AnnotatorJ Documentation

This document is supplementary material to the paper "**AnnotatorJ: an ImageJ plugin to ease hand-annotation of cellular compartments**".
It has been extended for the paper "**OpSeF: Open Source Python Framework for Collaborative Instance Segmentation of Bioimages**".

Citation

Please cite our paper if you use our method:

Réka Hollandi, Ákos Diósdi, Gábor Hollandi, Nikita Moshkov, Péter Horváth (2020): "**AnnotatorJ: an ImageJ plugin to ease hand-annotation of cellular compartments**", *Molecular Biology of the Cell*, Vol. 31, No. 20, 2179-2186, https://doi.org/10.1091/mbc.E20-02-0156

Prerequisites

The software is a plugin for ImageJ/Fiji, therefore their install requirements must be fulfilled. Please install Java JDK according to your operating system. No additional requirements are needed to run the plugin.

Note: code was tested on Windows 10 with JDK8 (1.8.0_162). Maven build was also tested on Ubuntu 16.04.4. CUDA 9.0 and CuDNN 7.0 are optionally required for the GPU implementation; CPU implementation is the default.

Installation

The plugin can either be (A) ran as a standalone version which requires no installation but JRE (Java Runtime Environment as opposed to JDK: Java Development Kit), (B) installed via the Fiji updater or (C) built from source.

Please follow instructions of the version you prefer:

A) **Use standalone version**

1. Install JRE8 (https://java.com/en/download/manual.jsp)
2. enable executing .jar files:
    a. on Windows: after installing JDK/JRE double clicking on the .jar file should execute it as a program

   if it does not: right-click on the .jar file → Properties → Security tab → Advanced button → allow execution
   b. on Ubuntu: right-click on the .jar file → Permissions → check *Allow executing file as program* checkbox
3. *annotator_Project-0.0.1-SNAPSHOT.jar* can now be run as an executable

B) **Use Fiji updater**

1. Please follow this tutorial on how to add a Fiji update site to your Fiji installation: https://imagej.net/Following_an_update_site
2. When prompted for the update site info, enter the following:
   URL: https://sites.imagej.net/Spreka/
   Name: AnnotatorJ

C) **Build from source**

1. Install JDK8 (https://www.oracle.com/java/technologies/javase/javase-jdk8-downloads.html)
2. (optional) Download ImageJ (https://imagej.nih.gov/ij/download.html) or Fiji (https://fiji.sc/) (Fiji is just ImageJ).
3. Clone this repository with

```
git clone https://github.com/spreka/annotatorj.git
```

4. Install AnnotatorJ by building the project with Maven (e.g. in Eclipse) or in the command line
   a. Eclipse:
      i. Maven clean
      ii. Maven build… → Goals: clean install | Add… → Name: `scijava.app.directory` | Value: `c:\path\to\fiji\`
      where *\path\to\fiji* is typically *\FIJI\Fiji.app\*
   b. command line:
   `mvn clean install -Dscijava.app.directory=c:\path\to\fiji\`

Troubleshooting

If you encounter the following error during maven build in Eclipse:

```
[ERROR] COMPILATION ERROR :

[INFO] -------------------------------------------------------------

[ERROR] No compiler is provided in this environment. Perhaps you are running
on a JRE rather than a JDK?
```

Please set your compiler to JDK as follows. Window → Preferences → Java → Installed JREs →
if there is a jdk1.8.x_xxx in the list but jre1.8.x_xxx is selected: select a jdk1.8.x_xxx → Apply → Apply and Close
if only a jre1.8.x_xxx is present in the list → Add… → Standard VM → Next → browse JDK home with Directory… → Finish → select this JDK from the list → Apply → Apply and Close.

If Fiji Console shows this error:

```
Exception in thread "Thread-5" java.lang.NoClassDefFoundError:
org/bytedeco/hdf5/Group …
```

Please check that you have *hdf5-1.12.0-1.5.3.jar* in your Fiji folder, typically under \Fiji.app\jars\. If you do not, copy this file from the pre-built version's folder to the \Fiji.app\jars\ folder and restart Fiji.

