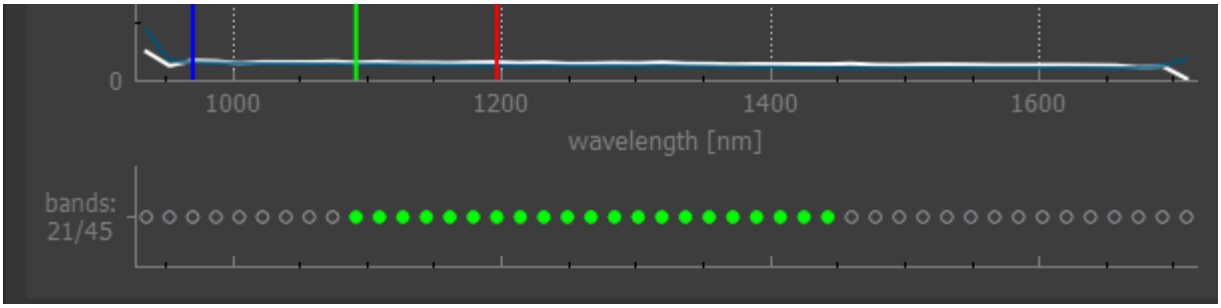
We may select and deselect all bands using the *Reset band selection* and the *Clear band selection* commands in the context menu.

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:

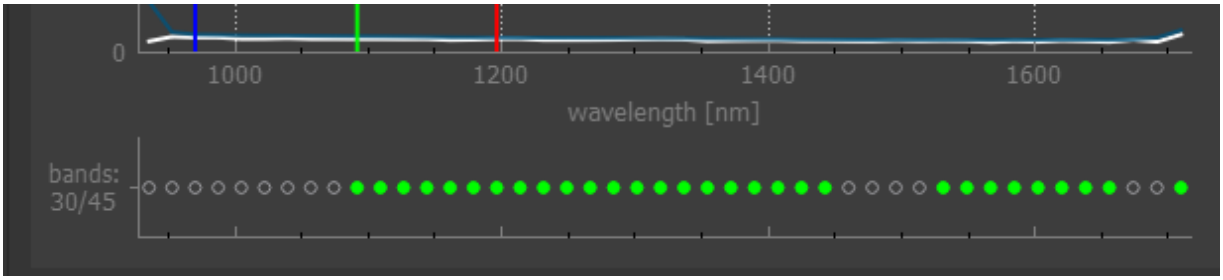


By default we may specify *band* indices. Alternatively, by selecting *Wavelengths* the selection happens on wavelength values in nanometers.

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:

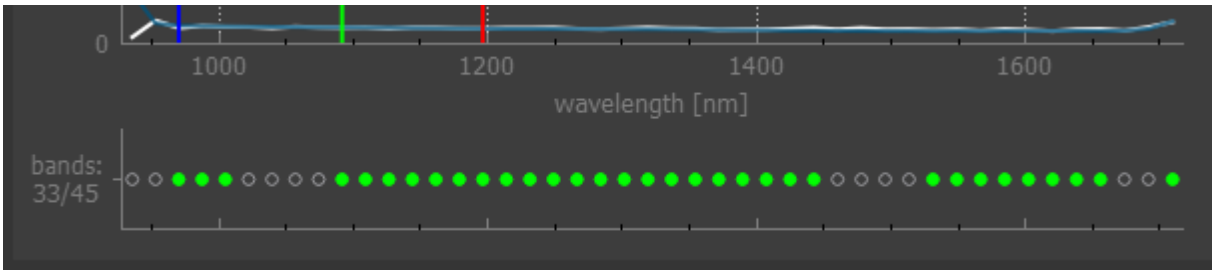


We may provide multiple regions at once separated by spaces. For example "10-30 35-42 45" will lead to these three regions:

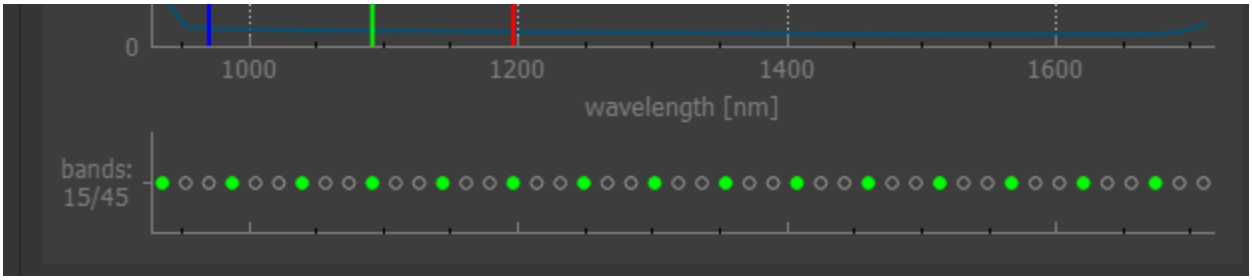


The default operation is to only enable the specified bands removing all previous state. Sometimes, we may wish to **add** or **remove** bands from the current selection. This is possible by prepending our specification with + (for adding) or - (for removing):

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:

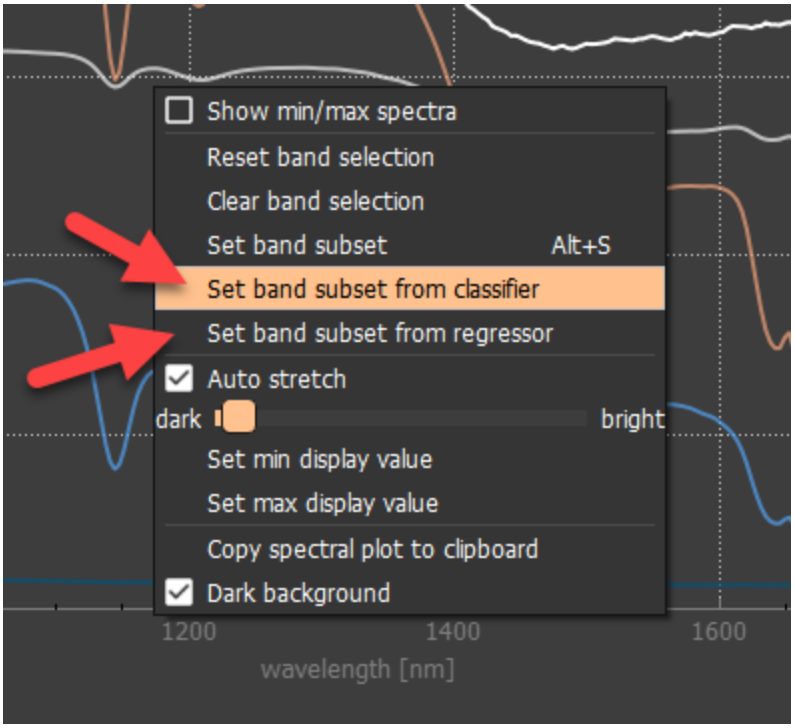


Band subsets used by models

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

Subseqent 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:

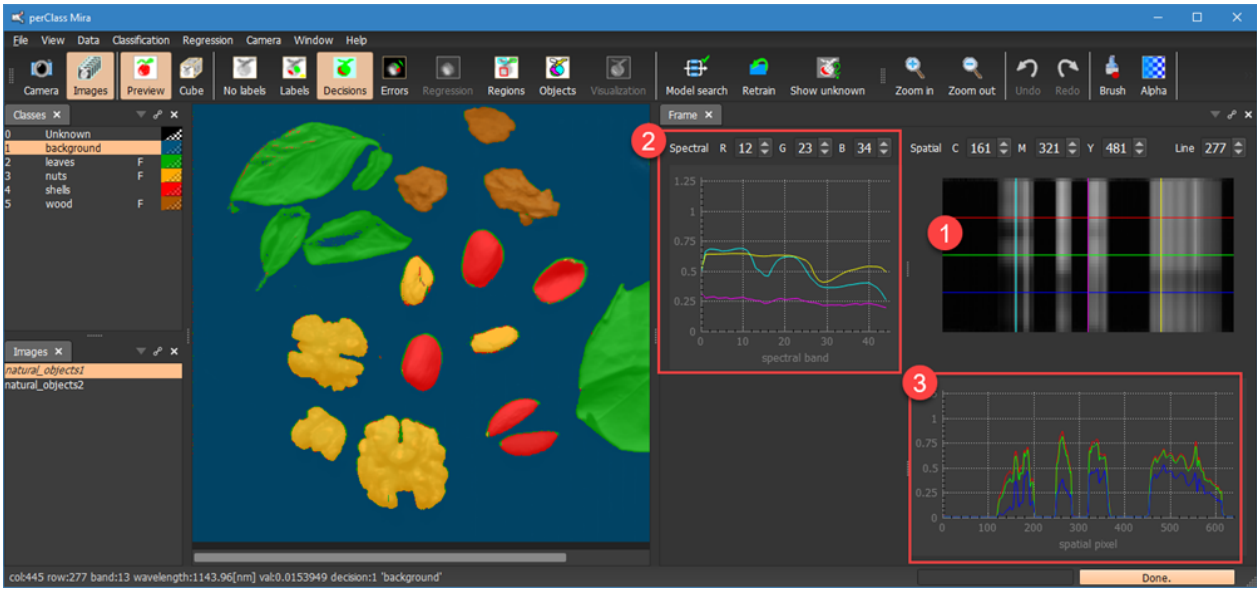


Frame panel

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 transfered from a camera i.e. spectral frames. Each frame provides spectral responses for a line of spatial pixels. Frame widget is useful during data acquisition as it helps us 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 cubes in line-scan use-case (BIL data layout in ENVI format). 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 1 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 2 and the spatial plot. The corresponding band and pixel indices are located on the top of the frame panel.

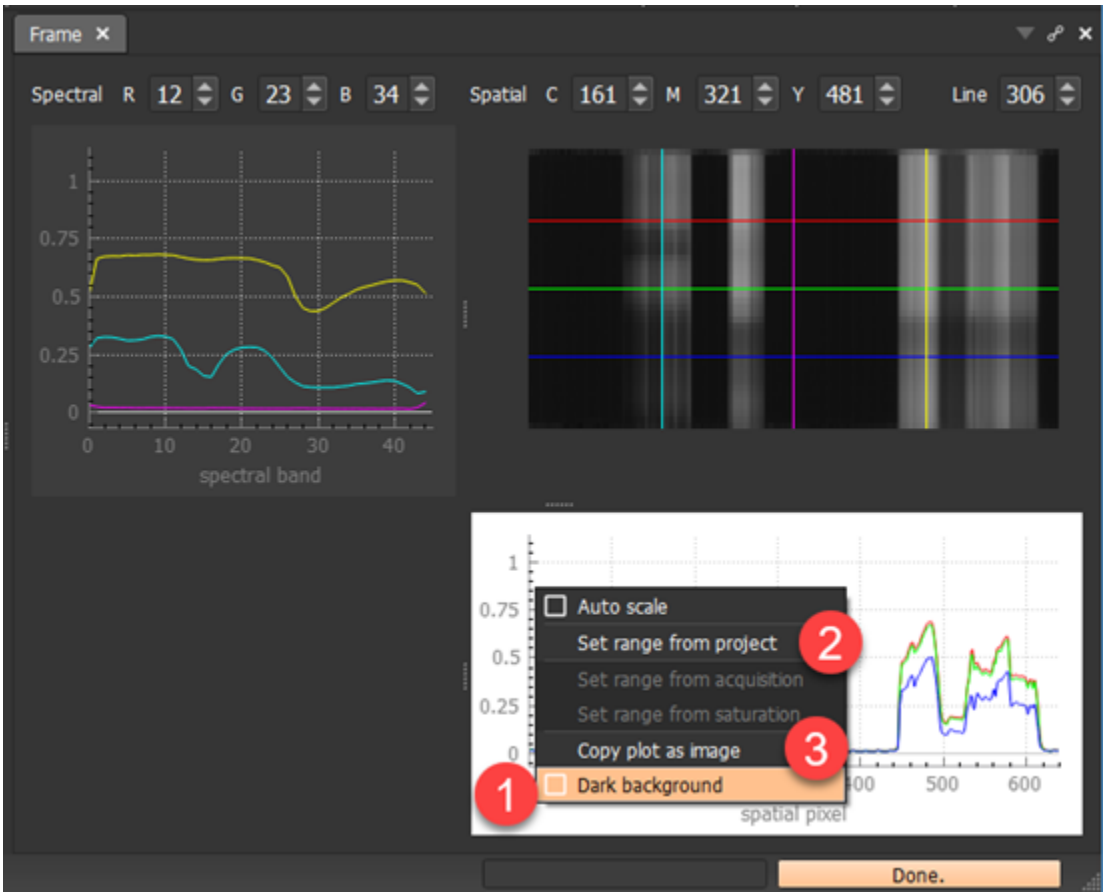


Context menu on the plots exposes number of options:

1. Switching between default dark and white background. White background may be beneficial when using the spatial profile plot to judge camera focus on a structured pattern

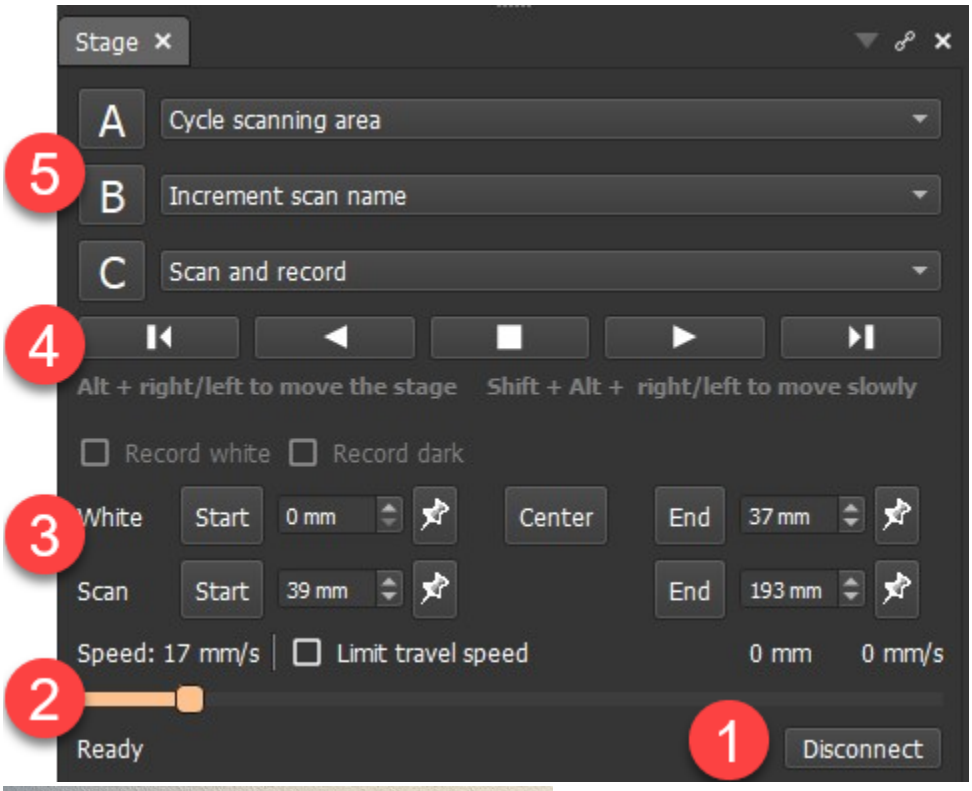
2. Several options for plot scaling. The *Auto scale* provides automatic stretch based onthe data. *Set range from...* options allow the user to set axis span from project, acquisition (frame) or saturation.

3. The plots can be copied to clipboard as an image



Stage panel

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



Connection

The stage needs to be connected to perClass Mira instance using the *Connect* button 1 . This button is changed into *Disconnect* when the stage is connected.

Status and speed

The lower section 2 of the *Stage* panel shows status information (Ready, Moving, ...), indicators of the current position and speed and the speed slider. The slider can be used to adjust the speed also during stage movement.

White and scan area controls

Above is located the control of white and scan areas 3 . For the white reference, we can set the start and end position. These are meant to represent the locations where the white reference is fully in view. For each position, there is

- A spinbox control allowing the user to edit position manually
- A pin button to use the current stage position
- and the button to move the stage to the set position

The *Center* button in between white start and white end controls allows the user to move the stage to the center of the white reference.

The scan area start and end can be set with the respective control. This is useful when samples do not cover the entire 400 mm of the stage table.

By default, the stage returns to home position after the scan is acquired as quickly as possible. In some cases, the samples may be displaced by the fast movement. The *Limit travel speed* checkbox serves in such situations to use slow speed also for this return movement. It is off by default.

Movement controls

The movement control 4 is located in the middle of the *Stage* panel. The buttons allow user to perform all basic movements such as:

- Go to start / end
- Move right / left
- Stop

The *Move right / left* buttons need to be pressed and held to continue the movement.

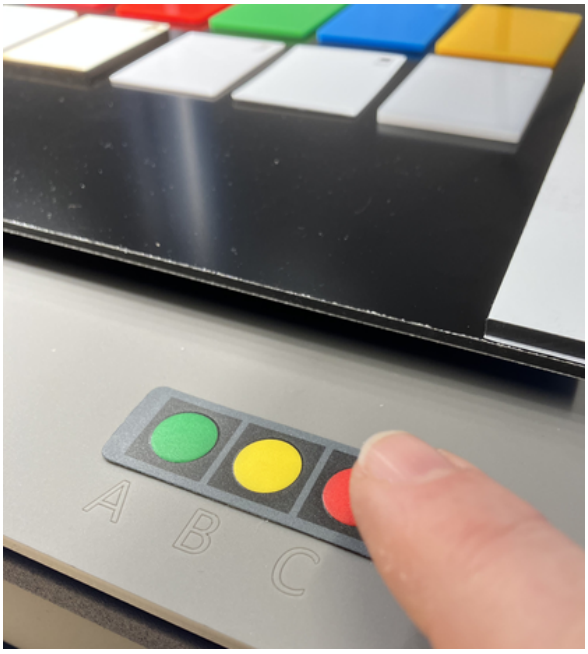
TIP Note the label dispaying keyboard shortcuts for the stage movement: *Alt + right / left* for normal movement and *Shift + Alt + right / left* for slow movenent. These shortcuts provide the easiest way to control stage position.

Stage buttons

Finally, in the top area of the *Stage* panel, we find the stage button configuration 5 . These buttons can invoke user-configurable commands.

User-defined stage buttons

perClass Stage comes with three user-configurable buttons: Namely, the green **A**, yellow **B** and the red **C** buttons.



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 sampels 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**
- **Increment scan name** - Increment index in the currently focused scan name part (underscore-separated parts, ending with number). See more information on storing meta-data information in scan names.

TIP Camera buttons are mapped to the numerical keypad keys -, + and Enter. This makes it easy to invoke button commands without looking at the keyboard or screen. Mnemnonics: the mapping is top-to-bottom in the same way as the A,B,C button orger in the *Stage* panel

Commands set by the user are restored in future sessions.

Slightly different command sets are used in *Camera* **scan mode** and **belt / waterfall mode**.

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.

The *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 miliseconds. Exposure-control of the camera is assumed where exposure settings influences possible frame rate settings.
- 2 Frame rate in frames per second.If framre 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.
- 5 Indicator of dropped frames. In cases, where the data processing cannot keep up with the camera acquisition, frames may be dropped. This counter is reset by starting each acquisition. Note, that some cameras may be set to buffer frames that cannot be processed internally. In such setups, frame dropping occurs only when the internal buffers are filled.

Adjusting scan quality

perClass Mira provides several simple-to-use tools that allow the user to quickly adjust the scanning process to produce high-quality scans.
These include:

- Adjusting focus
- Auto-exposure maximizing dynamic range of the scans
- Making sure line-scan mages provide square pixels

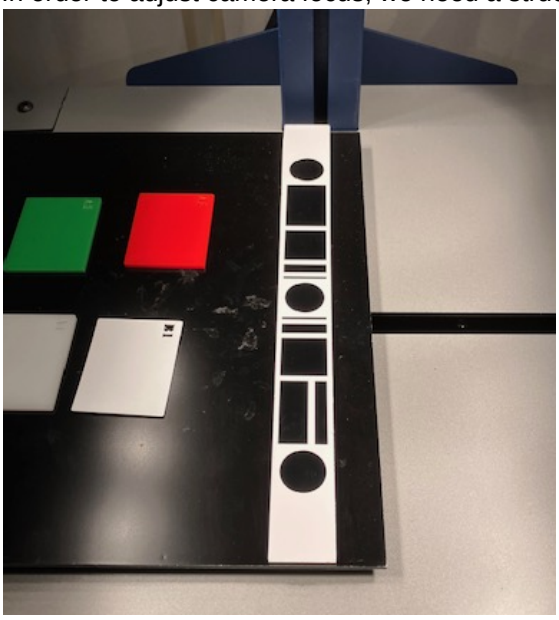
These tools are accessible from the respective tabs in the *Camera* panel

When using perClass Mira Stage, these tools rely on the white reference bar with a structured pattern

printed on its opposite side. It is possible to use the quality tools with custom white references or structured patterns.

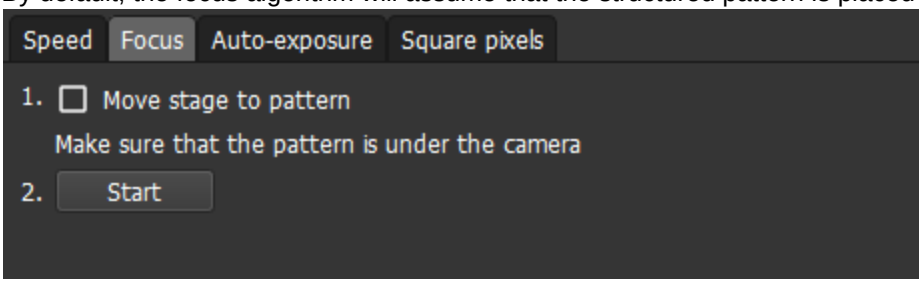
Optimizing focus

In order to adjust camera focus, we need a structured pattern showing sharp edges.

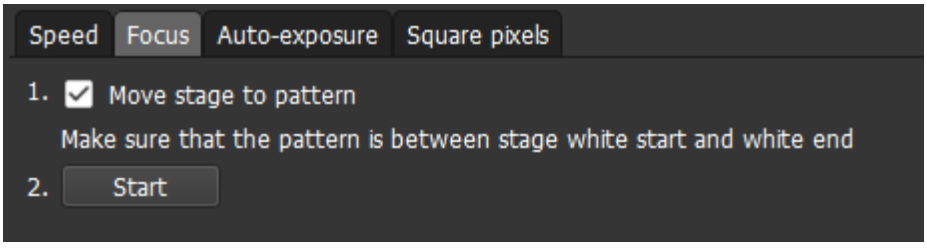


Starting the focus adjustment

By default, the focus algorithm will assume that the structured pattern is placed under the camera.



If we have already setup white reference start and end positions in the *Stage* panel, we may also enable the *Move stage to pattern* checkbox. When pressing *Start*, the stage will first move to the pattern before applying the focus processing.



Adjusting the focus

When we click *Start* button in the *Focus* tab, the semi-automatic focus adjustment algorithm will start. It is semi-automatic, because it relies on the user to adjust camera focus (or camera height) to provide direct feedback.