Usage

Annotation

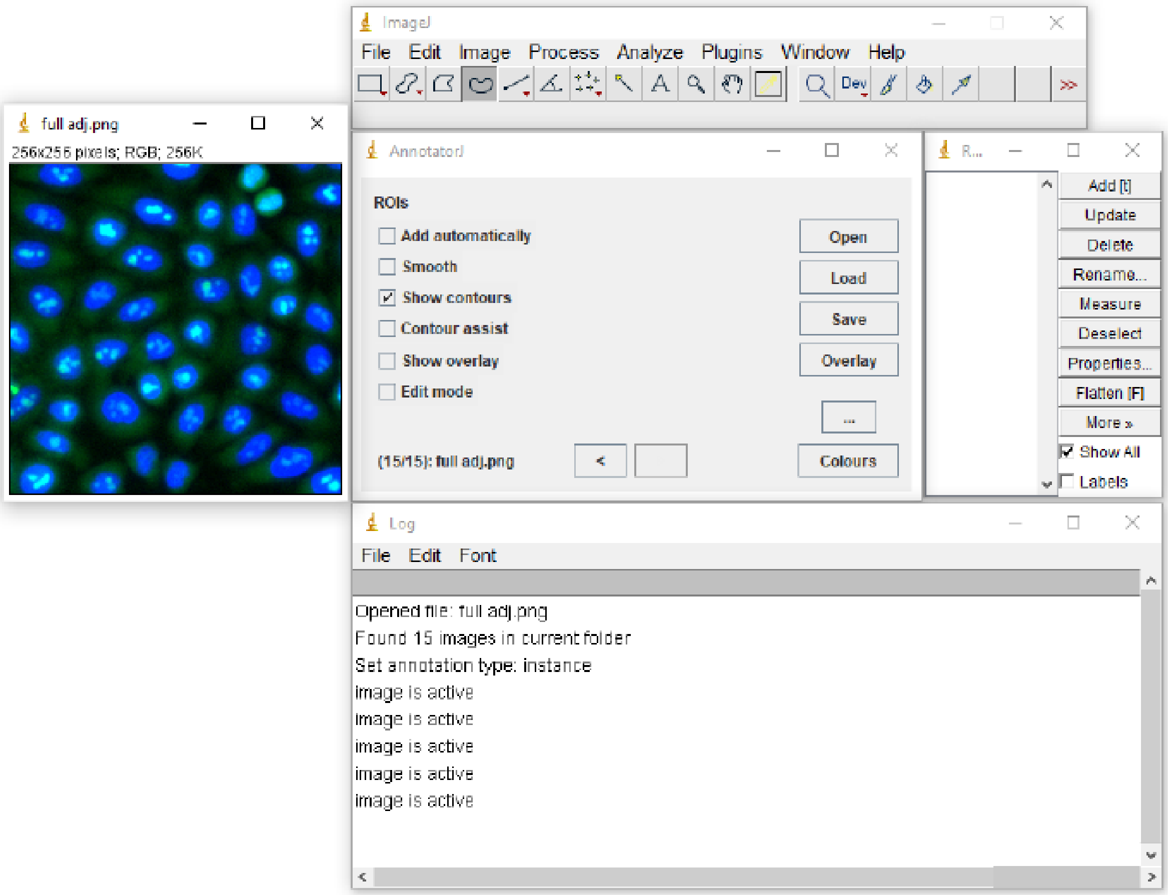
How to run the annotator plugin, AnnotatorJ as a Fiji plugin:

1. Start Fiji
2. Navigate to Plugins → AnnotatorJ

How to run AnnotatorJ from the pre-built version:

1. Locate *annotator_Project-0.0.1-SNAPSHOT.jar* in your file system
2. Run it

In the main window of the plugin an image must be opened first with the Open button then annotation type must be selected in a new popup window (see **Fig.2**). The default annotation type is instance, selecting it will also open a ROI Manager. The windows shown on **Fig.1.** will appear.



```
Opened file: full adj.png
Found 15 images in current folder
Set annotation type: instance
image is active
image is active
image is active
image is active
image is active
```

Figure 1. AnnotatorJ windows in instance annotation mode. *Free-hand selection tool is automatically selected in the main ImageJ/Fiji window, the main AnnotatorJ window contains the controls in the right to open/load an image/previous annotation, save annotations and further options. Checkboxes offer more functionality. The Log window displays information about the processes.*

The following figures demonstrate the functionalities of the plugin.

Annotation types - Open button

Figure 2 shows the annotation type selector window while **Figure 3** offers example images for each type and how they can be exported in the exporter of the plugin.

In instance annotation individual objects are defined by their contours or borders, separating the touching objects. Semantic annotation only separates objects from the background but not each other. "bbox" stands for bounding box, the smallest rectangle that an object fits in, for each object individually.

Multi-label export results in a grayscale image (mask) where each individual object is labelled with a different gray intensity thus they are separated. A binary mask contains all objects as white and everything else on the image is part of the background marked black. A multi-layer or stack mask contains each object on a different layer or slice, it is optimal for storing overlapping objects. Coordinates corresponds to a single .csv text file where the bounding box coordinates of objects are listed in each row as [top left x, top left y, width, height] according to the COCO dataset bounding box format.