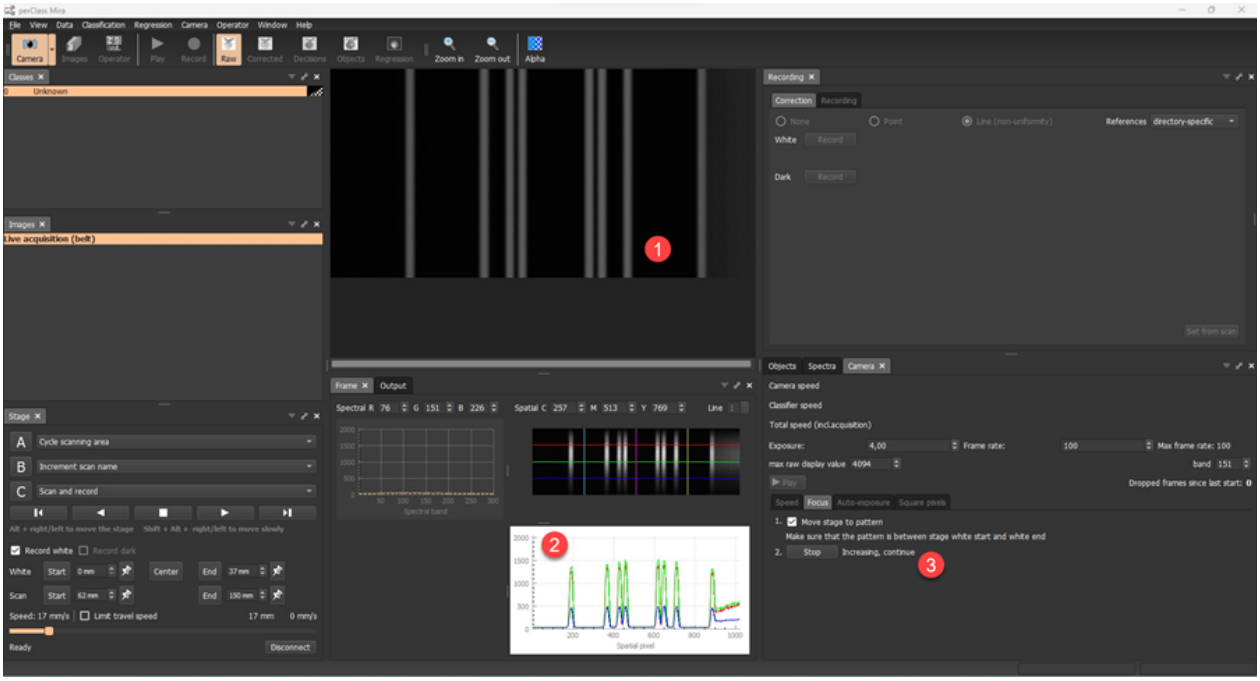
Recommended screen setup, when adjusting focus, is depicted by the following screenshot. The camera is out-of-focus, as we can see in the live view (1). It is recommende to view also the *Frame* panel and enable white background in its bottom, spatial, plot (2) via the right-click context menu. The focus

adjustment algorithm provide user-feedback through messages displayed next to the *Start / Stop* button 3 .

At the start of the session, the algorithm needs few seconds to estimate signal characteristics. Then, the user may start adjusting the camera focus. The feedback will be privided to either continue turning the focus ring of the objective lens in the same direction, or go back. Eventually, it will be indicated that the optimal focus is reached.

TIP The spatial plot of the *Frame* panel provides additional visual feedback to the focusing process.

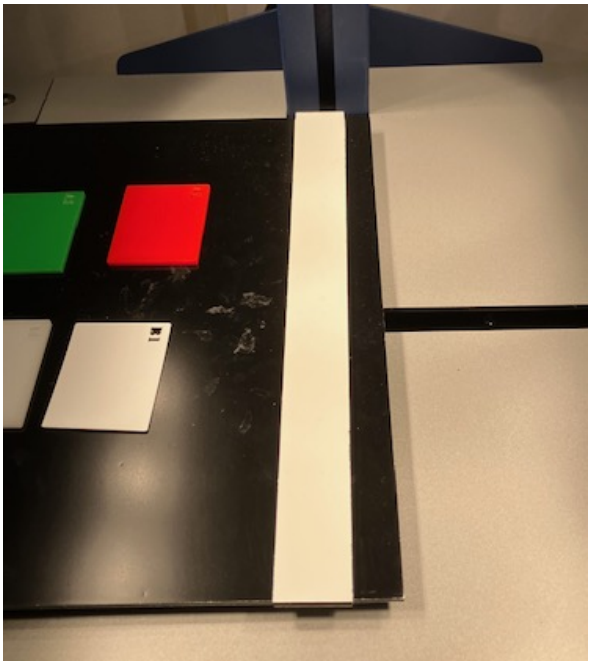
The user may also interrupt focus adjustment any moment with the *Stop* button.



Auto-exposure

Auto-exposure is used to maximize camera exposure for the given illumination conditions. In this way, the dynamic range of the resulting data is optimized.

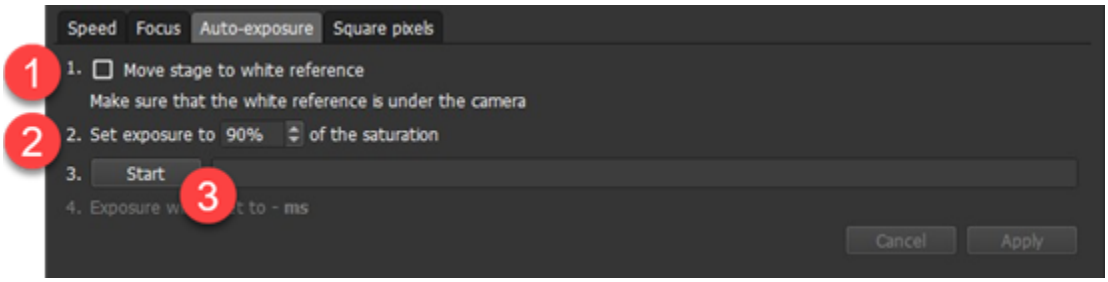
The auto-exposure tool expects white reference.



Starting the auto-exposure tool

By default, the auto-exposure tool assumes that the white reference is placed under the camera. Enabling the checkbox (1), the stage will first move to the center of white reference, as defined by the white start and end positions in the *Stage* panel.
The auto-exposure tool will increase exposure until the maximum shite response reaches a specific percentage of saturation. By default, this is set to 90% in the spinbox (2). This is to avoid saturated data for bright signal.

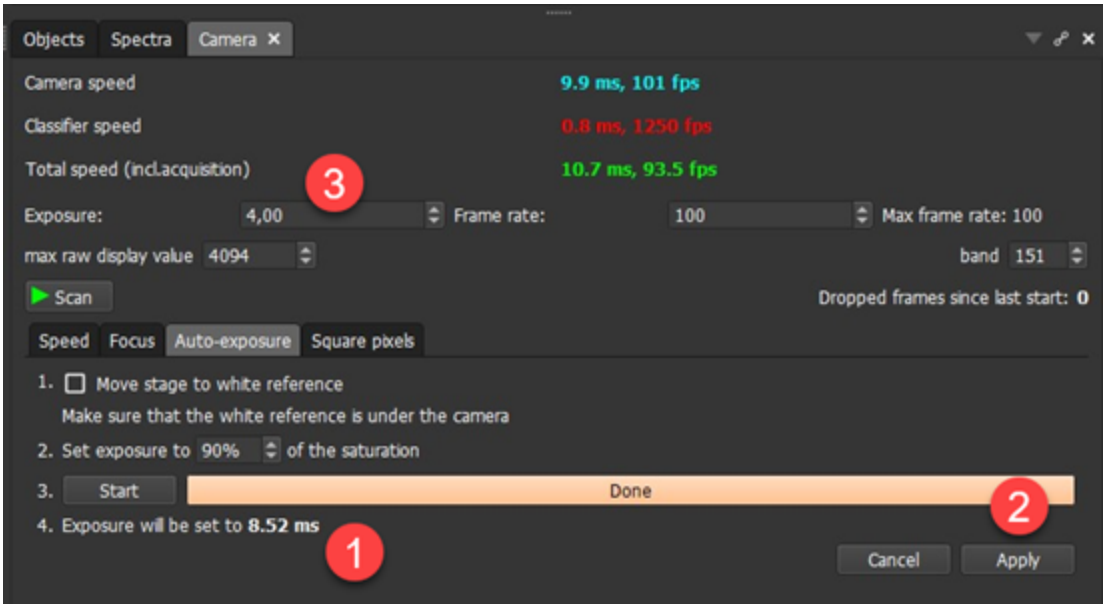
The auto-exposure process is started by the *Start* button (3).



Accepting auto-exposure value

After the auto-exposure process is finished, the tool provides the estimated optimal value (1). Note, that the user needs to explicitly confirm to use the value by clicking the *Apply button* (2). Without this confirmation or by clicking the adjacent *Cancel* button, nothing will be changed. When the user presses the *Apply* button, the estimated value will be entered in the *Exposure* edit field (3).

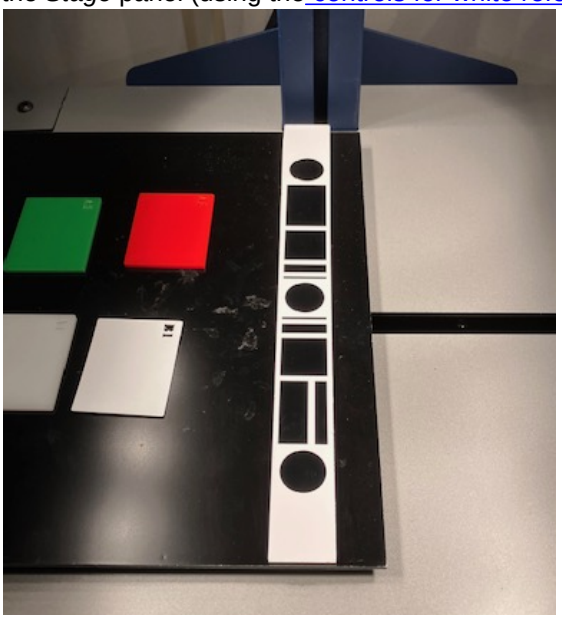
NOTE Adjusting camera focus will render existing dark and white references inapplicable. This is indicated in the *Recording* panel by red message next to the references. It is user responsibility to mak sure the references are re-acquired if exposure settings change.



Square pixels

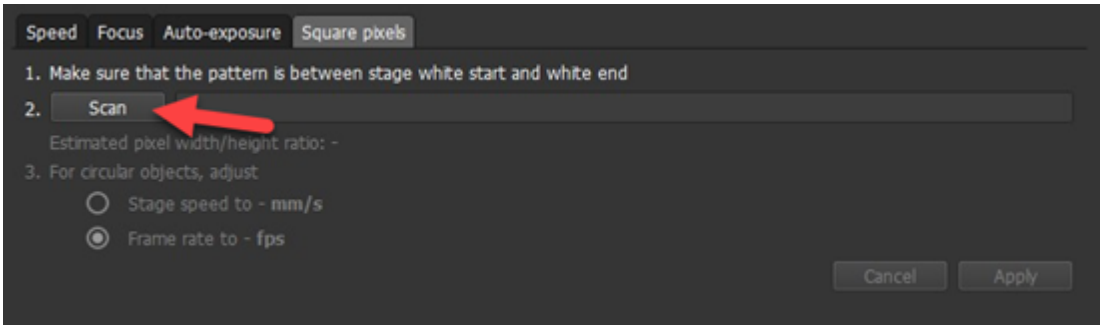
Square pixel tool allows us to optimize scanning speed of a line-scan camera system to assure identical true dimensions of image pixels in direction of movement and across the table/belt.

It assumes that a structured pattern is placed on the stage table and its start and end positions are fixed in the *Stage* panel (using the controls for white reference).



Starting the square pixel tool

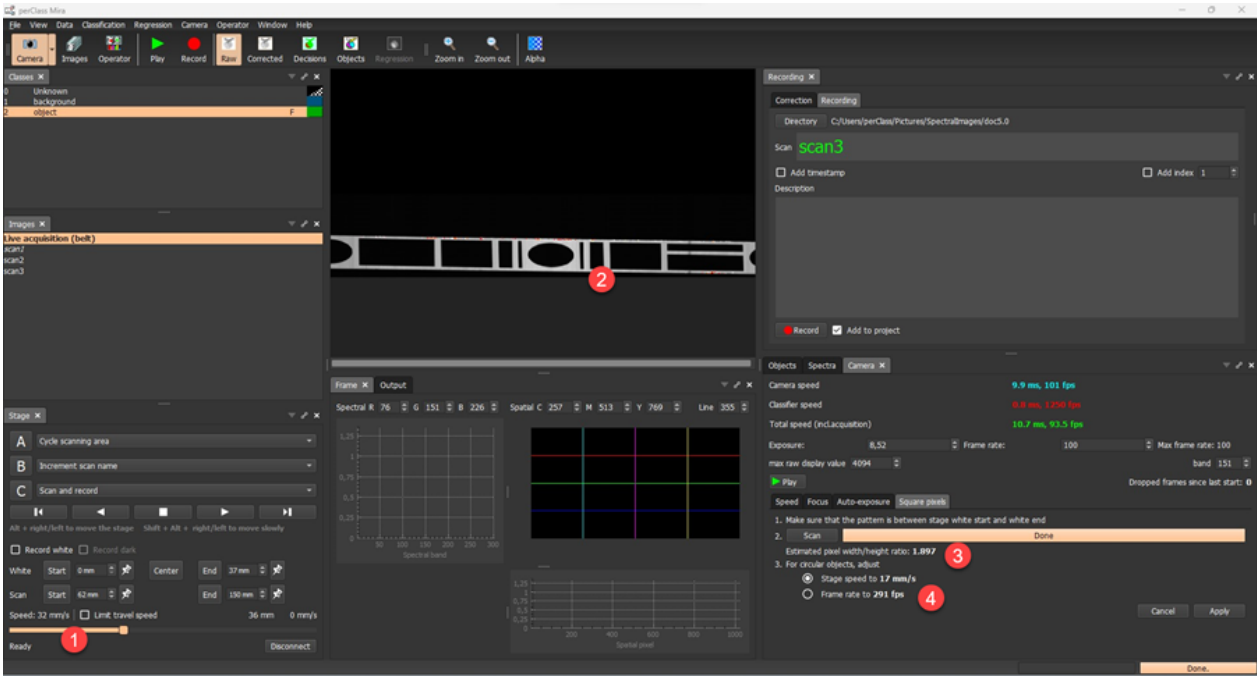
When the start and end positions of the structured pattern are adjusted in the *Stage* panel, press the *Scan* button.



Square pixel tool feedback

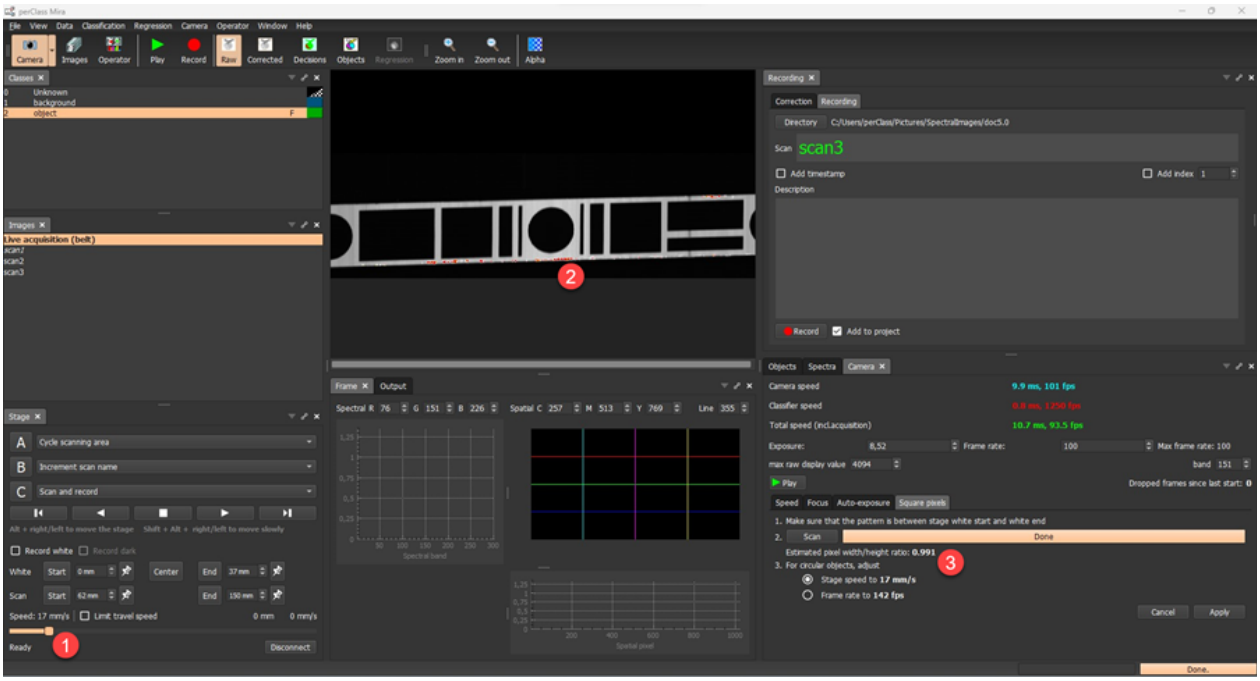
In the screenshot below, you can see that our stage speed was set to 32 mm/sec 1. The acquired structured pattern image 2 shows significant distortion of the circle shapes. The estimated excenicity is displayed at 3 and the proposed solutions in terms of optimal stage speed or camera frame rate at 4.

The user needs to choose the desired solution and explicitly confirm it by pressing the *Apply* button.



Confirming the correctness of the solution found.

When re-running the tool once more, we can observe (below), that the speed was changed to 17 mm/sec 1, which leads to proper circular shapes of the scanned pattern 2. The newly estimated excentricity 3 is now close to 1.0. We may either press *Cancel* or simply leave the *Square pixels* tool to keep this setting.



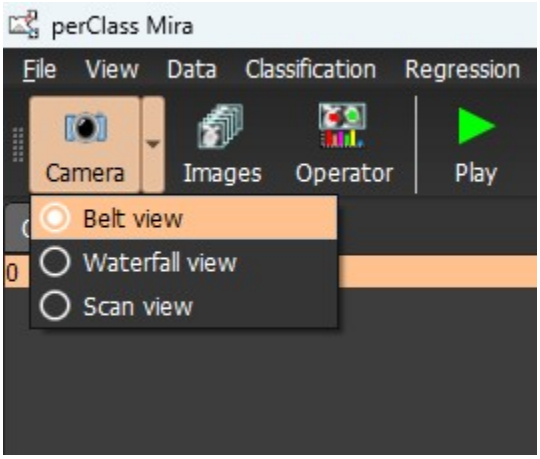
TIP Should the square tool not find the desired solution, it may be due to blur, low scan resolution or its inability to segment the structure pattern bar from the background. In such cases, you may use any other circle pattern printed in higher resolution.

Camera modes

In perClass Mira 5.0, camera may be operated in one of three modes:

1. **Belt view (default)** - The image stream is moving upwards simulating the view on top of a conveyor belt
2. **Waterfall view** - The image stream is moving downwards. When reaching the bottom of the screen, it starts again from the top.
3. **Scan view** - A pre-allocated scan buffer is filled with the data. When scanning stopps, the user may decide to save the data or discard it and re-take the scan

Camera mode can be changed by clicking small arrow next to the *Camera* toolbar button.



Apart from different visualization, the fundamental difference between these modes lays in the way how data is recorded.
In the **belt** and **waterfall** modes, the data from a line-scan camera is directly recorded to disk. In the **scan mode**, data is filled in a buffer and the saving to disk only happens when user decides to do so. This separation between acquisition and saving enables several new functionalities coming in perClass Mira 5.0 such as:

- **re-taking the scans** - this leads to cleaner currated data sets

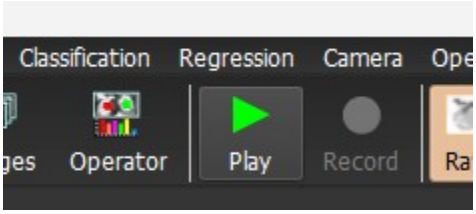
- **scan compression** - significantly loweing data storage requirements for large projects
- **high-speed acquisition of training data** (practically tested up to 2m/sec speed on an industrial belt)

Belt and waterfall mode

Belt and waterfall represent identical scan acquisition logis. They differ only in the way live data is visualized.

Data acquisition

When a camera is initialized, data acquisition can be started using the *Play* button and can be paused by the *Pause* button. Recording, started by the *Record* button happens directly to disk.



Live data visualization

In the **Waterfall mode**, the data moves downward. When reaching the bottom of the screen, data visualization restarts from the top of the screen. This approach can visualize per-pixel decisions but not per-object decisions, that are finalized after the object is already rendered.

In the **Belt mode**, live data moves upward simulating the top view of an industrial conveyor belt. In this view, perClass Mira can visualize both per-pixel and per-object decisions.

Due to their continuous nature, these modes enable live demonstrations of spectral data processing whether on the stage or industrial belts.

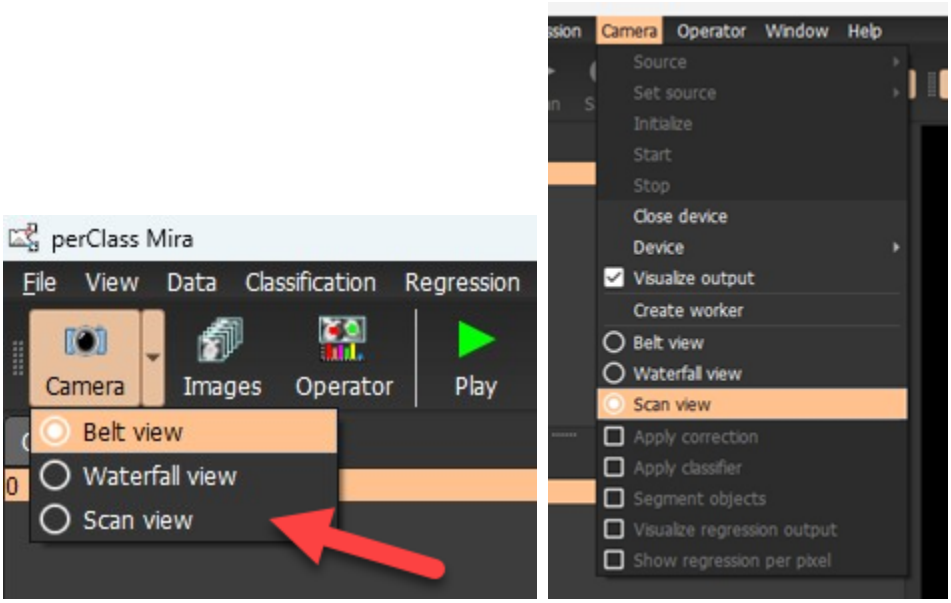
Scanning process on perClass Stage

In the belt and waterfall modes, the scanning and recording process are united. The user initiates *Scan and record* command that starts both the movement of the stage and recording to disk. Recorded scan cannot be "undone" as it is already stored in a file. In order to remove such as scan, the user needs to remove the corresponding files in addition to removing a scan from project with *Remove images* command in the context menu of the *Images list*.

Scan mode

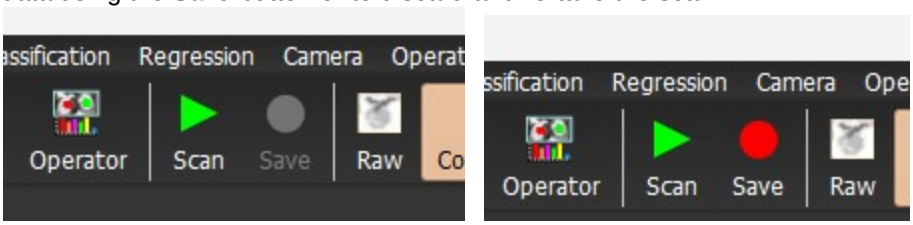
The Scan mode separates acquisition of a scan from its saving to disk. This makes line-scan cameras operate in a similar fashion to snapshots and standard machinve vision systems.

Scan mode can be enabled by clicking the small arrow adjacent of the *Camera* button or by the *Camera* menu:



Data acquisition

When a camera is initialized, data can be acquired by clicking the *Scan* toolbar button or the *Scan* button in the *Camera* panel. Data is read into an internal buffer. The user can then decide whether to save the data using the *Save* button or to discard and re-take the scan.



Live data visualization

The **scan mode** shows data acquisition only during the scanning, not continuously like the **belt** or **waterfall** modes. Note, that scan mode is not very useful for live demonstrations as it stops when the internal data buffer is filled. For continuous demonstrations, use **belt** or **waterfall** modes.