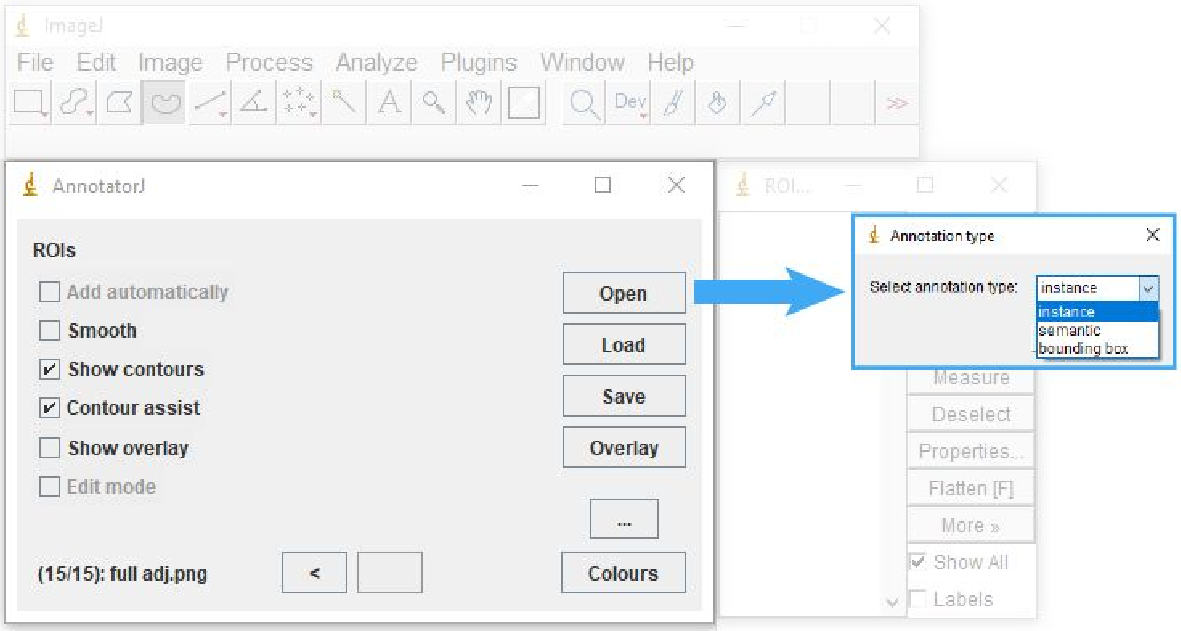


Figure 2. Annotation type selector. *Opens after an image has been selected to annotate. See Fig.3. for the possible annotation types.*

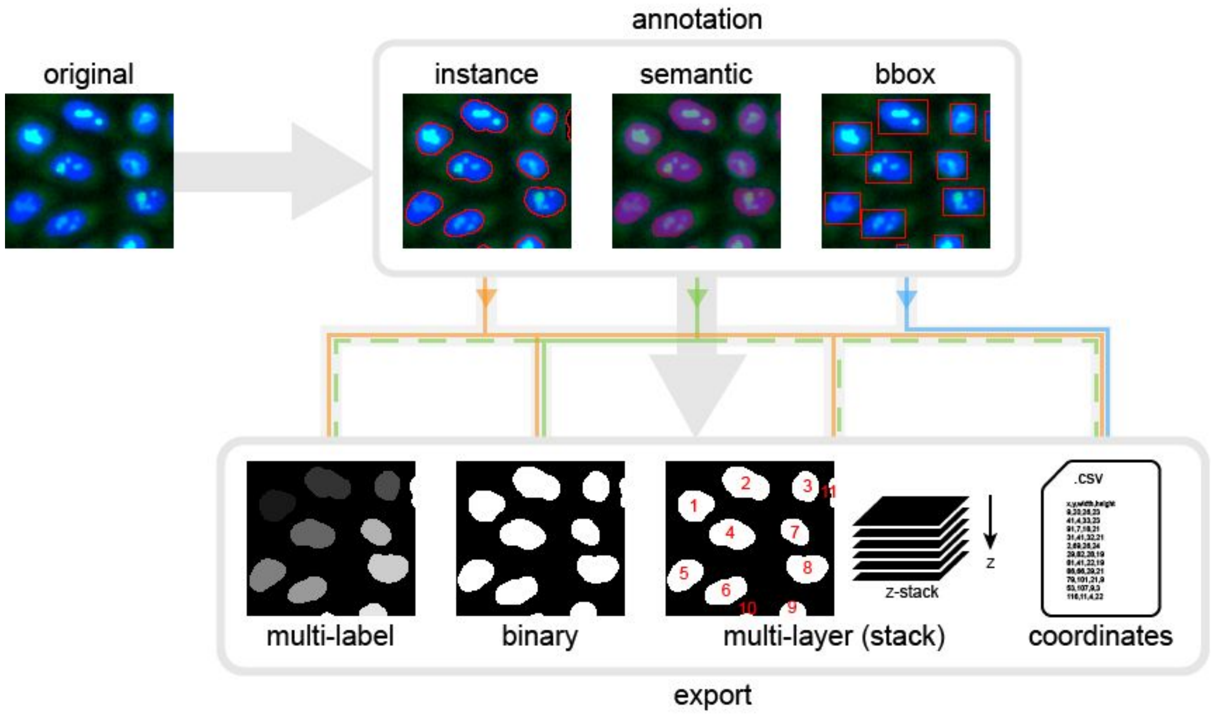


Figure 3. Annotation and export types represented on an example image. *The upper row displays the annotation types for the example original image on the left. On the bottom part export options are shown, the colour-coded arrows demonstrate which export option is possible from the available annotation types each. Dashed lines indicate that though the conversion is possible, the result might not be as expected: when exporting a semantic annotation as multi-label, multi-layer of coordinates file, the individual objects can only be separated if they do*

not touch on the semantic annotation image (as shown in this figure). Orange corresponds to instance, green to semantic and blue to bounding box.

Multi-class annotations - Load and Overlay buttons

A previous annotation file can be loaded into the plugin with the Load button for the currently opened image. This can be used to e.g. continue a previously created, incomplete annotation for a larger image.