Scanning process on perClass Stage

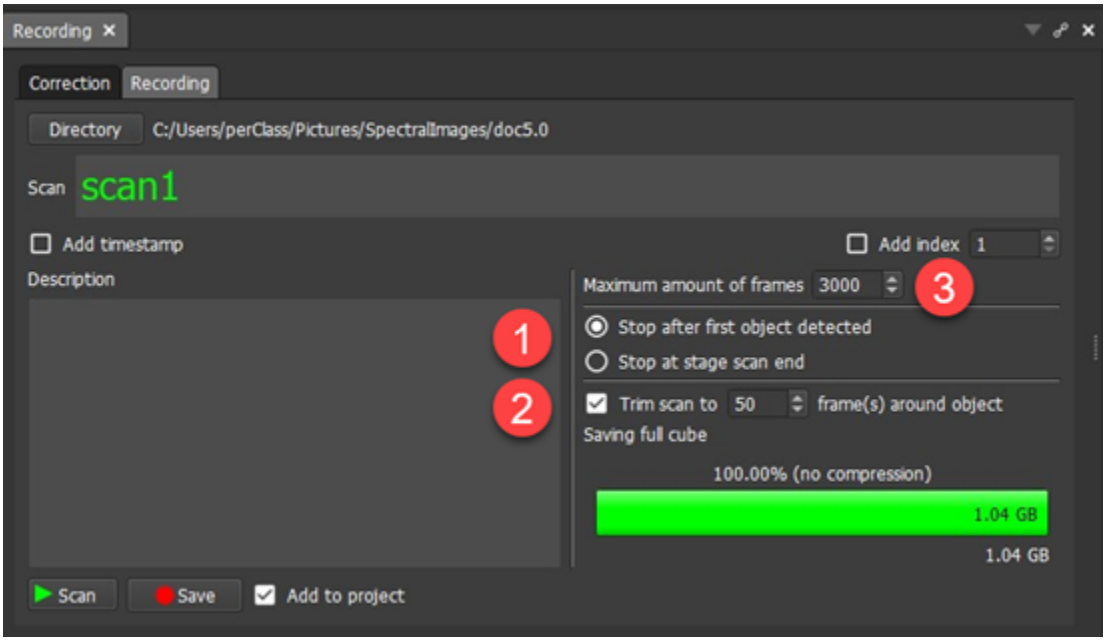
Scan mode brings significant benefits when scanning multiple objects in order to build large training data sets:

- Scans can be discarded and re-taken. This makes it easy to acquire high-quality curated data sets.
- Scans can be compressed by preserving only foreground objects of interest, not background.
- Scanning in memory enables high-speed acquisition of data on top of industrial belts. We have tested acquisition up to 2m/sec belt speeds on common hardware.
- Scans can be overwritten. When saving the scan while the same scan file exists on disk, user is prompted with a dialog requesting explicit confirmation whether to overwrite the existing scan or not.

TIP When using the directory-specific references, only the scan is overwritten, not the reference files. If camera settings such as exposure changed since the directory-specific references were recorded, this may lead to inconsistent scan. When using directory-specific references, do not change exposure when saving data into the same directory.

Stopping acquisition in scan mode

In the **scan mode**, the user has several ways how to stop the scan. These can be controlled by the settings 1 in the *Recording* panel:



The user may end the scan at the end of the scan area, defined in the *Stage* panel.

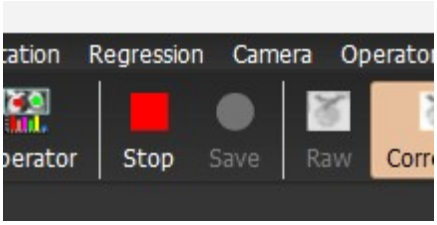
An alternative is to stop the scanning when an object is detected:

- **When working with perClass Stage**, this option significantly simplifies our scanning work-flow when acquiring training data sets. In this situation, we wish to have each object saved into a separate scan. Because the scanning is stopped by object detection, we can also place objects anywhere on the stage.
- Stopping on object detection enables also enables **comfortable data acquisition on top of industrial belt systems**.

In both of these scenarios, the extra option 2 trims the resulting scans to avoid extensive background areas.

Naturally, the scanning is also interrupted if the scan size reaches the pre-allocated number of frames 3 or the stage table reaches its maximum position.

At any moment during the scanning, user can also stop the acquisition using the *Stop* toolbar button



Scan compression

Scan compression is a major new functionality brought by perClass Mira 5.0. In projects developing robust models for classification or quality estimation, large data sets need to be collected with many objects (pieces of fruit, industrial parts or sorted product). Such data sets, collected on stage of industrial belts occupy significant amount of disk space where majory is the stage table ot belt background, not data of interest.

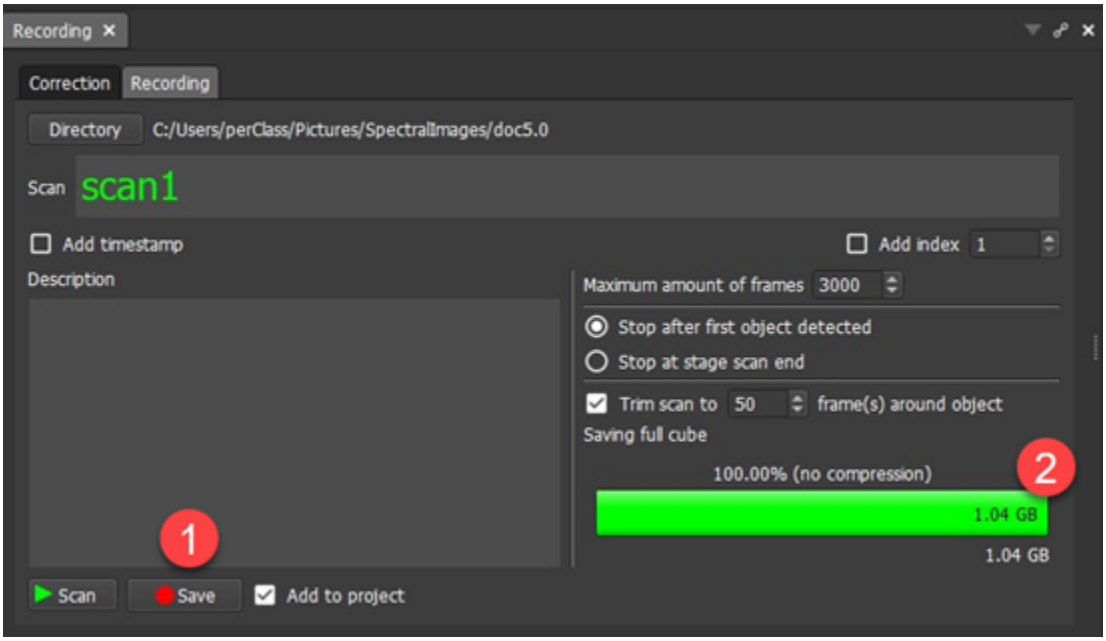
perClass Mira 5.0 make it possible to easily define the data of interest to be stored when saving scans to disk. This is enabled by the new Camera **Scan mode** acquiring data to memory before saving to disk.

Scan compression in perClass Mira is fully lossless. This means that data is not anyhow altered, reduced or changed.

Scanning with compression is also very easy to use. Before saving the data to disk, user can mark areas that will be preserved. Foreground definition can be automatically applied by a object segmentation defined in the project. User is always in full control deciding what exactly is saved and preserved. The scanning process is streamlined so that user can accquire large collections of high-quality samples in minimum amount of time while significantly reducing disk storage.

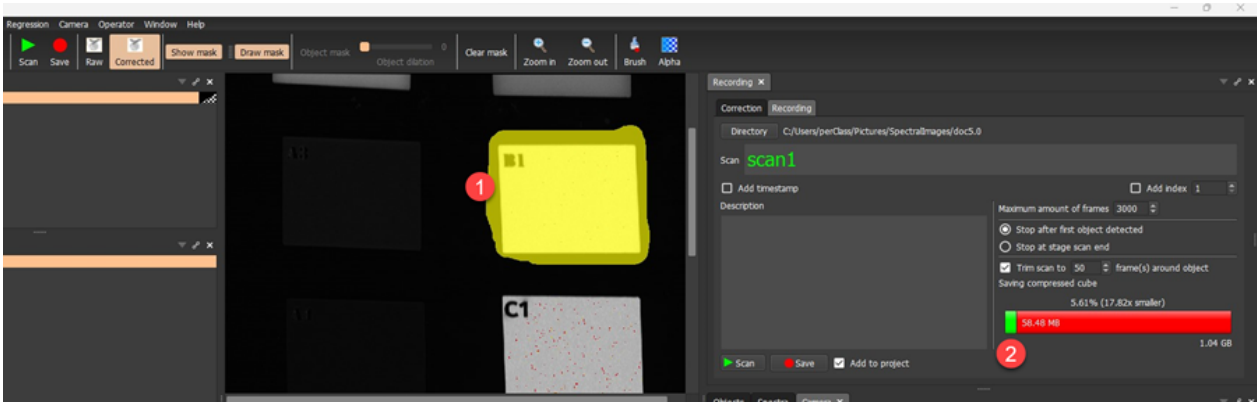
Example of using scan compression

In the camera **Scan mode**, we acquire a scan. By default, when saving the scan using the *Save* button 1 in the *Recording* panel or in the toolbar, the entire scan would be stored on disk. This is indicated by the scan size widget 2 in the right half of the *Recording* panel. Note that these controls are available only in the **scan mode**, not in the **belt** or **waterfall** mode.



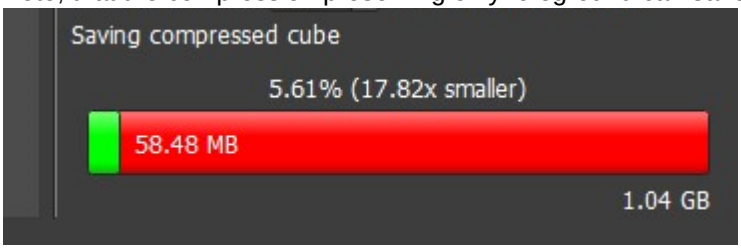
Defining foreground mask manually

User may paint in the scan and define foreground content to be saved 1. The scan size widget will show in green 2 the data size actually saved to disk as a fraction of the total scan cube size, which is displayed in red.



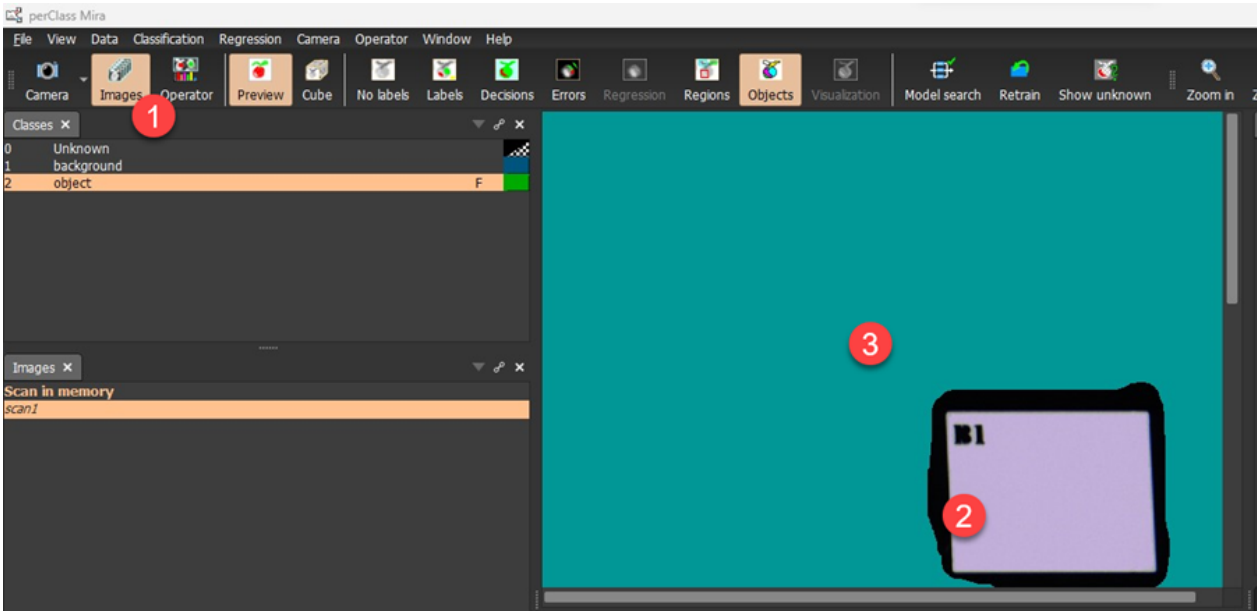
Similarly to normal label painting, the yellow foreground mask can be removed when holding the *Shift* key. Size of the brush used can be adjusted by the *Brush* toolbar button and its transparency by the *Alpha* button.

Note, that the compression preserving only fofeground can save very significant amounts of disk space.



Compressed scans in the workspace

When a compressed scan is saved, perClass Mira switches to the *Images* mode 1, as usual. The area 2, previously highlighted by the yellow label, is fully preserved. The scan geometry is also fully preserved. However, the remaining area 3 is rendered in cyan color. This part of the scan does not contain any information and cannot be used in any data analysis in perClass Mira environment.



Compressed scans on disk

When saving the data, perClass Mira stores the scan in a file with scan1.hdr header and scan1.pcz compressed file. This is naturally not a standard ENVI cube but requires perClass Mira to be read.

The following screenshot from Windows Explorer shows several scans. The scans 1 and 2 1 were compressed while the scan3 2 was not. It is, therefore, stored in the standard .pcf file which is a normal ENVI cube.

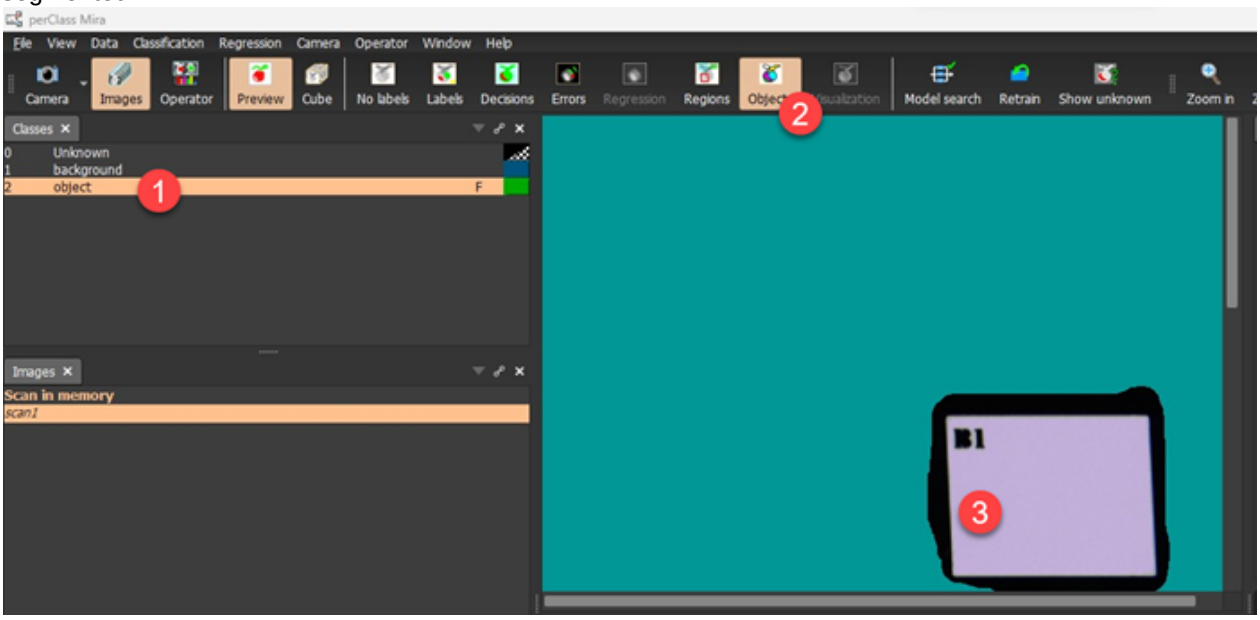
Name	Date	Type	Size	Tags
darkref.hdr	19/01/2024 14:13	HDR File	3 KB	
darkref.pcf	19/01/2024 14:13	PCF File	602 KB	
plasstic_lab.mira	19/01/2024 14:07	perClass Mira	36.353 KB	
plasstic_scan1_and2.mira	19/01/2024 14:14	perClass Mira	39.600 KB	
scan1.hdr	19/01/2024 14:30	HDR File	3 KB	
scan1.pcz	20/01/2024 12:52	PCZ File	57.106 KB	
scan2.hdr	20/01/2024 12:57	HDR File	3 KB	
scan2.pcz	20/01/2024 12:57	PCZ File	52.440 KB	
scan3.hdr	20/01/2024 12:59	HDR File	3 KB	
scan3.pcf	20/01/2024 12:59	PCF File	266.084 KB	
whiteref.hdr	19/01/2024 14:13	HDR File	3 KB	
whiteref.pcf	19/01/2024 14:13	PCF File	602 KB	

Automatically applying compression

Scan compression can be applied in an automated way to new scans. This only requires the user to train a classifier able to highlight areas of interest and flag the relevant class as foreground.

Enabling automatic foreground mask for scan compression

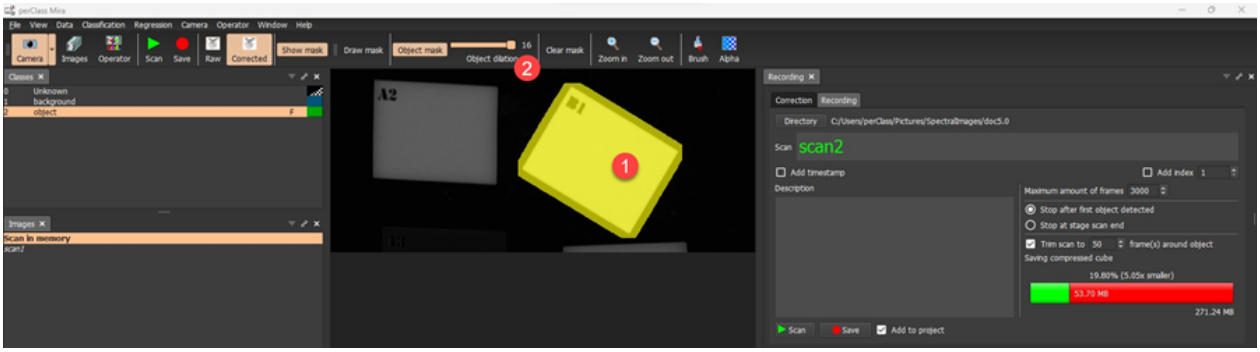
In our example, we created *background* and *object* classes (1) on the saved scan, trained a model using *Model search*. The following screenshot shows the *Object* view (2) with the object of interest (3) segmented.



When a classifier and object segmenter are available in perClass Mira project, acquiring a new scan in **scan mode** will automatically highlight the foreground in yellow

Applying foreground mask automatically

We have shifted plastic tiles on the table and acquire a new scan. In the screenshot below, we can see that the foeground area (1) is now automatically highlighted in yellow. We also can use the toolbar slider (2) to dilate the foreground mask out from the objects. In this way, we will also save some of the bacground. This is useful so that we will be able to train good models along object boundaries.



Adjusting the automatic mask

The user can anytime paint in the mask. In this way, interesting areas to be preserved even if not highlighted by the the classification model.

Other mask controls are available in the toolbar:

- *Show mask* toggles mask visualization on and off
- *Draw mask* is enabled at any moment that the user paints the mask manually. Disabling it removes only the manually painted parts, preserving the automatic model decisions
- *Object mask* re-applies the model to the scan, re-creating the mask
- *Clear mask* removes the mask

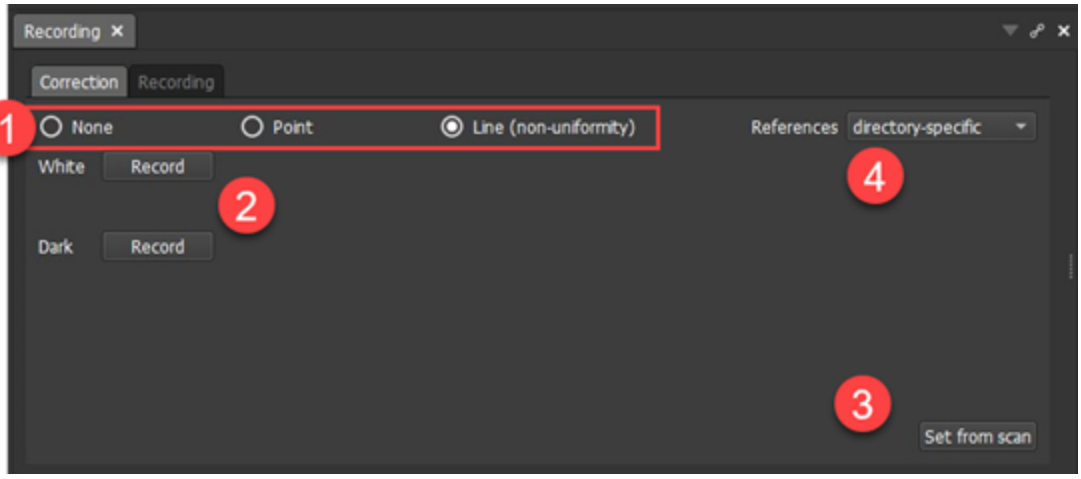
Exporting compressed scans as ENVI

Compressed scans in .pcz format cannot be loaded to other software packages as they are not ENVI cubes. It is, however, possible to batch-export any set of compressed scans as ENVI cubes that can be loaded in other data analysis tools.

Use *File* / *Export* / *Export* cubes command to export compressed cubes as ENVI files.

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 robust to illumination changes.



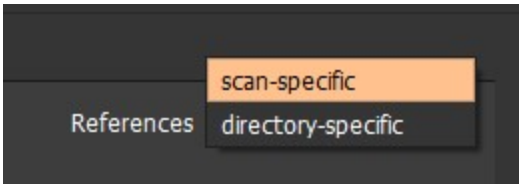
The radio button group 1 allows selection of different correction work-flows.

- The default "*line (non-uniformity)*" means that full frame dark and white references are used to correct each pixel in the image. This is a recommended approach for most applications.
- The *Point* referering means that we may define a scan area used to compute average white and dark spectrum that will be used for correcting all pixels. This is needed when white reference cannot cover the entire field of view of our sensor (for example in remote sensing or outdoor applications)
- The *None* setting disables referencing workflow. The acquired data will be used *as is*. This makes the developed models highly sensitive to illumination changes. It is generally useful only for situations where the sensor already provides relfectance-corrected data (Cubert, Imec) or where we wish to analyze raw data content. When selecting this option, recording tab is directly enabled.

The *White* and *Dark* 2 references can be acquired from the sensor by pressing the respective buttons.

The button *Set from scan* 3 allows us to set the references from the currently selected image in *Images* list. Care needs to be taken that the references of this selected image really correspond to the current illumination conditions and camera settings.

The combo-box 4 allows us to change the default location of reference files.



- **The directory-specific references** are stored only once per entire directory in the files names *whiteref* and *darkref* with the respective extensions for header (.hdr) and data (.pcf) content. This is advicable in situations where each of our data directories collects strictly only related files from the same scanning session. The resulting data storage is then cleaner and simpler.
- **The scan-specific references** means that each scan is accompanied by specific reference files appending *_whiteref* and *_darkref* to the scan filename. With this setting, we may easily mix scans from different scanning sessions or days in one directory. Disadvantage of the scan-specific referencing is larger amount of scans in our data storage.

Both approaches can be mixed in our project. This setting only affects the scans to be scanned in the future, not existing data.