The button Overlay opens a previous annotation file and displays it on the currently opened image as an overlay with the current overlay colour, as in **Figure 4**. This functionality aims to help distinguish between different object types present on the same image. On this example image, a previous object class is highlighted in red, the currently annotated object type in yellow, so that the user would not have to decide the object class again when annotating similarly looking classes.

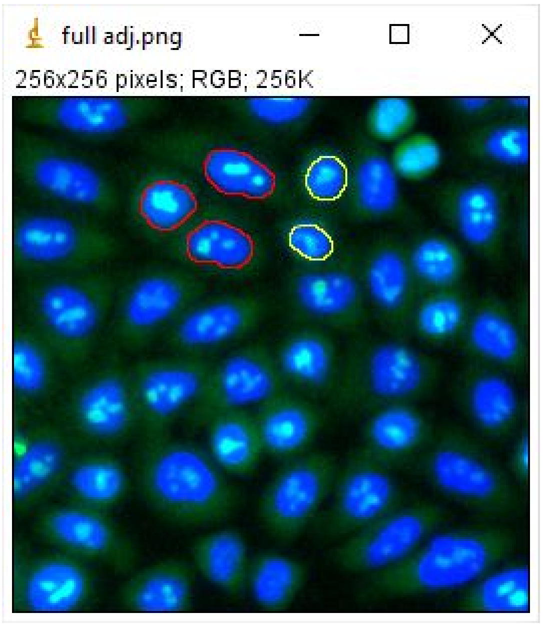


Figure 4. Overlayed annotations. *Red contours show a previously annotated object type (e.g. a specific phenotype like healthy cells) overlayed, while yellow contours are the currently annotated objects.*

Colours - Colours button

Overlay and annotation colours can be set with the Colours button (see **Fig.5**). The default ImageJ colours are offered in the drop-down lists: yellow, black, blue, cyan, green, magenta, orange, red, white.
In semantic annotation mode the annotation colour is overlayed on the original image with a 40% opacity (the annotation looks semi-transparent).

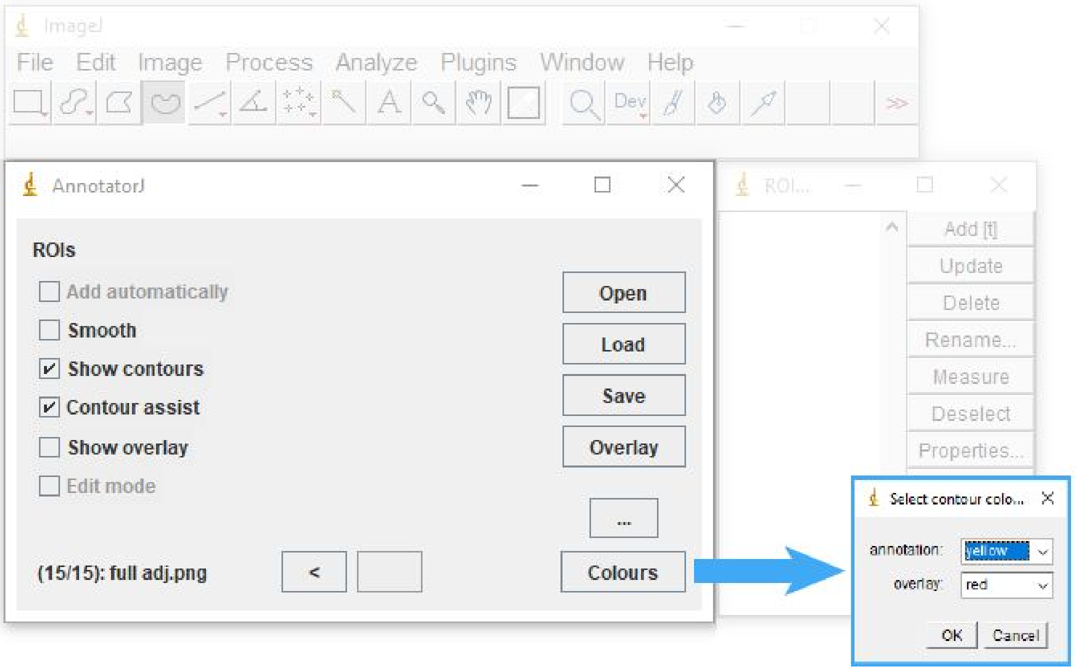


Figure 5. Colour selector for annotation and overlay. *It is recommended to use different colours when an overlay is displayed to better distinguish the objects.*

Saving annotations by class - Save button *(See OpSeF-related functionality later)*

When the annotation is finished, it can be saved with the Save button. This will open a class selector window (see **Fig.6**) where some pre-defined object classes are listed by default, and custom classes can be added by selecting "other...": the text field below the list will be activated and the name of the new object class can be typed.

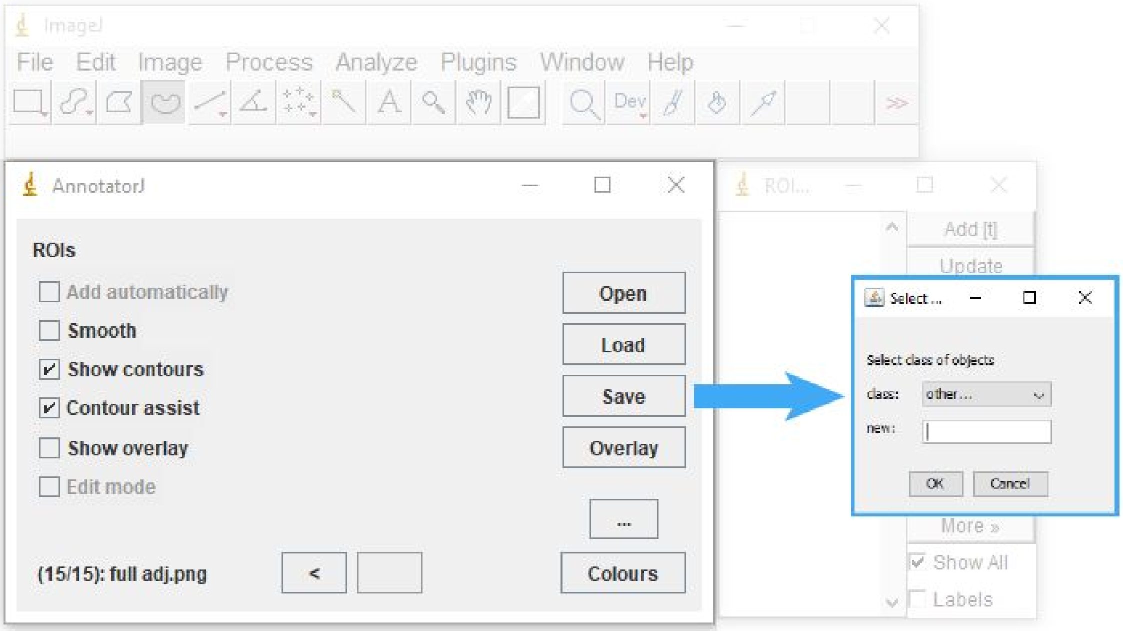
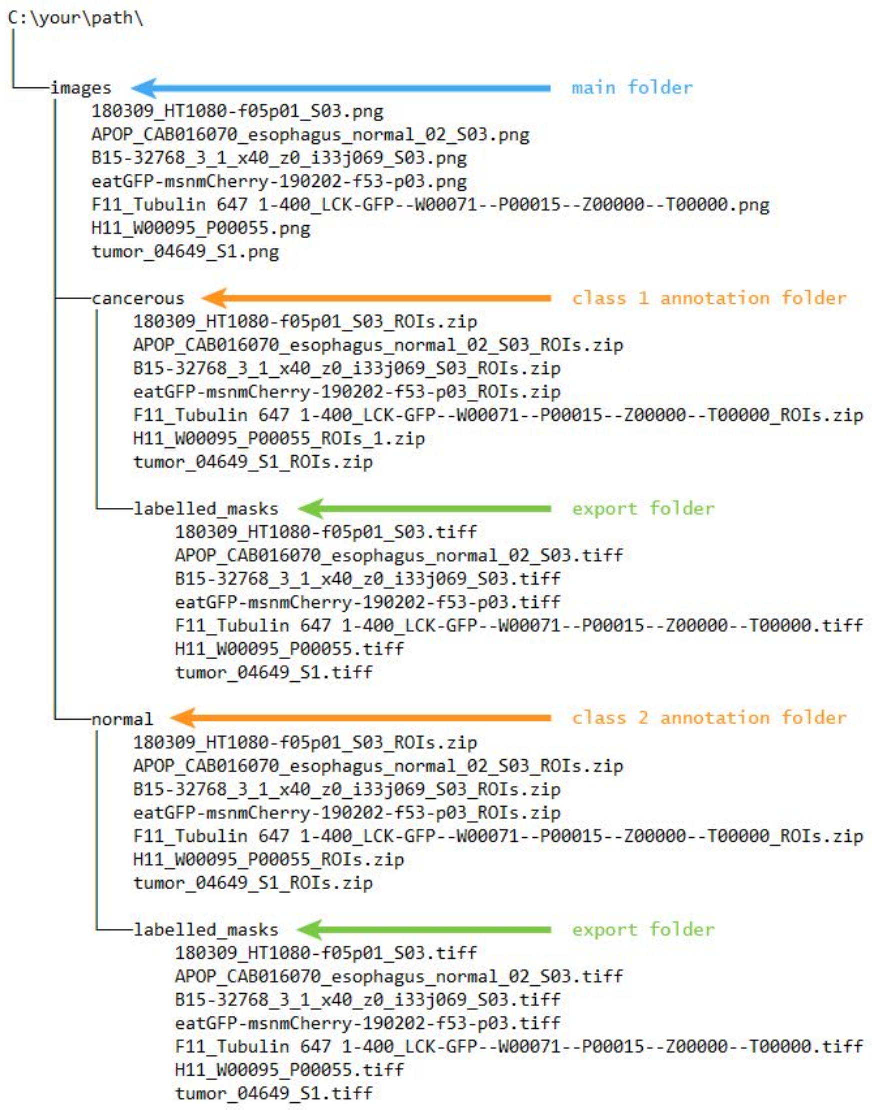


Figure 6. Class selector. When saving the annotations of any annotation type a class (or phenotype) must be selected. A new class can be added by selecting "other..." and typing the name of the new class.