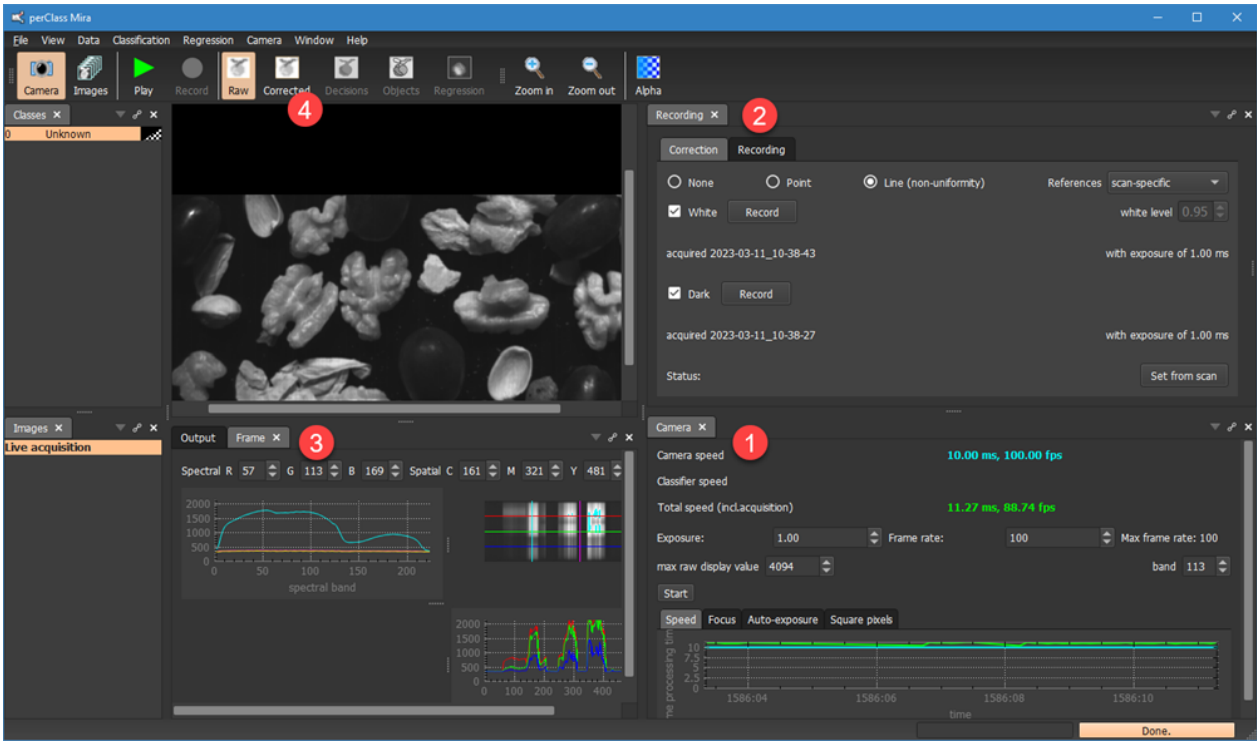
Acquired references

Once the references are acquired, their time-stamps (1) are and exposure times used (2) 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. When the exposure, set in the *Camera* panel differs from the settings stored when acquiring references, the difference is emphasized by red color in (2).

The *Recording* tab (3) 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.



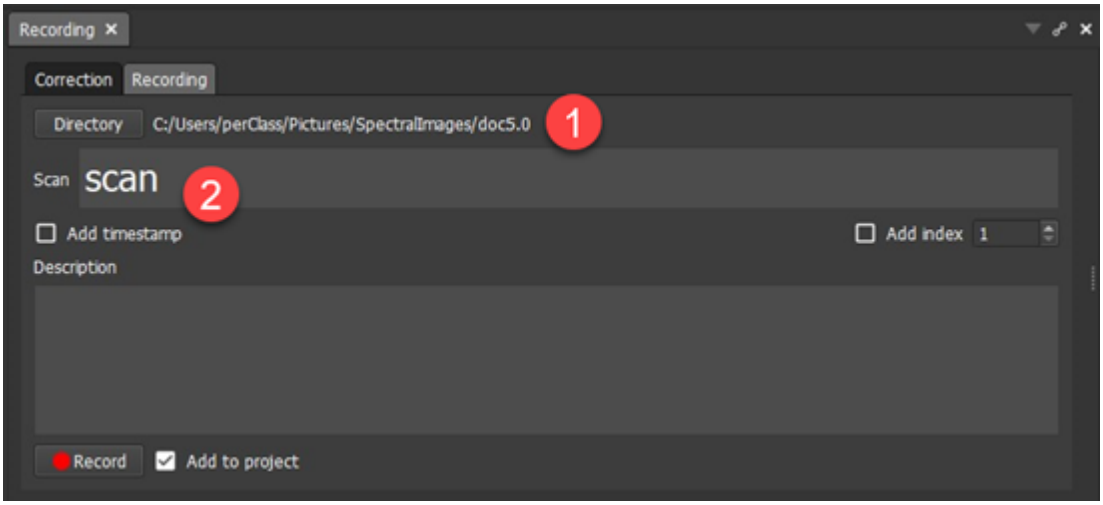
Both, the *Camera* (1) and *Recording* (2) panels are available above each other. For line-scan

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 1 specifies, where the recorded scans will be stored. This directry 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.

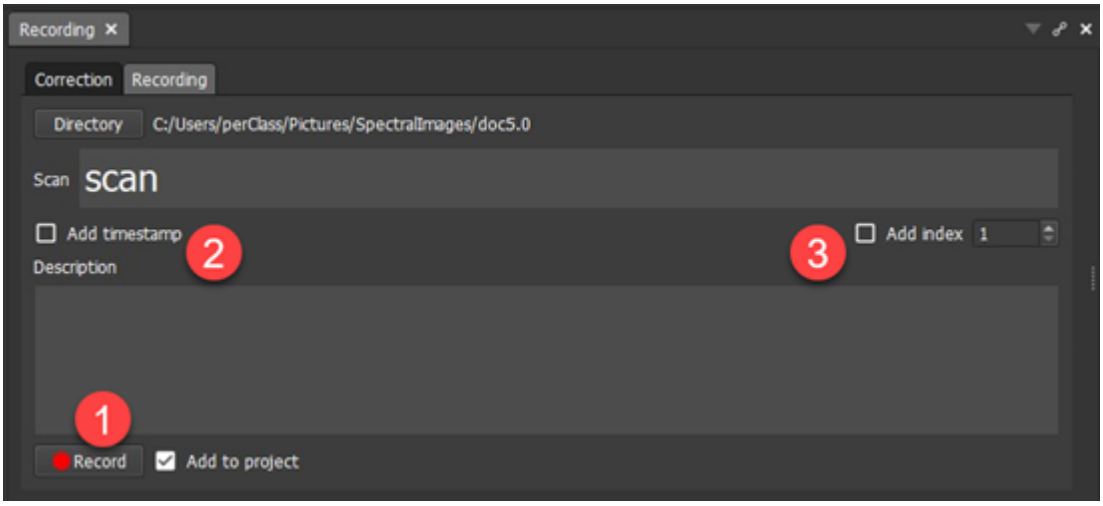
The scan field 2 defines the scan name to be recorded. Note the bigger font used for the scan name. This is because perClass Mira Stage users can fully operate the scanning process using only the stage buttons at a larger distance from the computer and the screen.

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.

Making scan names unique upon saving

The *Record* button 1 starts the data recording (in *Camera* **belt** or **waterfal** modes). In the **scan** mode, this button invokes saving of the data. In any of these operations, we may wish to make the scan name automatically unique. We have two options:

- Appending a time-stamp 2
- Appending an index (to be precise an underscore, followed by an index) 3 . The index auto-increments after each recording.



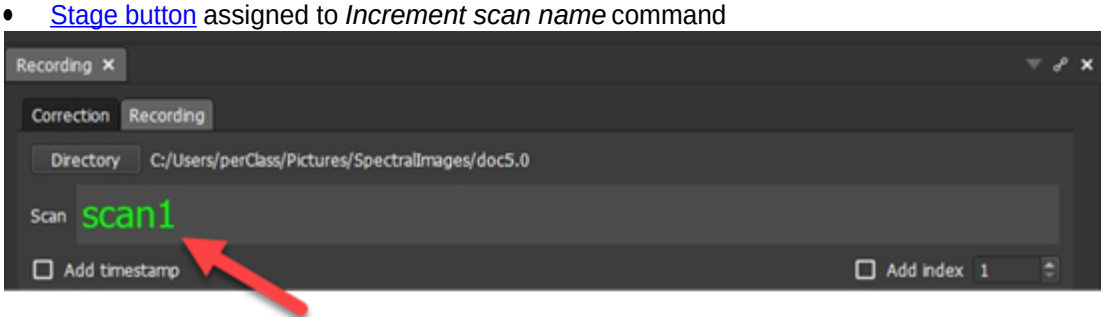
TIP: When recording scans with training examples, it is recommeded to use only examples of one class and use its name as a scan name.

Manual incrementing of scan indices

If we wish to increment indices in scan names manually, perClass Mira provide us a very simple by efficient assitance. If a scan name is a set of letters followed by a number, it is recognized as an "incrementable" pattern and it is colored in green. Compare the following image with the screenshot above where the scan name did not end with a digit.

The user may increment the index using:

- Cursor up / down keys
- Mouse wheel
- Stage button assigned to *Increment scan name* command

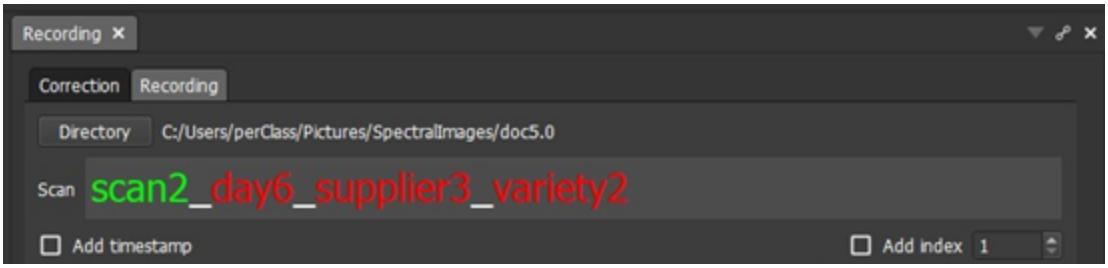


Meta-data information in scan names

In larger projects, it is recommended to store different types of meta-data information about the samples in the scan filenames. It is, for example, useful to distinguish the day of scanning, supplier, fruit variety or others. The main advantage is that we may easily select images in our project based on these textual patterns. This helps us to test or cross-validate our models on unseen days, suppliers or varieties.

The scan name widget recognizes underscores as bounday between name items. One item is always selected. It is rendered in green. Other items are colored in red. We may select the item by clicking or positioning the cursor.

Above listed methods of incrementing the scan name then apply only to the current item.

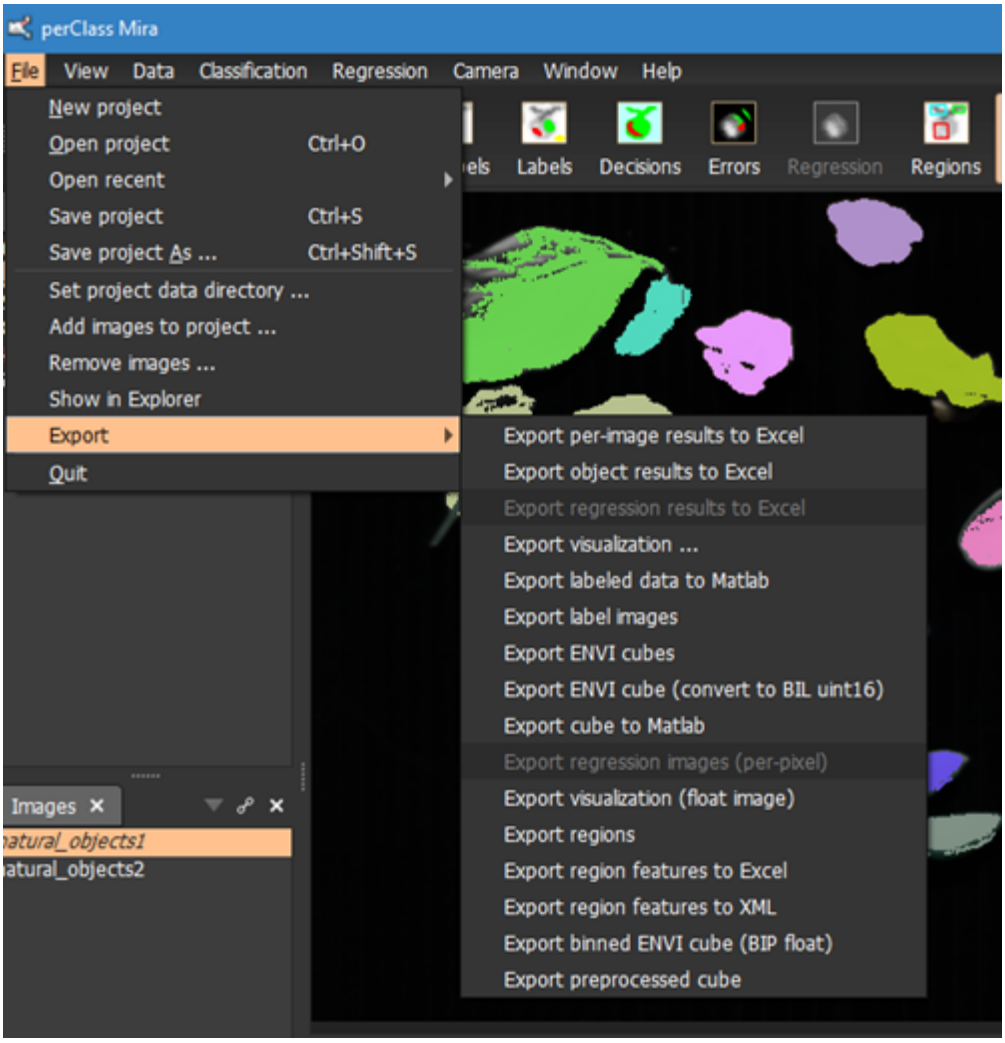


Exporting

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:

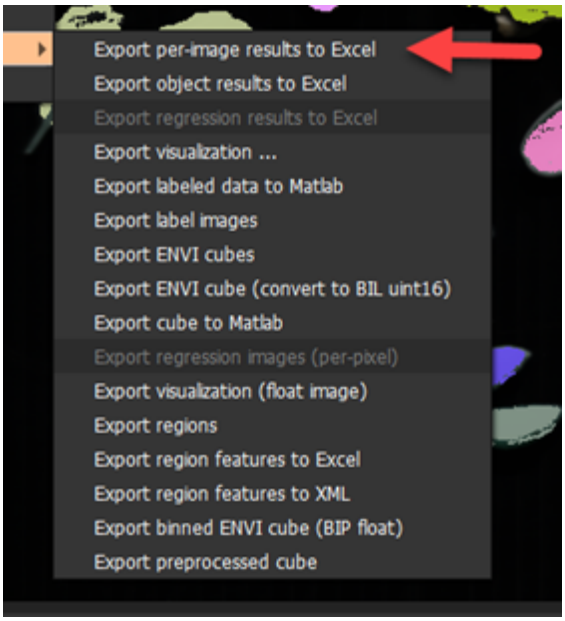
- Per-image results - for further analysis
- Per-object results - for further analysis
- Extracted features - for further analysis
- Visualizations - for display
- Visualizations as float images - for further analysis / external model training
- Cubes
- Labeled data
- Regions - connected to Region importing



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

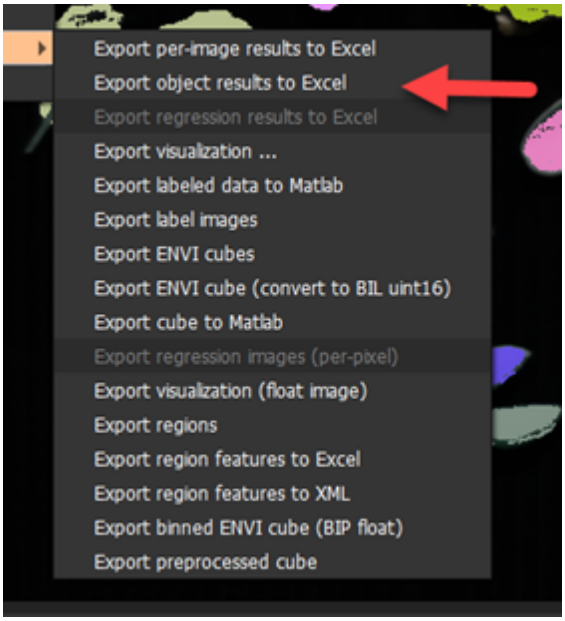


The resulting Excel file contains, for each image (1) the pixel count in each class (2). Note, that foreground/background object flags are also provided. Additionally, for all foreground classes, their fractions within foreground are present (3).

	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			backgrour	foreground		leaves	nuts	shells	wood
6			class is fore	FALSE	TRUE	TRUE	TRUE	TRUE	(2)		pixels	pixels					
7													(3)				
8																	
9																	
10	index	image name	(1)														
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.

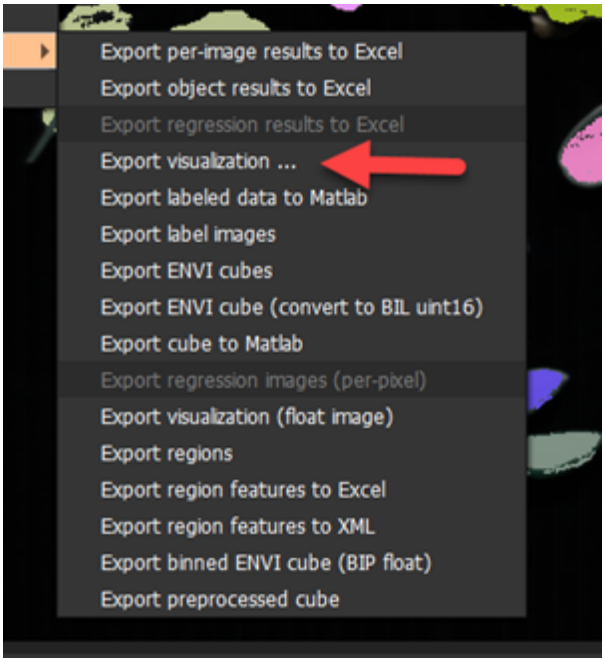


Exported results provide scan names (1), training/test status in the project (2), object size and bounding box (3), per-object decision by perClass Mira object classifier (4) and brake-down of pixel counts in all foreground classes (5). The last can be used to define and validate custom object classification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

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1														
2		version:	perClass Mira 4.2 (9-mar-2023)											
3														
4		(1)	(2)			(3)				(4)		(5)		
5				object						object	foreground classes			
6	index	image name	status	index	size	col	row	width	height	decision	1:leaves	2:nuts	3:shells	4:wood
7	1	natural_objects1	training	1	1906	313	3	59	43	4	0	0	0	1662
8	1	natural_objects1	training	2	11575	41	21	190	105	1	9233	0	1	0
9	1	natural_objects1	training	3	560	511	48	48	28	5	51	0	0	0
10	1	natural_objects1	training	4	2327	231	72	64	53	4	0	0	0	1950
11	1	natural_objects1	training	5	3566	333	73	91	67	4	1	0	0	2905
12	1	natural_objects1	training	6	608	566	106	35	28	5	19	0	0	0
13	1	natural_objects1	training	7	8060	72	118	181	97	1	6994	0	0	0
14	1	natural_objects1	training	8	2611	309	144	50	68	3	99	0	1906	0
15	1	natural_objects1	training	9	1795	238	175	42	58	2	81	1035	1	0
16	1	natural_objects1	training	10	2499	388	177	68	53	3	55	0	1614	0
17	1	natural_objects1	training	11	27559	441	200	181	201	1	26626	0	0	0
18	1	natural_objects1	training	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.



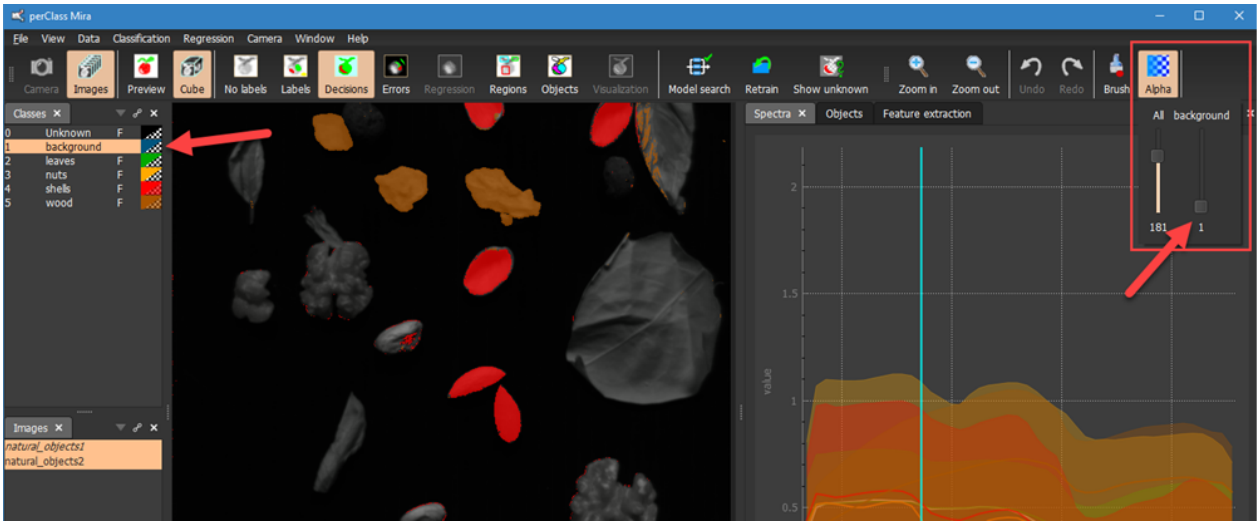
Select one of more images in *Images* list and use *File* / *Export* / *Export visualization...* command. A dialog box appears where the destination directory can be selected or a new one created. In addition, perClass Mira requests an optional suffix appended to image names. This is useful to distinguish 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

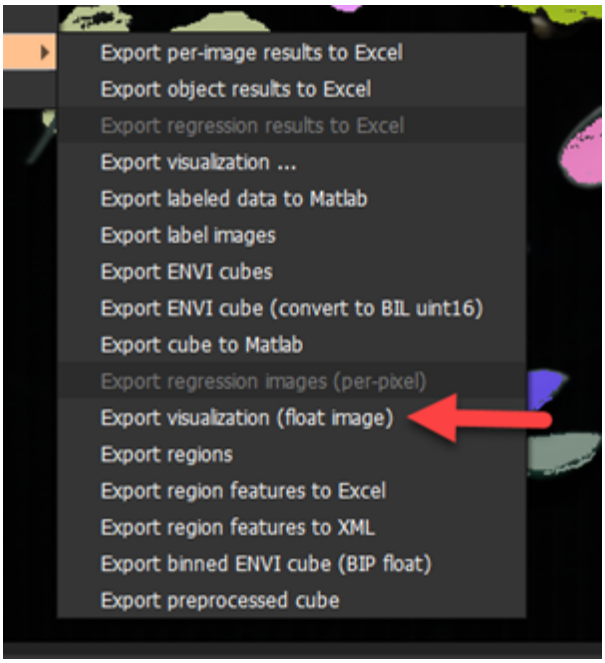
For quick visual identification of specific detections, it is convenient to control alpha layer per class and make unimportant classes fully transparent. This can be achieved by selecting specific class, using Alpha toolbar button class and adjusting the transparency only for this class. For stronger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



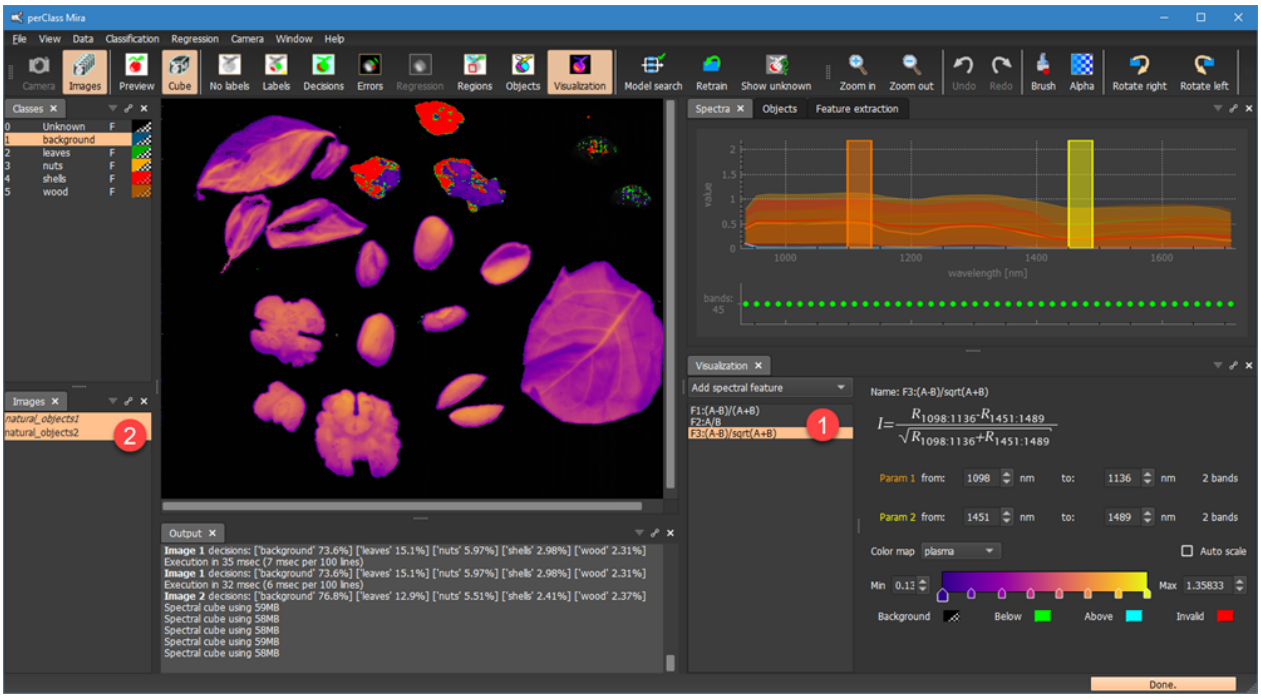
Exporting visualizations as float images

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.