These object classes might represent certain phenotypes in case of the example cell culture microscopy image shown in **Figure 1,3,4** or an everyday object like car, bicycle, mug, person etc. as in general object detection tasks.

Folder structure

The annotation files will be saved in a separate folder under the currently opened image's folder with the name of the object class selected. **Figure 7** shows the folder structure created by the plugin, also including export folders. The plugin expects that the images to be annotated are located in a single folder anywhere in the file system, and will create additional folders under this image folder.



```
C:\your\path\
│
└──images  <──────────────── main folder
    │  180309_HT1080-f05p01_S03.png
    │  APOP_CAB016070_esophagus_normal_02_S03.png
    │  B15-32768_3_1_x40_z0_i33j069_S03.png
    │  eatGFP-msnmCherry-190202-f53-p03.png
    │  F11_Tubulin 647 1-400_LCK-GFP--W00071--P00015--Z00000--T00000.png
    │  H11_W00095_P00055.png
    │  tumor_04649_S1.png
    │
    ├──cancerous  <──────────── class 1 annotation folder
    │   │  180309_HT1080-f05p01_S03_ROIs.zip
    │   │  APOP_CAB016070_esophagus_normal_02_S03_ROIs.zip
    │   │  B15-32768_3_1_x40_z0_i33j069_S03_ROIs.zip
    │   │  eatGFP-msnmCherry-190202-f53-p03_ROIs.zip
    │   │  F11_Tubulin 647 1-400_LCK-GFP--W00071--P00015--Z00000--T00000_ROIs.zip
    │   │  H11_W00095_P00055_ROIs_1.zip
    │   │  tumor_04649_S1_ROIs.zip
    │   │
    │   └──labelled_masks  <──────── export folder
    │          180309_HT1080-f05p01_S03.tiff
    │          APOP_CAB016070_esophagus_normal_02_S03.tiff
    │          B15-32768_3_1_x40_z0_i33j069_S03.tiff
    │          eatGFP-msnmCherry-190202-f53-p03.tiff
    │          F11_Tubulin 647 1-400_LCK-GFP--W00071--P00015--Z00000--T00000.tiff
    │          H11_W00095_P00055.tiff
    │          tumor_04649_S1.tiff
    │
    └──normal  <──────────────── class 2 annotation folder
        │  180309_HT1080-f05p01_S03_ROIs.zip
        │  APOP_CAB016070_esophagus_normal_02_S03_ROIs.zip
        │  B15-32768_3_1_x40_z0_i33j069_S03_ROIs.zip
        │  eatGFP-msnmCherry-190202-f53-p03_ROIs.zip
        │  F11_Tubulin 647 1-400_LCK-GFP--W00071--P00015--Z00000--T00000_ROIs.zip
        │  H11_W00095_P00055_ROIs.zip
        │  tumor_04649_S1_ROIs.zip
        │
        └──labelled_masks  <──────── export folder
               180309_HT1080-f05p01_S03.tiff
               APOP_CAB016070_esophagus_normal_02_S03.tiff
               B15-32768_3_1_x40_z0_i33j069_S03.tiff
               eatGFP-msnmCherry-190202-f53-p03.tiff
               F11_Tubulin 647 1-400_LCK-GFP--W00071--P00015--Z00000--T00000.tiff
               H11_W00095_P00055.tiff
               tumor_04649_S1.tiff
```

Figure 7. Folder structure used by AnnotatorJ and AnnotatorJExporter. The main folder containing the images is called "images" in this example, it can be any folder of your choice, the

example folder contains 7 .png image files. Annotations are saved to folders named after the selected class, here "cancerous" and "normal" (these are the default classes offered in the plugin, they can be any class of your choice). Under these folders, the annotation files are named [original file name]_ROIs.zip in instance annotation mode. When exporting the annotations, the exported files, here multi-layered masks are saved to the subfolder "labelled_masks" as 16-bit .tiff image files, named [original file name].tiff. Additional export options create new subfolders under the selected class's folder.

Instance and bounding box annotations are saved in .zip format, semantic annotations in .tiff format. Their name is generated automatically from the original image file's name:

- instance: [original image name]_ROIs.zip
- semantic: [original image name]_semantic.tiff
- bounding box: [original image name]_bboxes.zip

If a file with the automatically generated name already exists in the output folder corresponding to the class selected in saving, increasing numbers are appended to the end of the file name:

[original image name]_[annotation type]_[number].[zip or tiff]

where [annotation type] can be either "ROIs", "semantic" or "bboxes", as above.
About exported file names, please see the section ***Export*** and **Table 1** below.

The plugin lists all image files found in the main folder of images to annotate and displays the currently opened image's name and place in the image list (see **Fig.8**). The arrow buttons ("<" and ">") can load the previous or next image in the list, respectively. Before actually opening a new image for annotation or quitting, the plugin always displays a confirmation window to save the current annotations. If the first image in the list is opened, the previous ("<") button is inactive. Similarly, if the last image in the list is opened, the next (">") button is inactive.