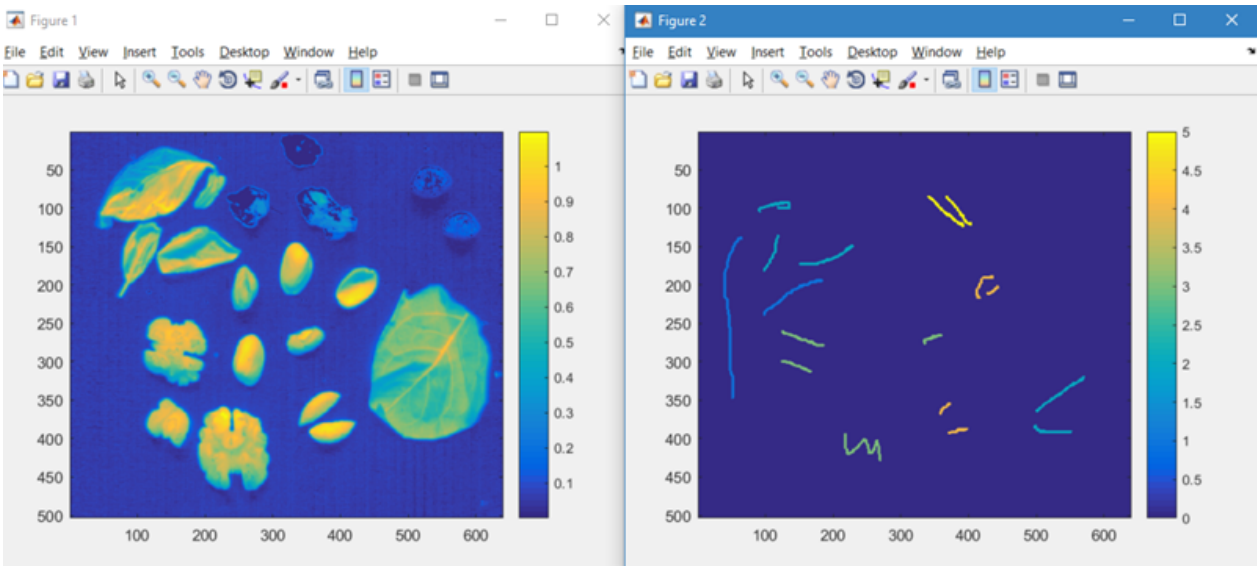
In the following example, we defined three spectral indices 1 . We select the scans we wish to process 2 and use *File* / *Export* / *Export visualization (float image)*. We may then select or create destination directory. For each selected image a .mat Matlab binary file is created with floating point 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')
```

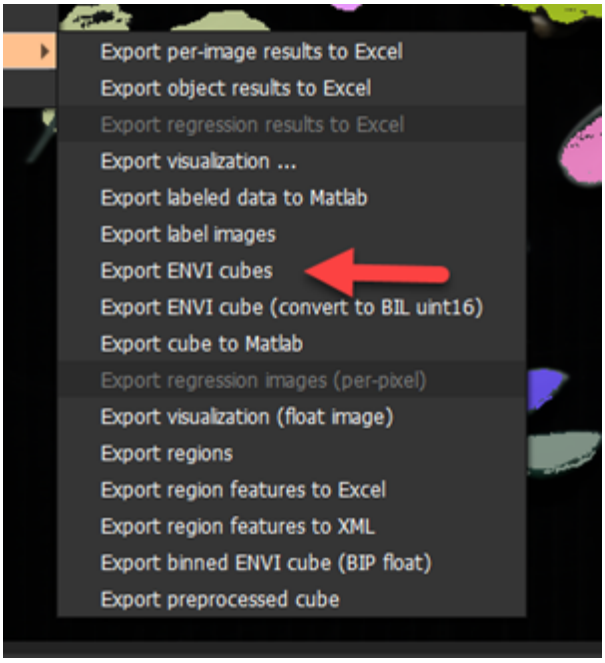


Comments:

- 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 varienty.

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:

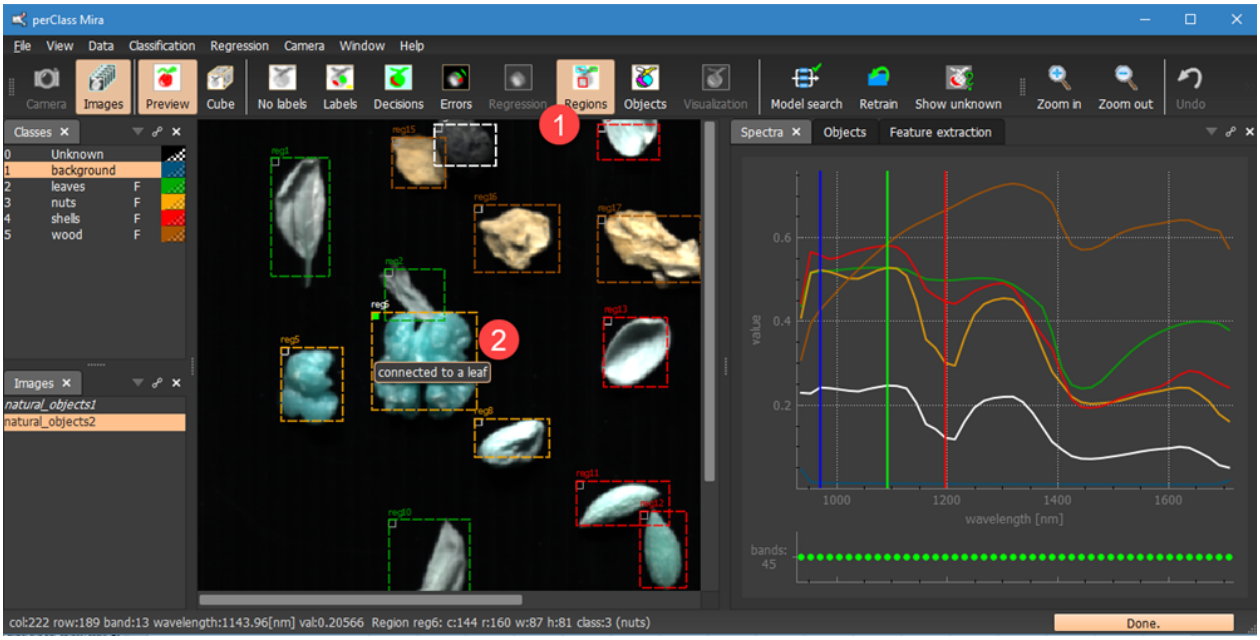
- Images are exported in exact same representaion as in perClass Mira workspace. This means that, for project types converting scans on load to reflectance, the exported images are stored as reflectance (typically BIP layout, float data type)
- For images, that are cropped in perClass Mira workspace, only the crop area is exported, not the full original image. Therefore, we may use this export type to convenniently focus only on relevant parts of the scans
- For compressed cubes, non-existent content in the background is replaced by zeros. The data is exported from the perClass Mira workspace *as is*. This means that, if the scans are already corrected into reflectance on load, the reflectance is exported (BIP layout, float data type)

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 aditional command Import of regions into perClass Mira from Excel file

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



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

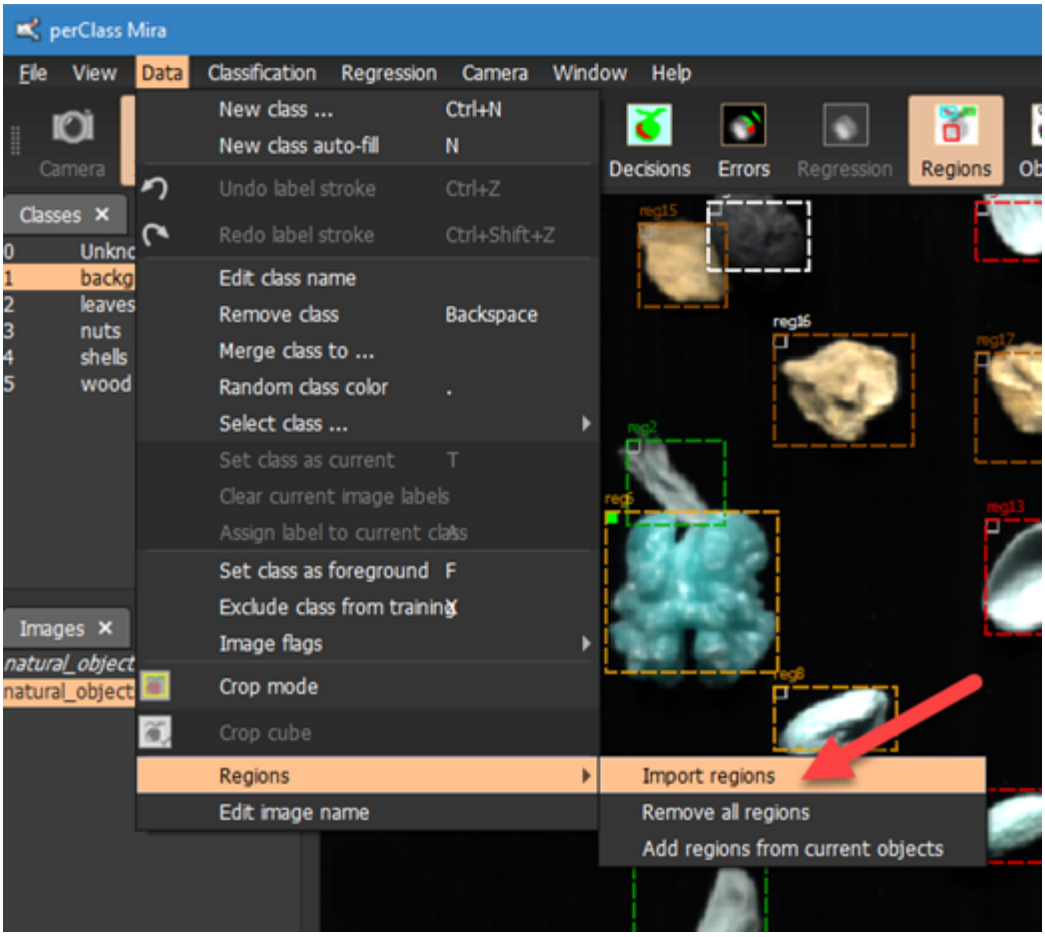
For exach exported scan (1), each region is given including its name (2), bounding box (3), class (4) and optionally also notes (5)

	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:

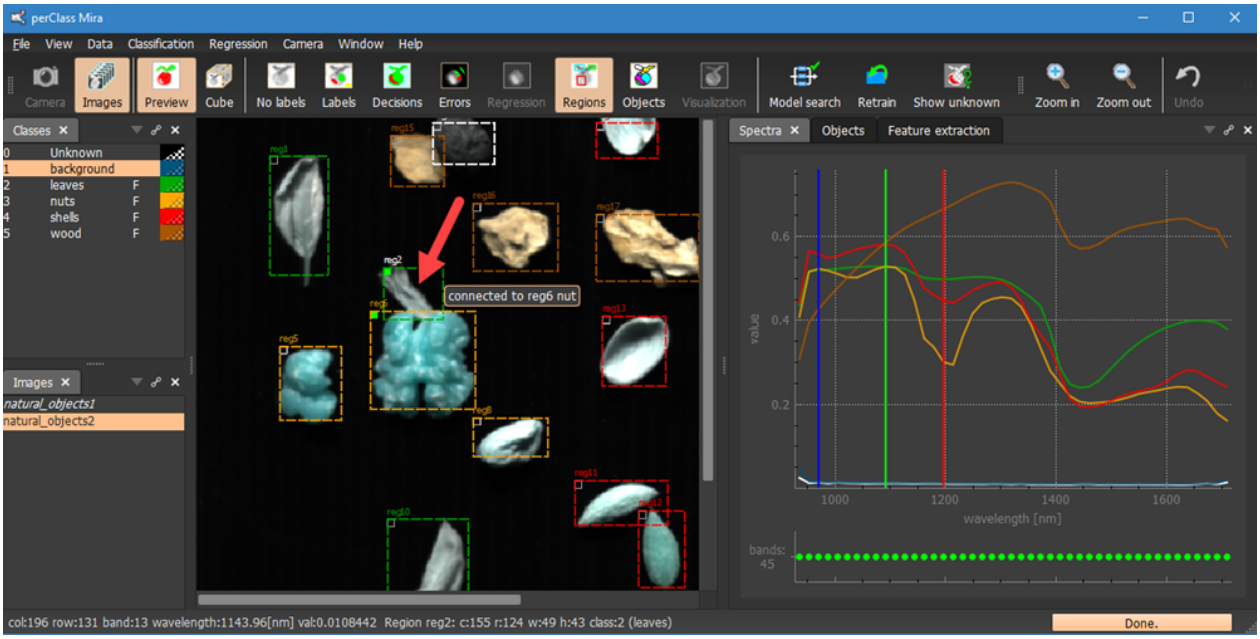


Extending our export regions example, we now include an extra note to reg2 region in Excel and save the file.

	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	shells			

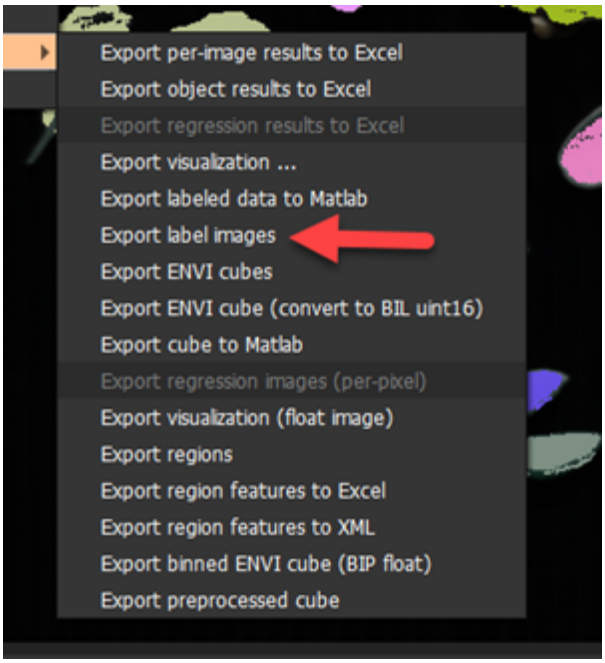
- In order to import regions, we need to first remove existing once using *Data / Regions / Remove all regions* command (when specific scan or scans to be affected are selected in the *Images* list). Note, that this follows general perClass Mira design principle of not destructing any information behind 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 assignmnets of existing ones.

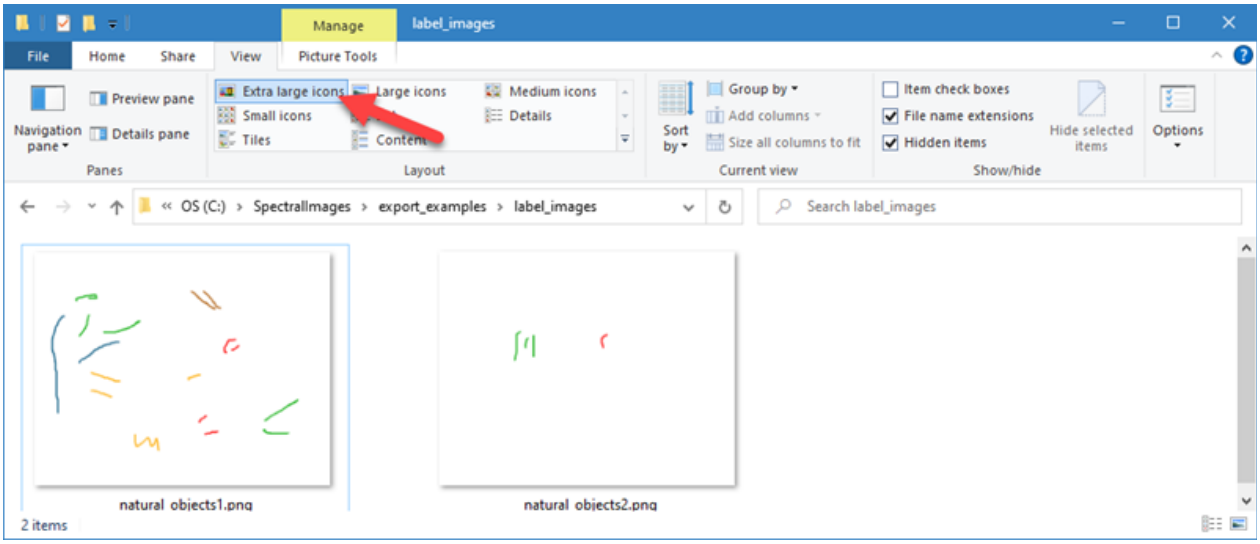


Exporting label images

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 of 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

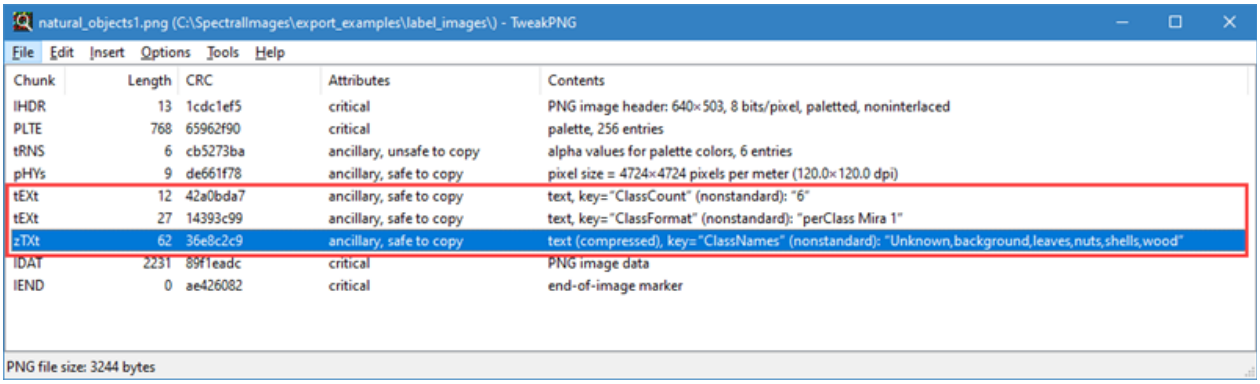


Class name meta-data

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 imginfo 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 =

  3×2 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 refering 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 retrainning will not use these images for any of the steps.

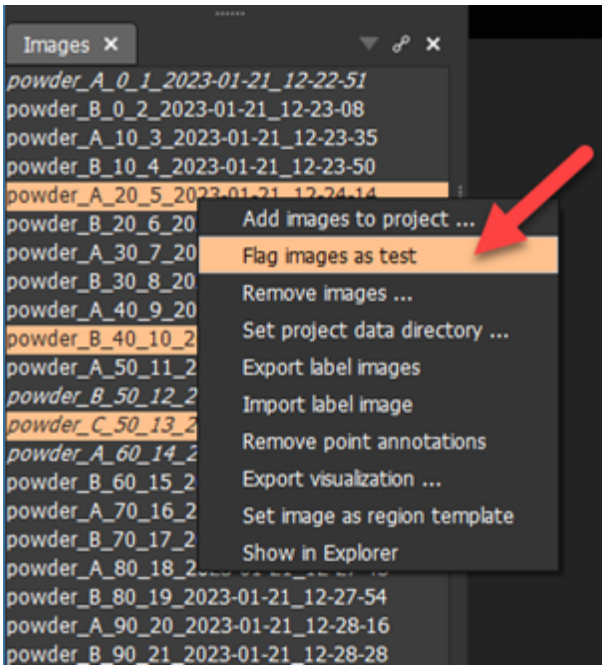
The software also provides extensive support for cross-validation used in model comparison. Cross-validation splits the data set into trainng and test parts multiple times. Each time, a model is built and the performance estimated. Eventually, we end up with a mean and standard deviation of model performance. This simplifies comparison of different models based on statistical significance.

Cross-validation is supported both over images and also over groups defined by file names (for example over days of scanning, varieties, scanning with replicas and others)

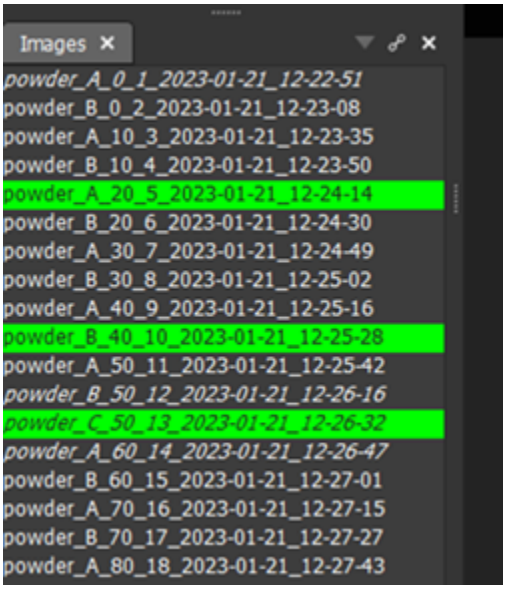
NOTE that flagging an image only does not change already existing models. The user needs to explicitly retrain a model or perform new model search in order for the new image falgs to take effect.

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:

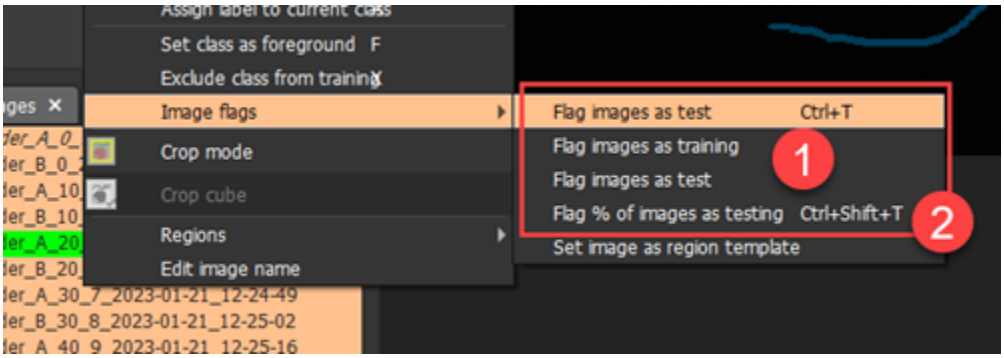


TIP: Ctrl+T keyboard shortcut performs the same action.

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 loose 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 meaningfuly 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.

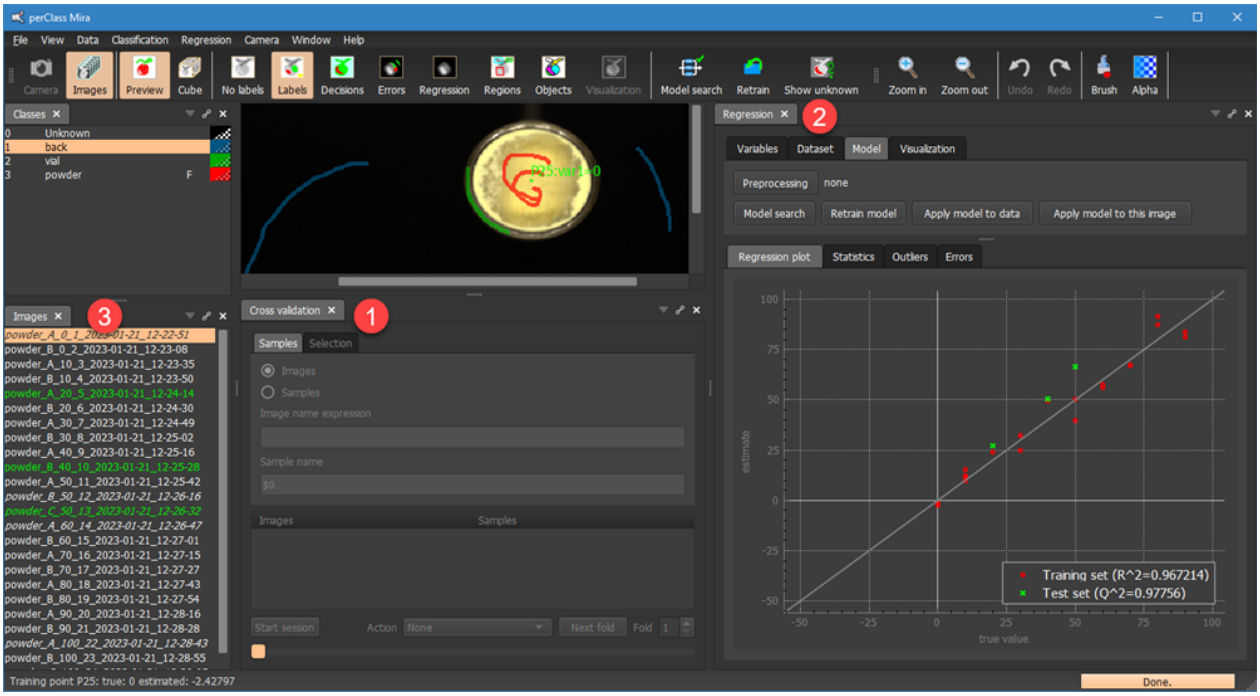
What is cross-validation?

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

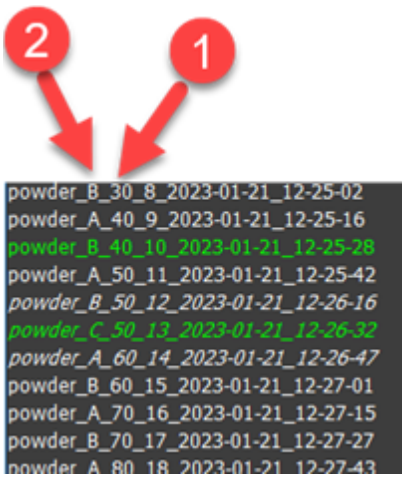
allows us to conclude whether one model performs significantly better than other.

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*.



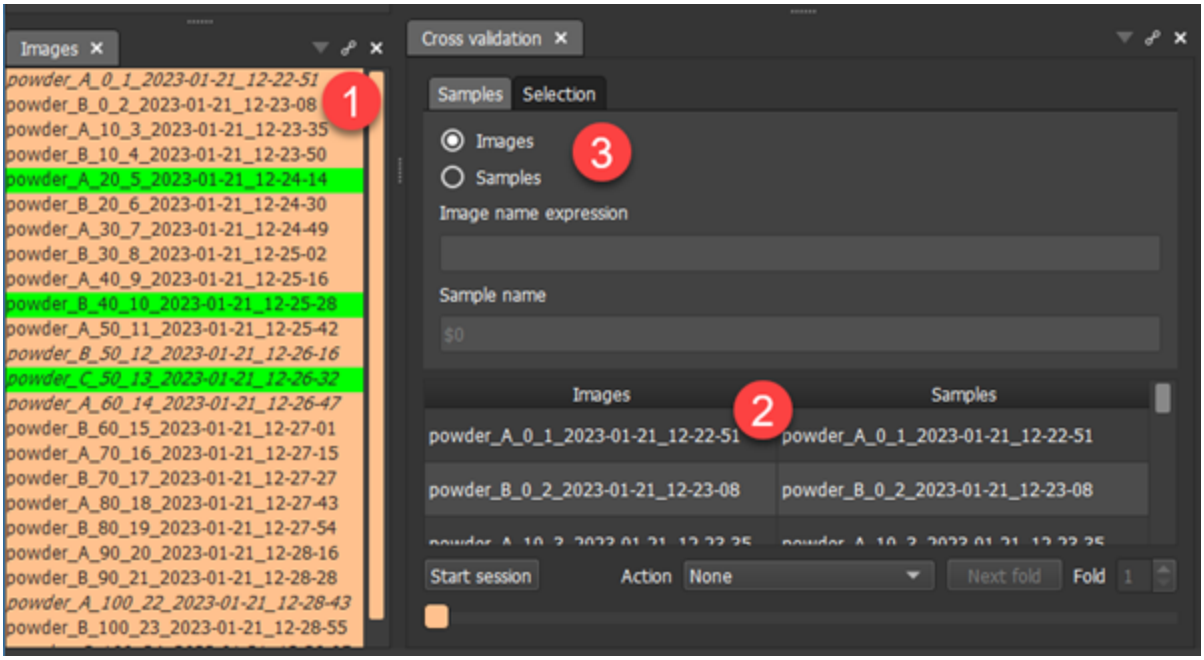
For this example, it is useful to clarify the scanning task and the structure of file names. We have a set of plastic vials with mixed powder. Each vial containing one specific mixing proportion. We scan each vial alone multiple times. Each time, we record which mixing proportion the vial contains by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



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.



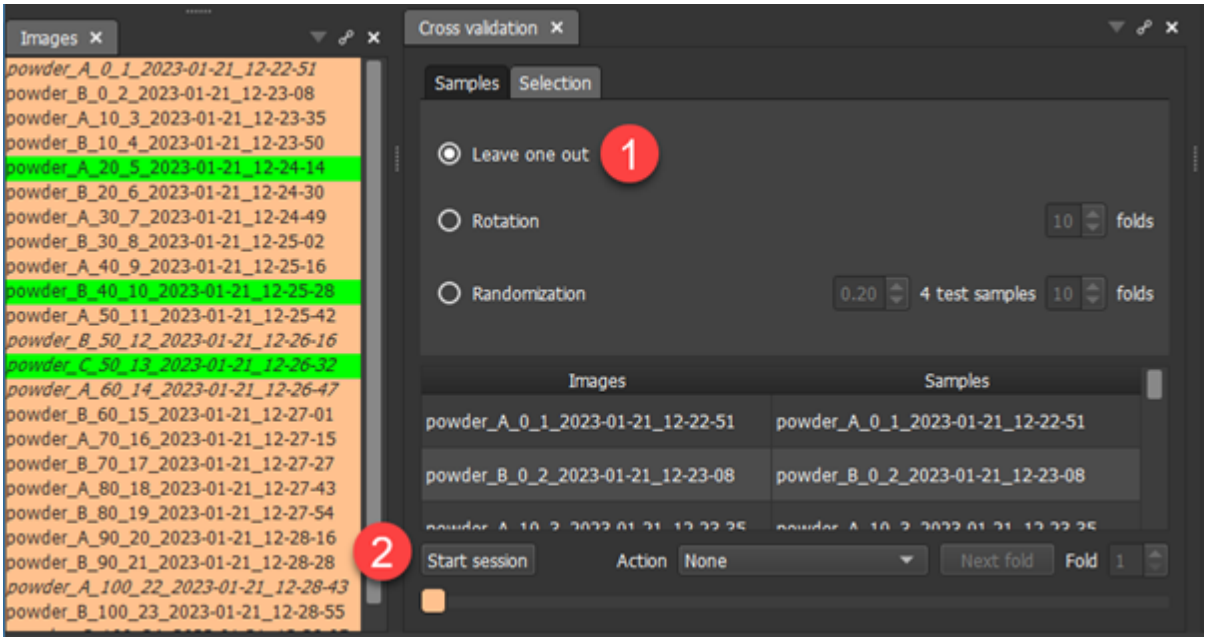
We will first explain default cross-validation over images.

Cross-validation over images

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

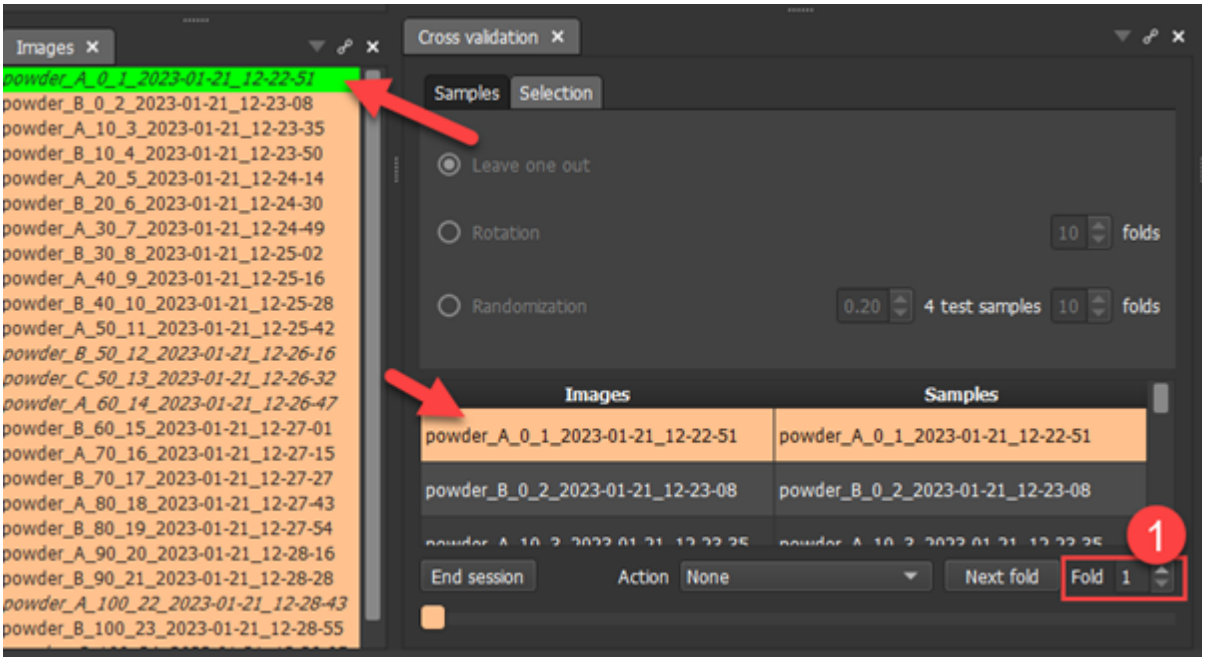
In the *Selection* tab, we have three choices for common cross-validation strategies:

- **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 trainig.
- **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.

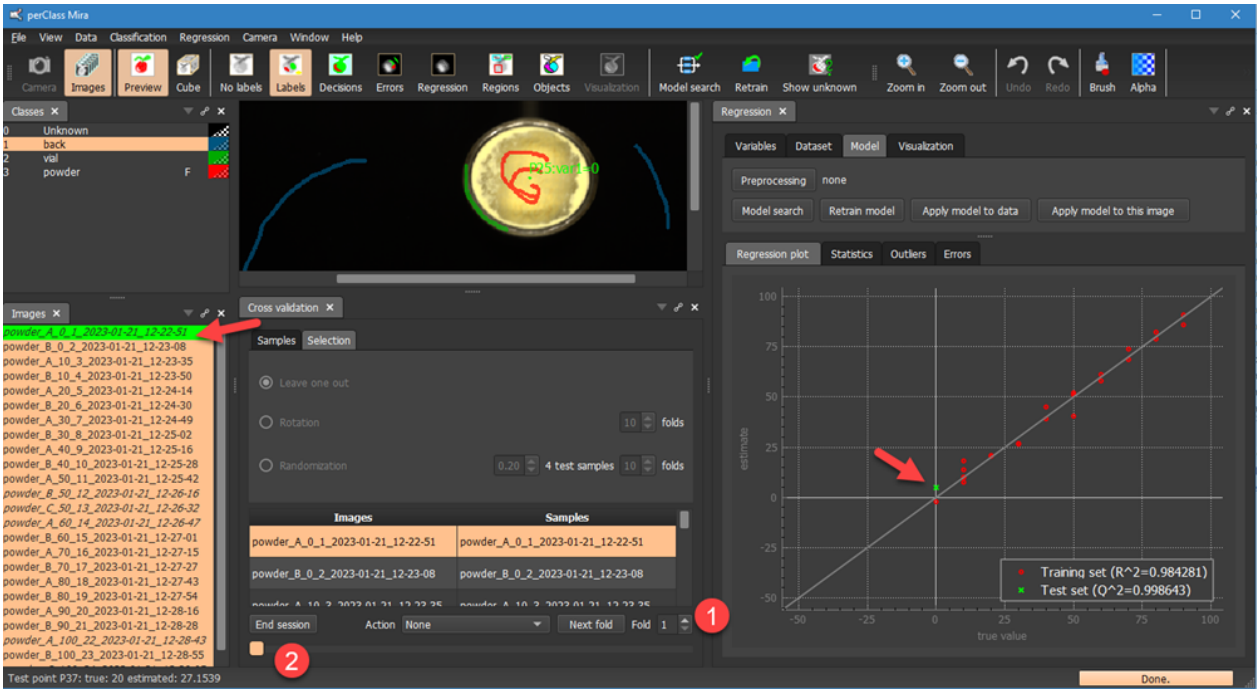


To start the cross-validation session, we click on *Start session* button 2 .

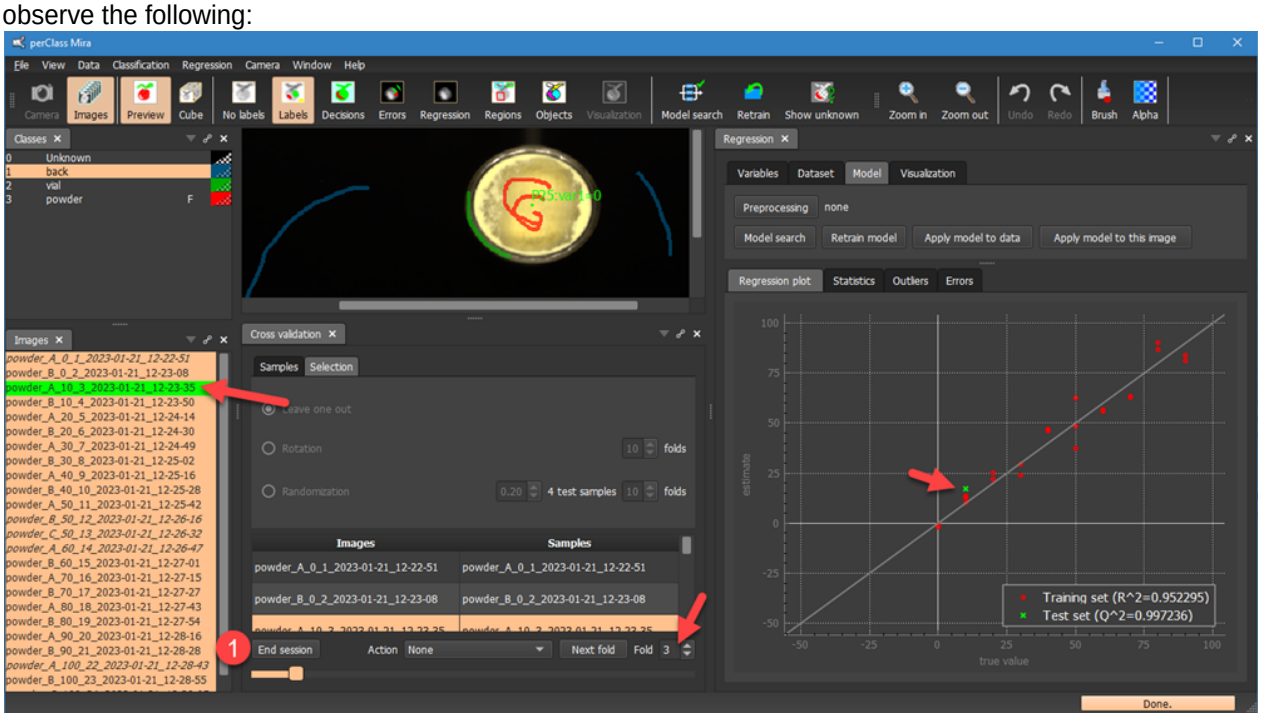
The first fold of a leave-one-image-out scheme will look like as follows:



We may now perform what ever 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 1 or the slider 2 . 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 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. Whe the Cross-validation tool really shines is the cross-valiadtion 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 1 above. Alternatively, cancelling image selection also disables the session. When selecting multiple images again, we must

explicitly start the cross-validation session.

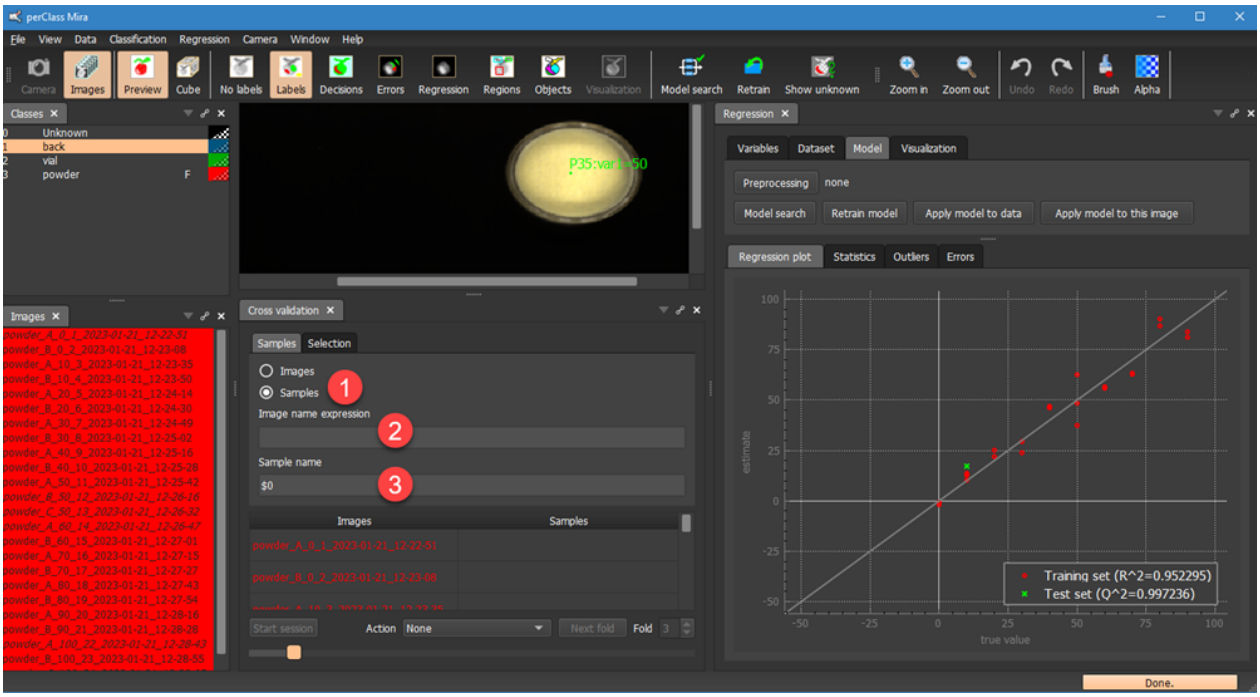
Cross-validation over replicas

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 perfrormance 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

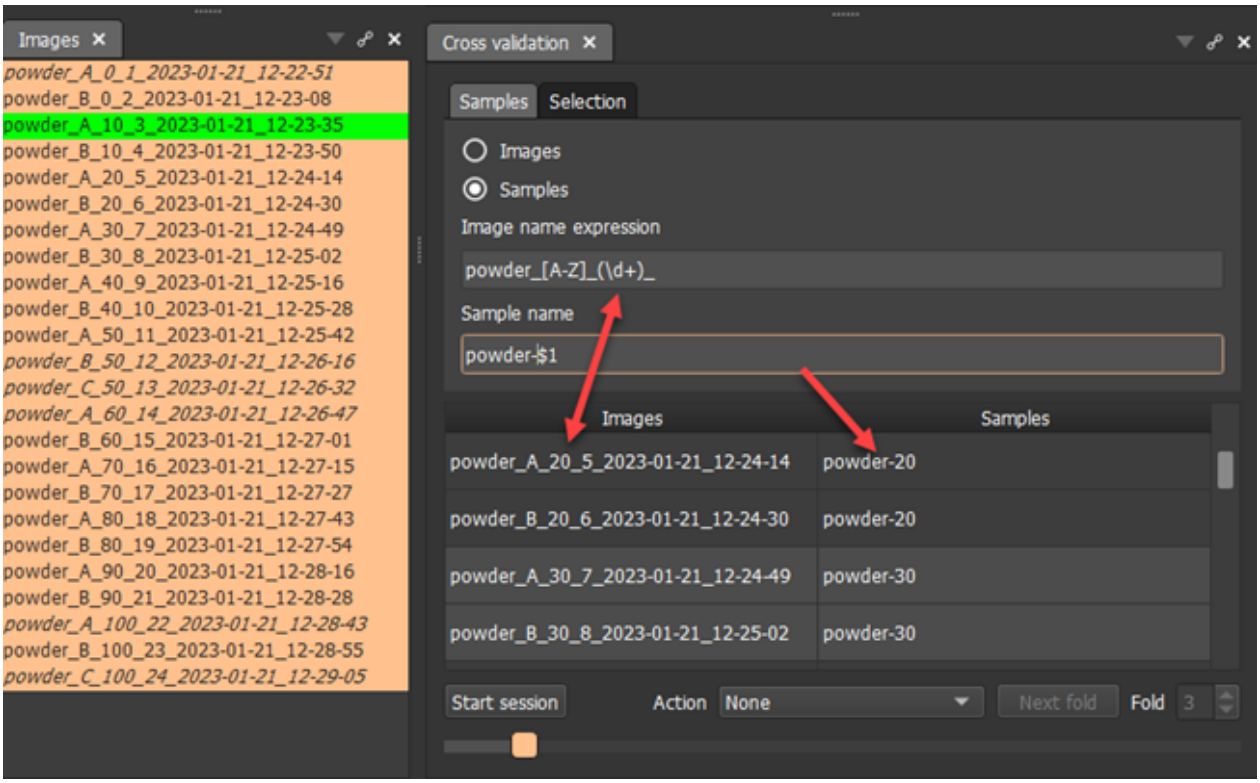
If the replica status is preserved in the scan filenames, we can easily instruct the *Cross-validation* tool to perform data splitting over replicas not over images. For example, in our poweder data set, the A/B/C... letter indicates a replica of the same physical vial (container).
We keep the leave-one-out method selected in the *Selection* tab, return to *Samples* tab and change the selection from *Image* to *Samples* (1). This means, that we may define what consititues 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 (2) that parses image names and the sample definition in (3) that ise used for the cross-valiation.



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 (2) 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 avilable to us for reference) is enclosed in round brackets. The \d referes to a digit, the + sign after means that the digit repeats one or more times. This is a standard regular expression syntaxt that is very handy when dealing with structured patterns in strings.

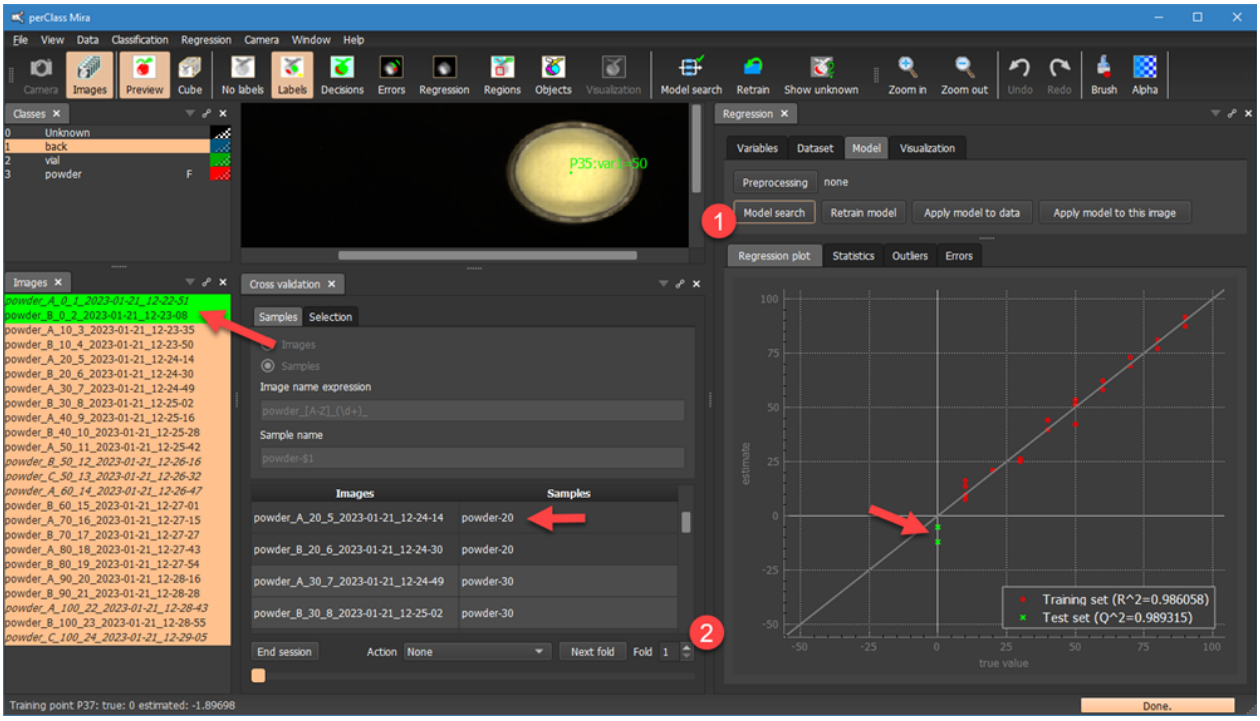
TIP: For a reference information on regular expressions, see https://en.wikipedia.org/wiki/Regular_expression



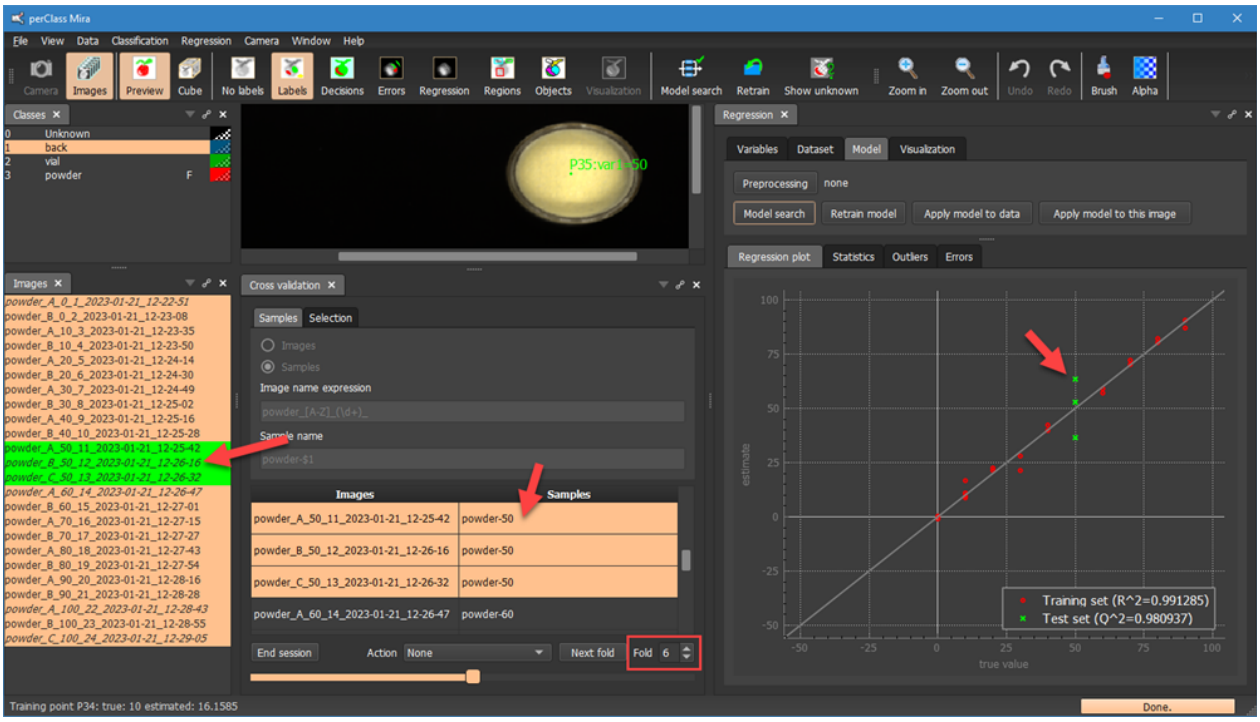
We also fill the output pattern in the *Sample name* field 3 above. The important point is that we may refer here to any captures using $1, $2 etc. syntax denoting the 1st, 2nd or later captue (text matched within the round parentheses of the regular expression).

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 2 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.

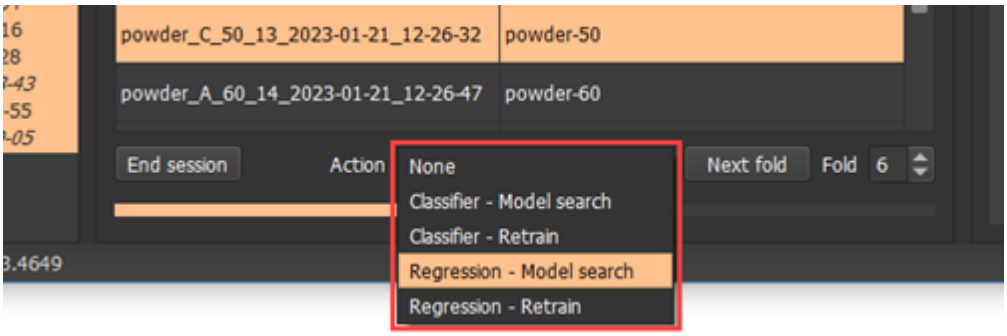
For each fold model, we may copy the regression performance from *Statistics* tab to clipboard, paste into Excel sheet. In this way, we gradually build a table of per-fold results where we will be able to assess statistical variability of each measure.

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

Release notes

5.0 24-jan-2024

- Acquisition improvements
  - Scanning to memory
    - User can decide after the scan is made whether to save it or re-take it
    - **Lossless scan compression** based on a classifier or user-defined mask (.pcz files with .hdr for meta-data)
    - Stopping scan after a number of frames is reached or an object is detected
  - Scan naming improvements
    - User can easily increment counters in a scan filename. Words ending with a number and separated by underscore can be incremented by cursor keys, mouse wheel or perClass Mira Stage buttons

    - Fast scanning work-flows
        - User can fully operate scanning session by perClass Mira Stage buttons (for example: A = scan, B=increment filename counter, C=save)
        - Stage movements and stage button commands also mapped to keyboard shortcuts.
    - Dual sensor scanning allowing the user to perform one scanning session resulting in two data sets (VNIR and NIR) on the same objects
        - Master/worker paradigm: Master instance of perClass Mira can open new "worker" instance
        - Master controls perClass Mira Stage and one sensor; Worker controls the second camera
        - User can set each instance specifically for the camera needs and choose different data storage directory
        - User then interacts with Master instance (to initiate scanning from both instances)
        - Master and worker can run on the same or different computers
    - User can define spectral and spatial ROI for camera acquisition in the frame panel. This needs to be done before adding the first scan to a project which fixes the spectral wavelength definition
- Operator mode
    - Simplyfied interface to deploy live demonstrators for solution operators
    - Support for classification and regression projects
    - Can enable user-permission system separating developers (who can change models) and operators (who cannot)
    - Operator session supports referencing
    - Can enable logging detections and results into a data base (locally to SQLite or remotely to MariaDB)
    - Project/customer logo can be changed via mira.ini file
- Image selection improvements
    - User can select images flagged as training/test or via a regular expression on image names (*Data* / *Image flags* / *Set flags by pattern* or via / keystroke)
- Regression improvements
    - Support for object classification by regression output (new rules in Object panel)
    - Support for defining region ground-truth from object point annotations (region labels by applying decision rules)
    - Full support for object confusion matrices based on regression and for cross-validation
- New acquisition plugins
    - (experimental) PhotonFocus SDK supported for
        - Inno-spec RedEye
        - Headwall MV.C NIR
    - (experimental) Resonon
        - New Applied Vision (VimbaX SDK) plugin supporting Resonon NIR systems
        - New Bassler (Pylon SDK) plugin supporting Resonon VNIR systems
        - These new plugins will replace the deprecated Resonon SDK acquisition

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 improvememnts
  - 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 enables full support Windows 11 (eBUS 5.1 and 6.1 are also supported)
  - Resonon acquisition adds a setting for a number of aquired 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 favorities)
- 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 uing 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 systens and NIR Pika systems
        - Specim - FX series (via SpecSensor SDK)
        - Unispectral - Monarch
    - redisigned 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 camers and user-defined reflectance correction work-flows
        - point correction (based on user-localized white reference in the scene)
        - non-uniformity correction to account for inhomgeneous 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 se 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 machiens
- 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, cirtularity)
    - 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 disabla 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 trully 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 2md 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 sampes 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-ASCI 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 decreases/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 Specsensor 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 diffrerent 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

Integration

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

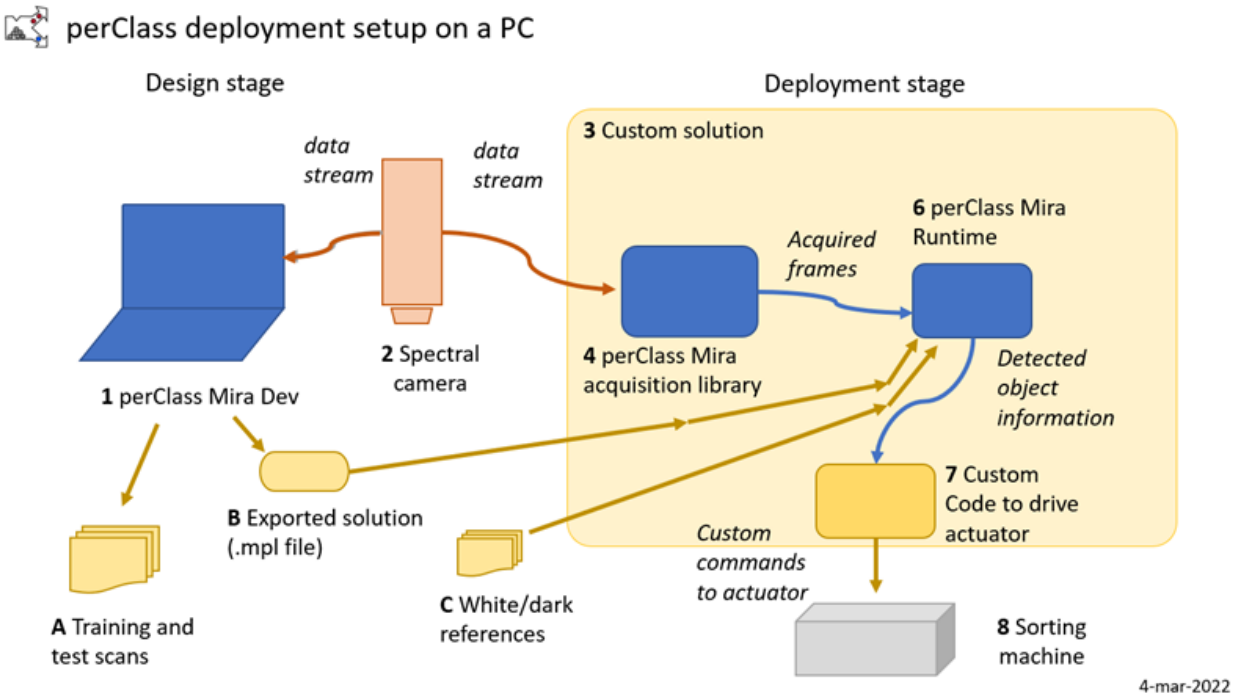
The scheme below describes the modules involved. While perClass software components are rendered in blue, the spectral sensor in red and customer-specific parts in yellow.

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 typicallt contains a classifier able to identify objects. Objects are detected by the Runtime (6) and their details such as size, centroind 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.



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 acuisition 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 Plu
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=%
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.
```

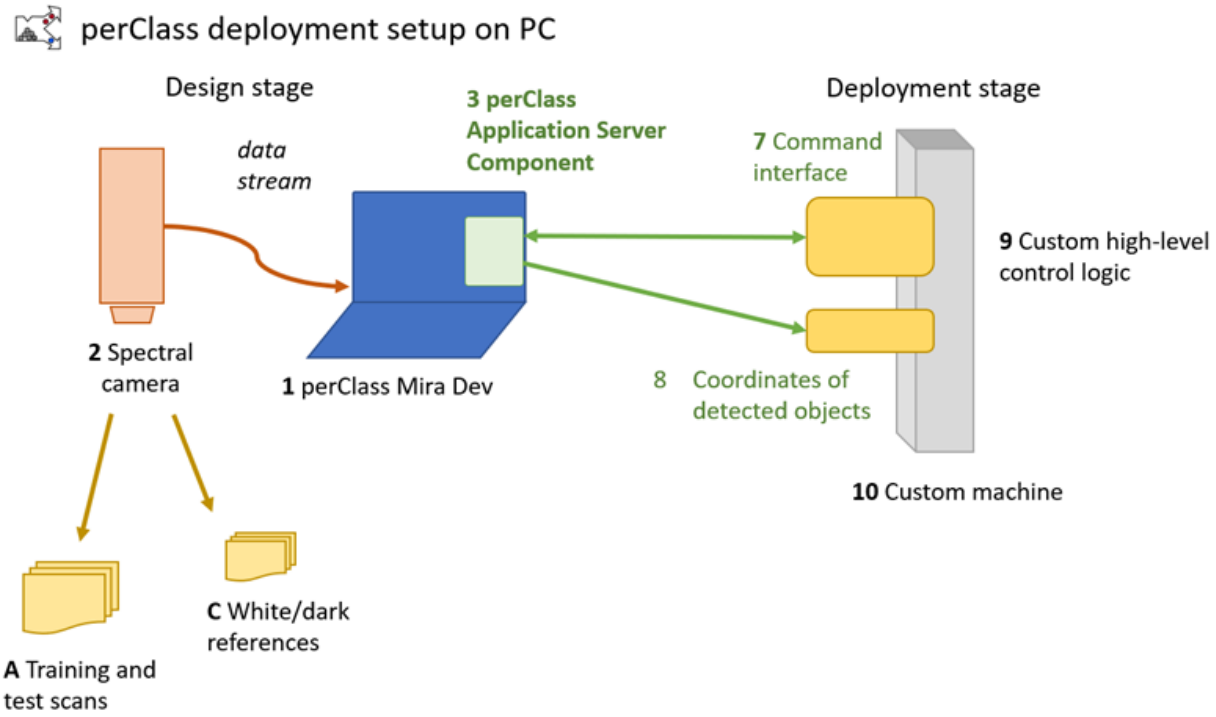
```
Error:
    if( res!=MIRA_OK ) {
        printf("Error %d: %s",miraacq_GetErrorCode(pma),miraacq_GetErrorM
    }

    miraacq_Release(pma);

    if( pBuf!=NULL) free(pBuf);

    return 0;
}