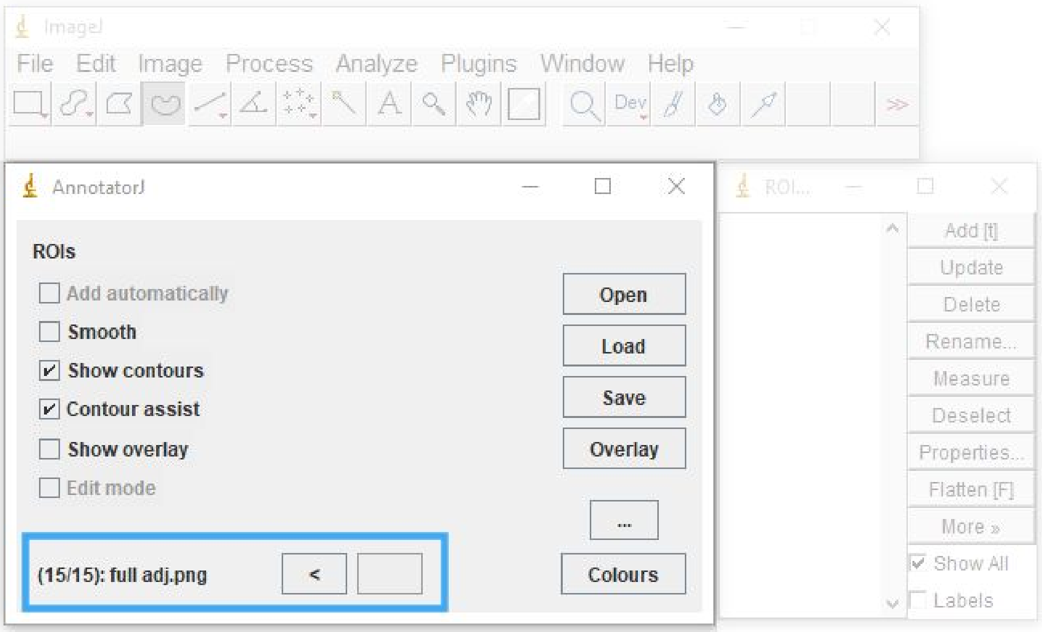


Figure 8. Image file list controls. *In the example, the image folder contains 15 images of which the 15th image, named "full_adj.png" is currently opened. Since this is the last image in the list, the next (">") button is inactive (the arrow is not shown), only the previous ("<") button.*

Annotation options - "..." button and checkboxes

Figures 9-10 show the annotation options offered in the plugin: selectable options are marked by checkboxes on the left side of the main window and "..." button opens a new window of options for *Contour assist* mode and semantic annotation type.
Let us see each option after these figures.