```

Application Server

perClass Mira provides an Application Server functionlity 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) enviroment. 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.



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:

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/

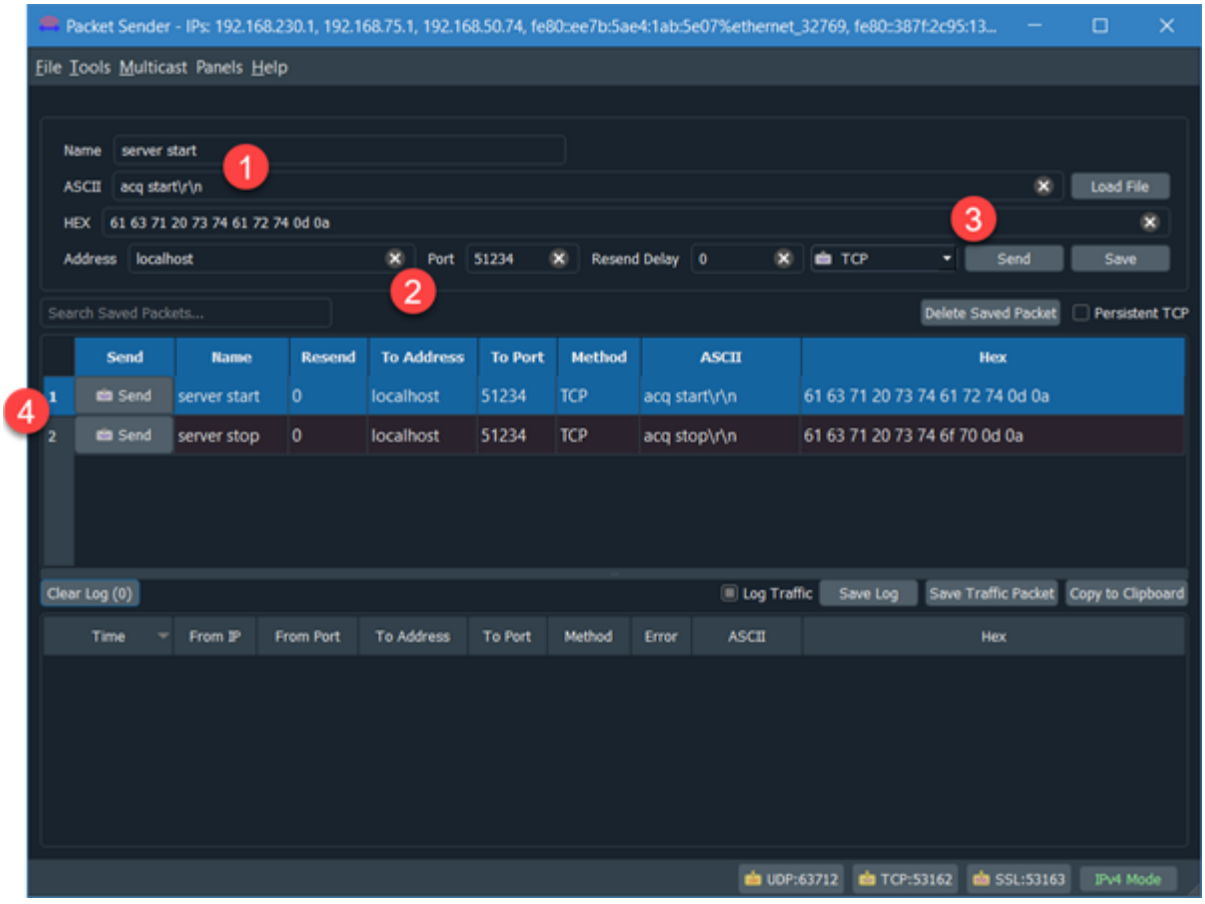
To prepare perClass Mira for running the Application Server session, we need to initialize an acquisition device (either a camera or a filereader).

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.

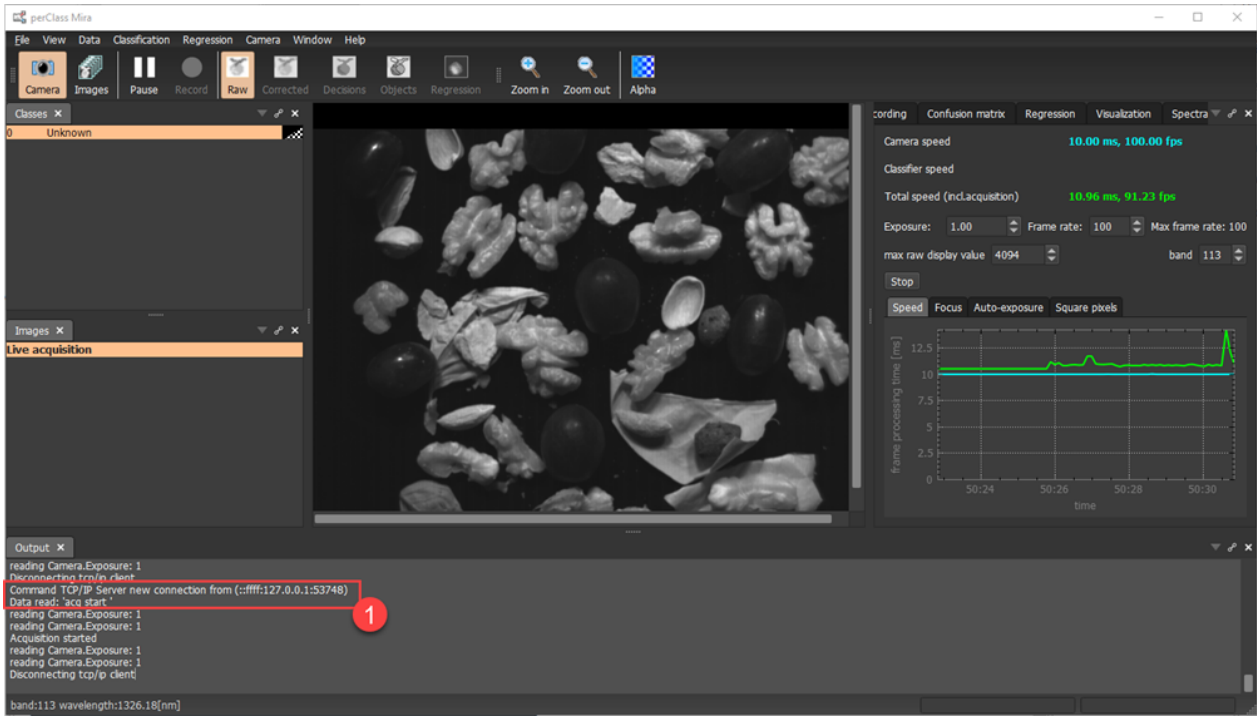
TIP: perClass Mira installation contains a ready-to-use data base of commands in PacketSender format in the lib directory. You may import these to a PacketSender session.

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 1. Secontion 2 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.

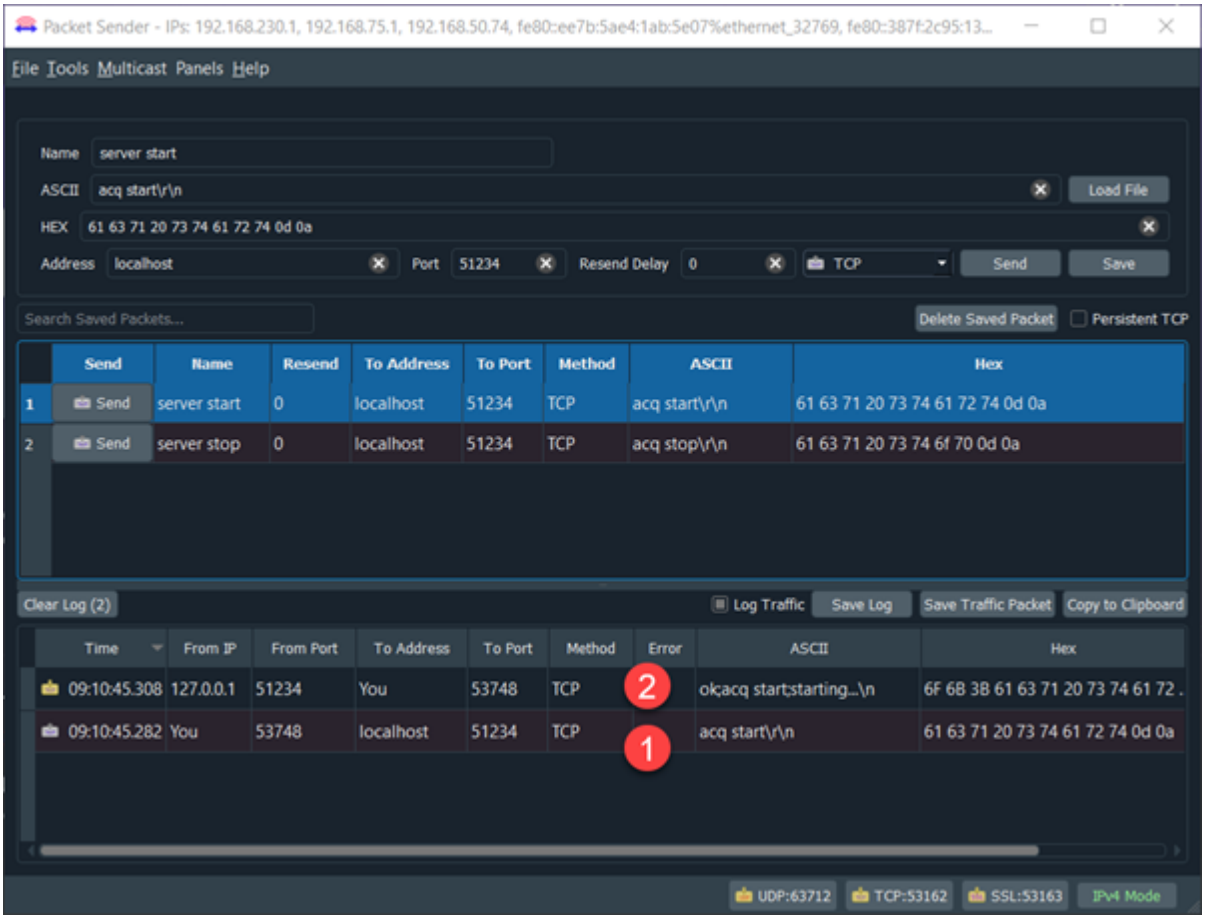
We can send the command by pressing *Send* button 3. The command can be saved for later use with *Save* button 4.



In perClass Mira window, we will see the text command listed in the *Output* window 1 and the acquisition will start:



Below you can see, that the PacketSender window lists both the command sent 1 and also the response of the Application Server 2 . The response always starts with "ok;" or "error;" string denoting whether the command is understood, followed with the actual command string, another semicolor delimiter and an aditional 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 Specsensor 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:

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

Extended commands

- **specsensor Shutter.IsOpen** - return if the shutter is open
- **specsensor Shutter.IsConnected** - return if the shutter is connected.
- **specsensor 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 here for its simplicity to demonstrate control principles as it can getestablish communication using only 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

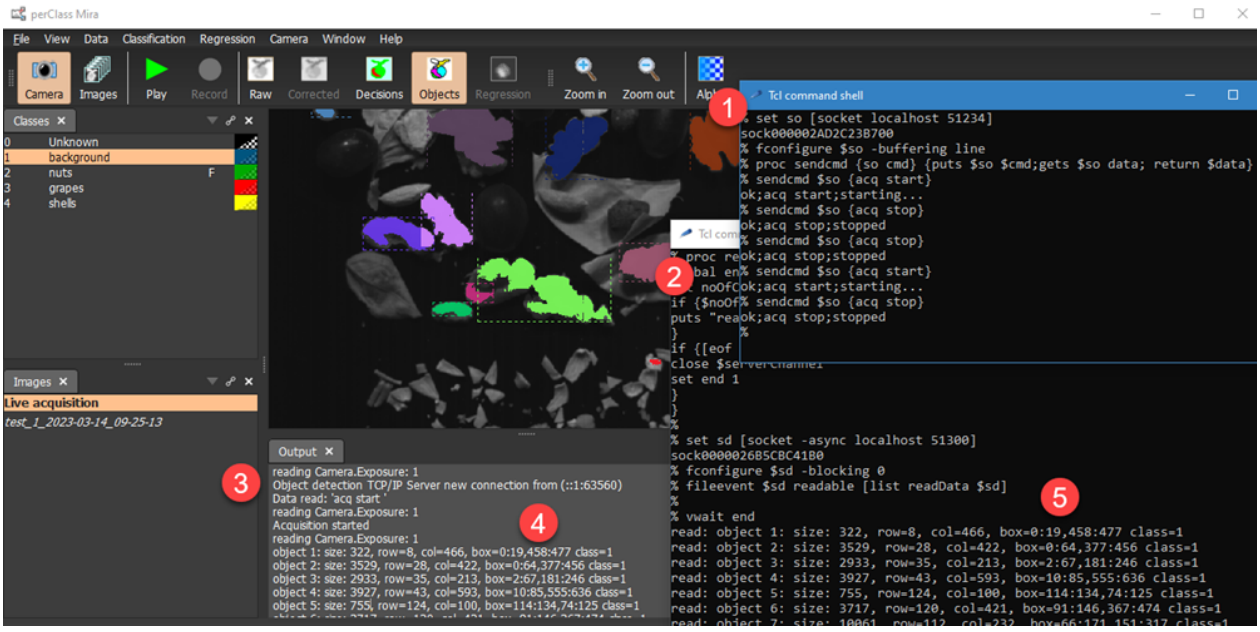
The Application Server provides the second TCP/IP channel where object detections are announced. We open the second Tcl comand shell, create an async socket and define a call back function that would be executed, when any data arrives. In the callback, we print the data sent by Application Server to 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 1 to issue commands and one shell receiving object detections 2). .Once we connect in 2 to the object detection channel on port 51300, perClass Mira *Output* window displays the connection details 3 . In the command shell 1 , we can now start the acquisition using "acq start" command (note that also "acq info enable" cammand 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 4 , the respective messages are sent over the object detection socket, captured in 5 and printed to the standard output.

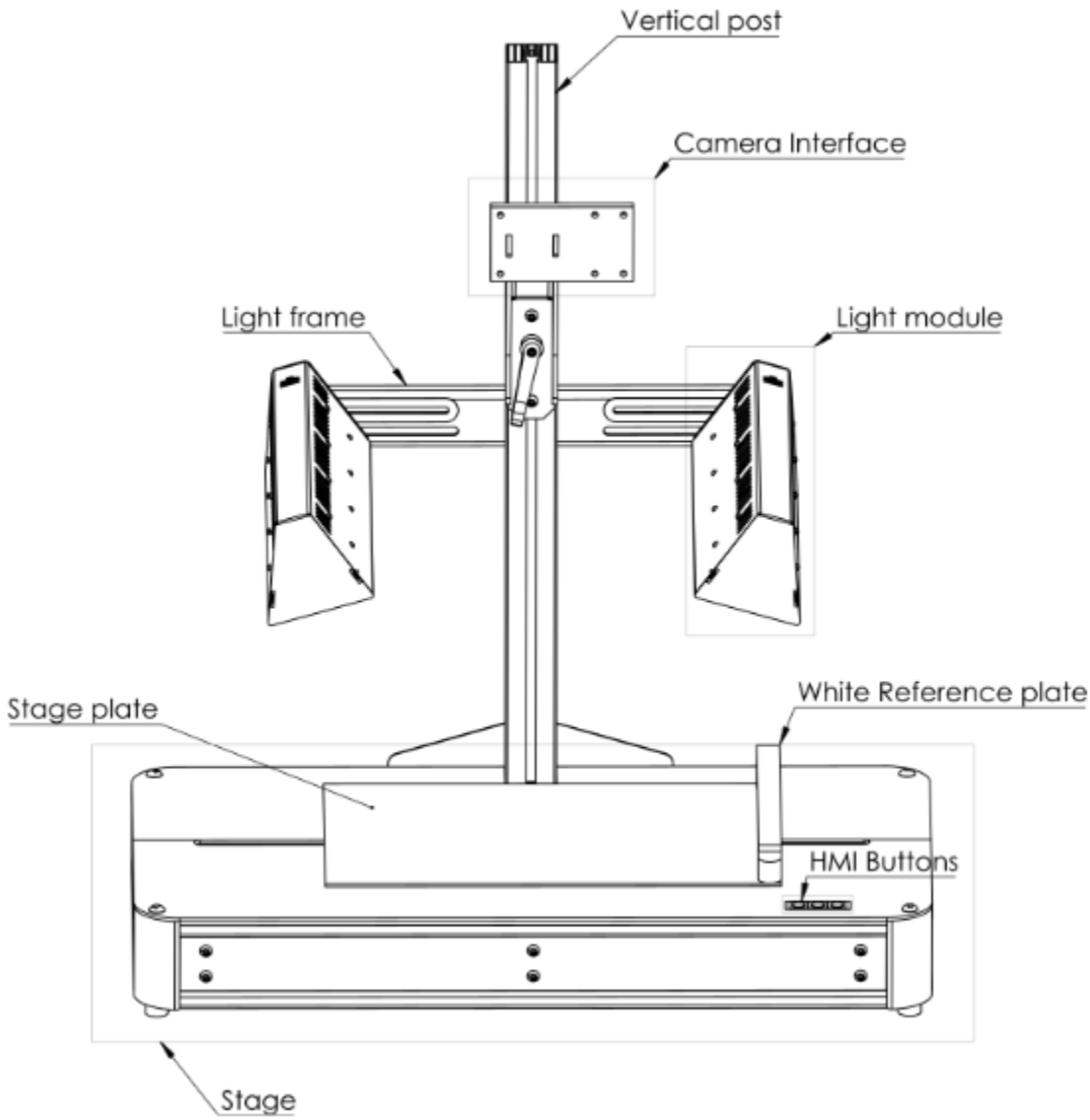


perClass Mira Stage

perClass Mira 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.

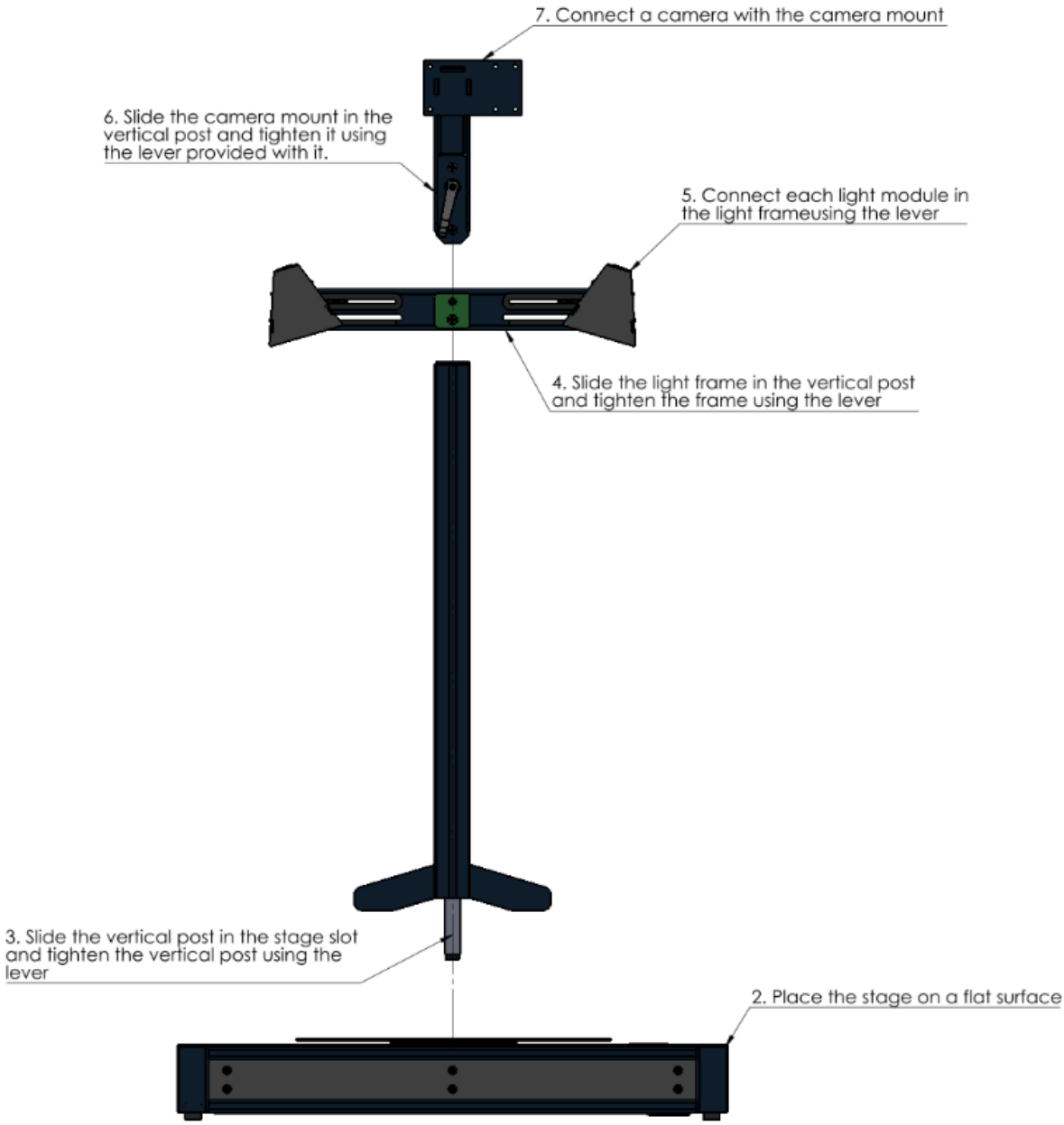
The end-user product name is "perClass Mira Stage", a bundle of the "perClass Stage" hardware and "perClass Mira" software. This is to emphasize that the perClass Stage hardware is fully operated from the perClass Mira software.

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

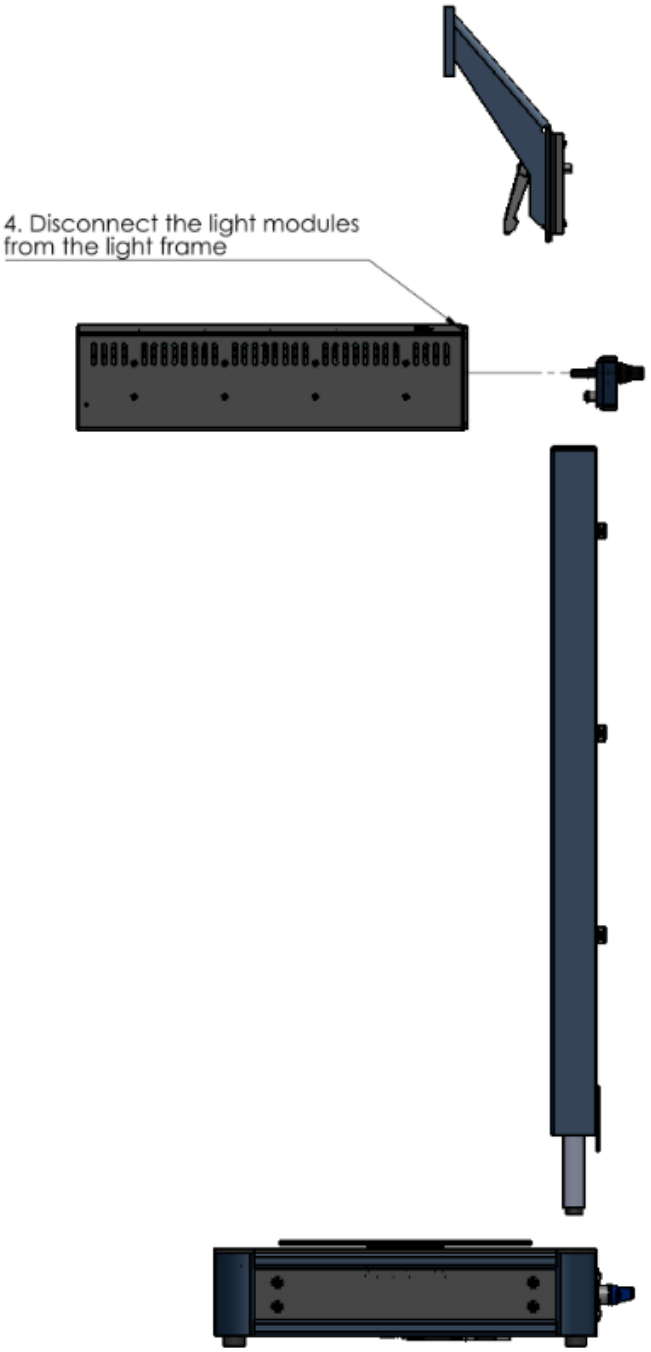
1. Remove all components from the Peli Case enclosure
2. Place the stage (base) on a flat surface
3. Slide the vertical post, with the alignment plate at the rear, in the stage slot
4. Tighten the vertical post by attaching the manual lever
5. Slide the light frame in the vertical post
6. Tighten the 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

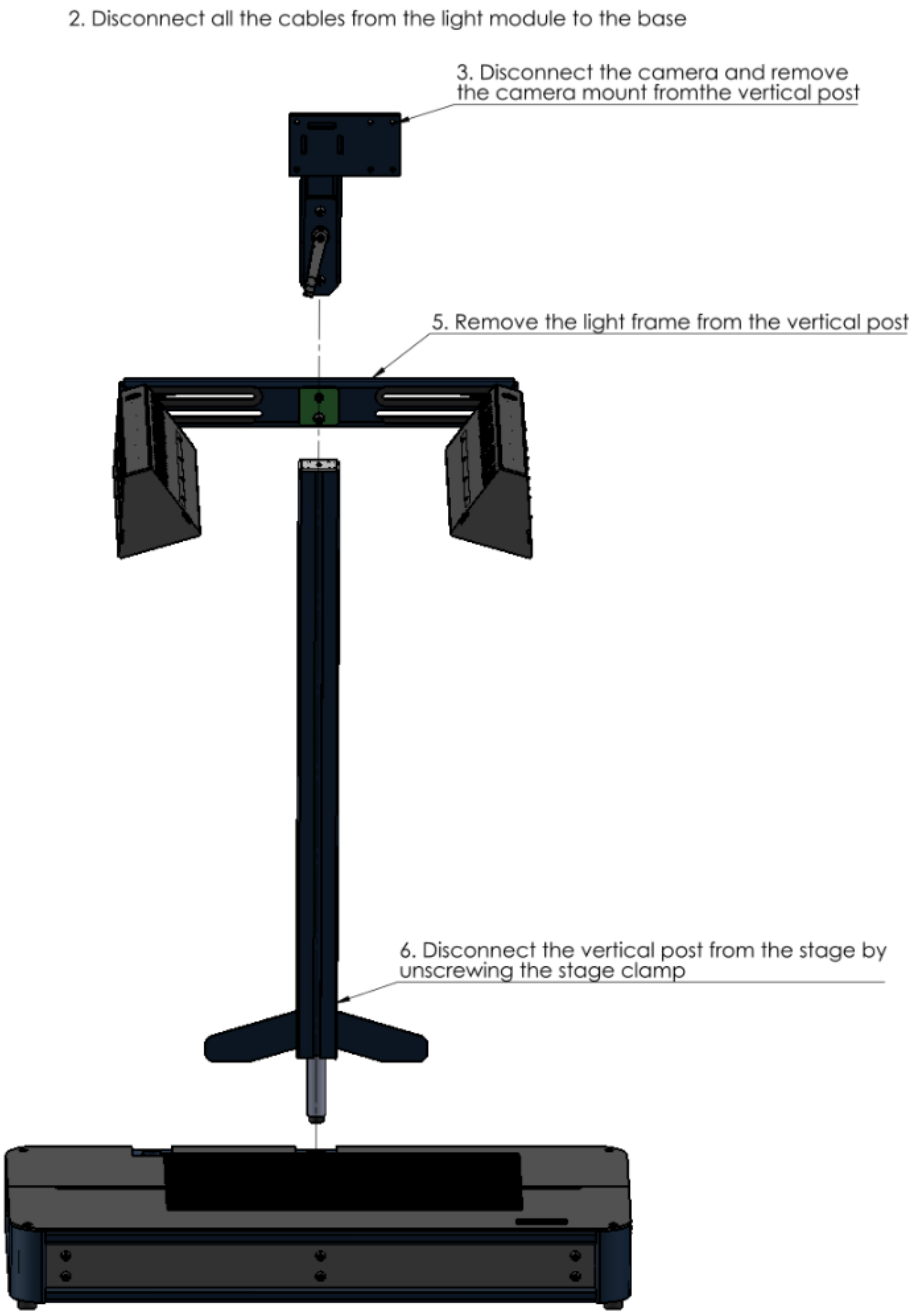


1. Remove all the components from the Peli case enclosure

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).


4. Disconnect the light modules
from the light frame



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)
    - Photonfocus SDK

Resonon

- Both VNIR and NIR systems using
    - the legacy Resonon SDK
    - NIR systems supported through AlliedVision VimbaC SDK
    - experimental support for VNIR systems using the Basler Pylon 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 adviced 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 dedicatet 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 cased 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 also a 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 acuisition library reference. Please note this is referring to "acquitision 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 pmr

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 miracq.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 openned using `miraacq_OpenDevice`

miraacq_DeviceIsSnapshot

Returns if a currently openned device is snapshot camera

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

Input:

- Runtime environment pointer

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

This function returns the type of currently openned 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 acqusition.

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 acqusition.

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 acqusition. 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_DATALAYOUT_BIP      1 /* spectrum-by-spectrum (dimensions: bands-width-
height) */
ACQ_DATALAYOUT_BIL      2 /* frame-by-frame (dimensions: width-bands-height)
*/
ACQ_DATALAYOUT_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 as 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 spectal 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 succesfully 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 miliseconds. 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 miliseconds. 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_saveImage: 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 cound 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 segmenttion

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 >= 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 >= 0 or an error code if < 0

```
MIRA_DATALAYOUT_UNKNOWN  0
MIRA_DATALAYOUT_BIP      1      /* spectrum-by-spectrum (dimensions: bands-width-height) */
MIRA_DATALAYOUT_BIL      2      /* frame-by-frame (dimensions: width-bands-height) */
MIRA_DATALAYOUT_BSQ      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 >= 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 acquisiion */
```

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 (mackBackground==1) or not (maskBackground==0). If masking is requested, the last paramer 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*` funcitons, 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 acquisiion */
```

mira_GetObjDataClassSize

For complex object segmentation, returns number of class pixels wihtin the obejct

```
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 foregound" i.e. when complex objects are segmented.

mira_GetObjDataClassFrac

For complex object segmentation, returns the fraction of class pixels wihtin the obejct

```
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 foregound" 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)

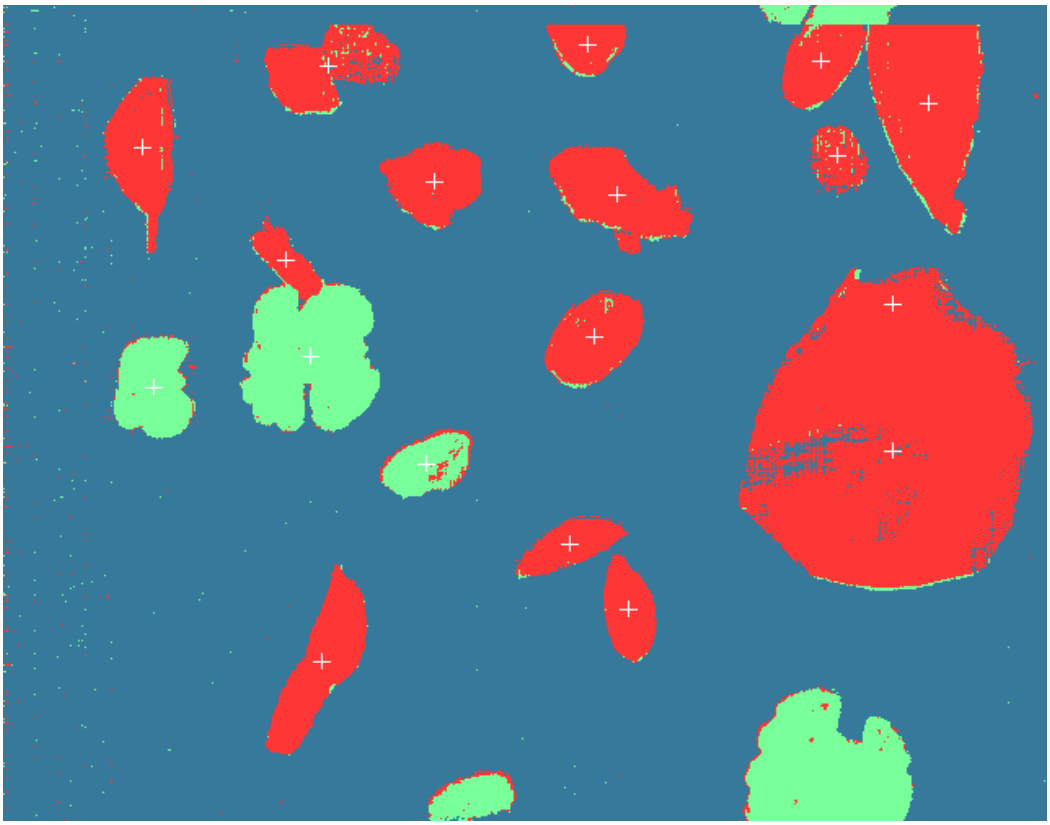
Description:
`mira_SaveImage` saves internal segmentation buffer into a PNG file. It is intended as a quick visualization of what the classifier can "see" in a deployed system.

Segmented objects are highligted 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

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 incurr a performance penalty