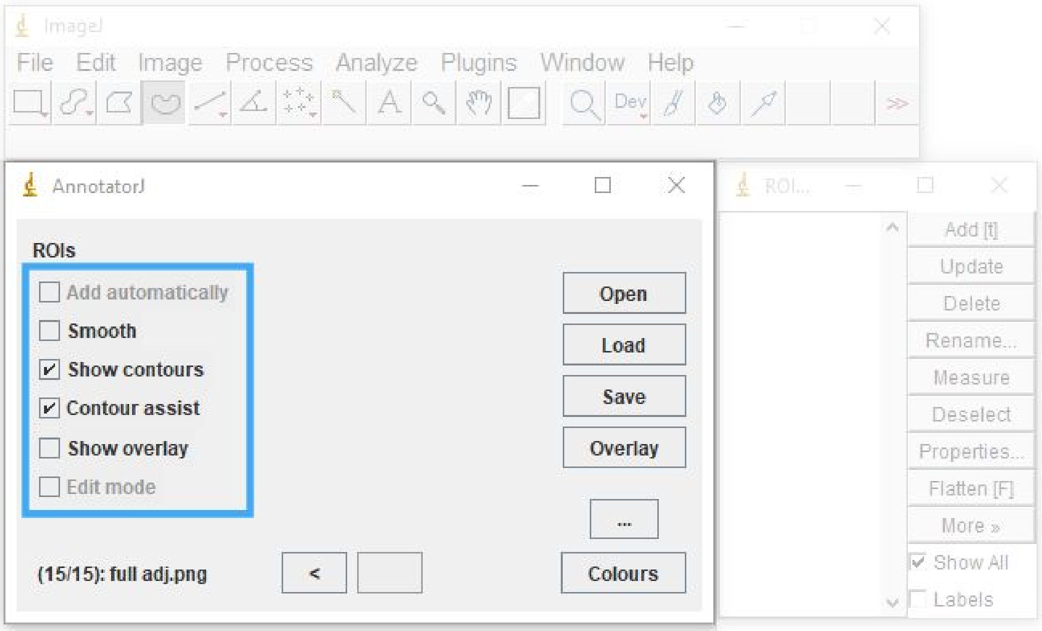
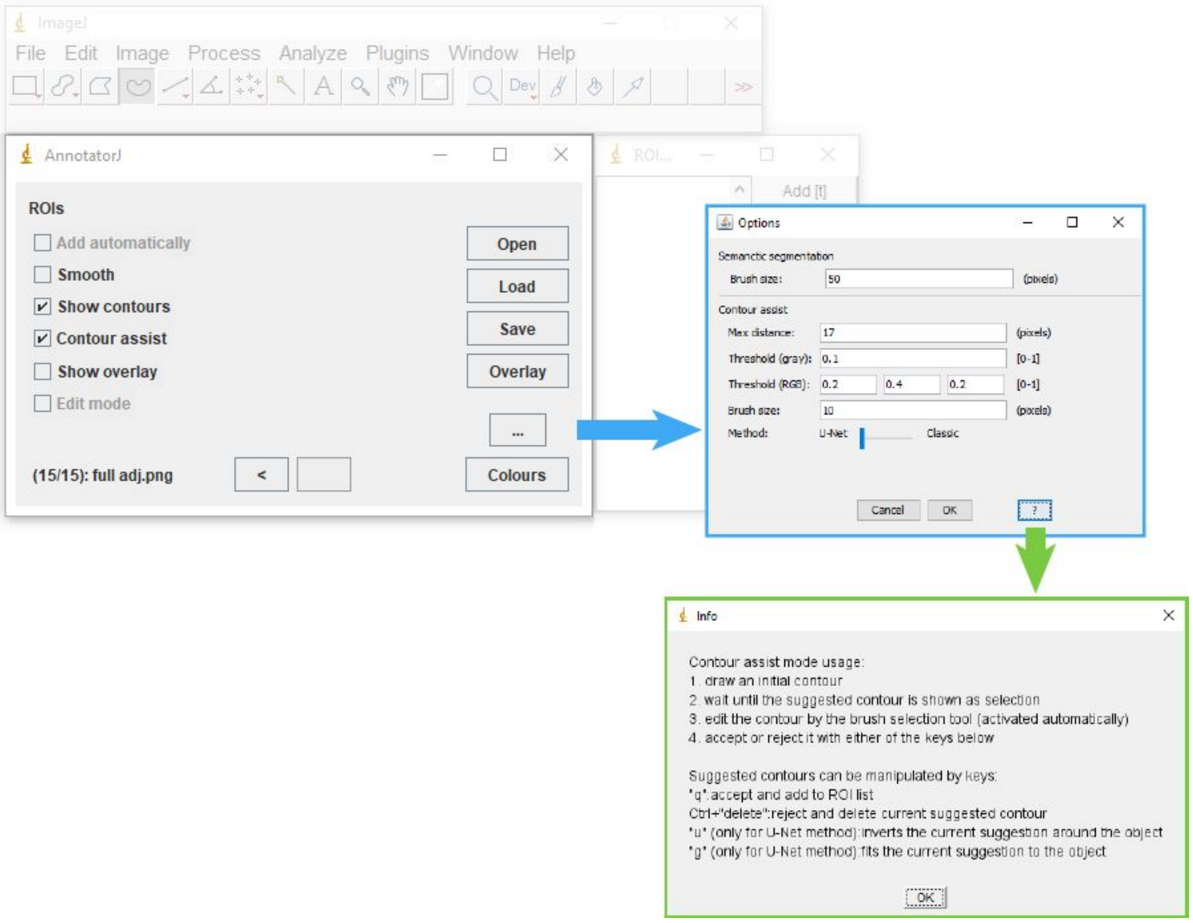


Figure 9. Checkbox options. *By default only the Show contours option is selected. In this example Contour assist is also selected inactivating both Add automatically and Edit mode options.*



Semanctic segmentation
Brush size: 50 (pixels)

Contour assist
Max distance: 17 (pixels)
Threshold (gray): 0.1 [0-1]
Threshold (RGB): 0.2 0.4 0.2 [0-1]
Brush size: 10 (pixels)
Method: U-Net Classic

Contour assist mode usage:
1. draw an initial contour
2. wait until the suggested contour is shown as selection
3. edit the contour by the brush selection tool (activated automatically)
4. accept or reject it with either of the keys below

Suggested contours can be manipulated by keys:
"q":accept and add to ROI list
Ctrl+"delete":reject and delete current suggested contour
"u" (only for U-Net method):inverts the current suggestion around the object
"g" (only for U-Net method):fits the current suggestion to the object

Figure 10. Options window. *Additional info about the options and key bindings of the plugin can be opened with the "?" button.*

Add automatically

This option is only available in instance or bounding box annotation types. In instance annotation, the default tool selected by the plugin is the *Freehand selection* tool of ImageJ/Fiji. When the user finishes drawing a contour and releases the left mouse button used to draw it, the contour will be automatically added to the ROI Manager if this option is selected. Otherwise, the default key binding of "t" must be pressed to add a new contour to the ROI Manager. This option can be used to ensure that no annotation drawn is missing from the ROI list when annotating a large number of objects.
Contours added to the ROI list are named by increasing numbers: 0001, 0002, etc. When loading previous annotations, the loaded contours will be numbered after any existing contour numbers present in the current ROI Manager.

Smooth

Smoothing is only available in instance annotation. When selected, the currently drawn contour appears smoothly around the pixelated borders of an image object as in **Figure 11**. Smoothing appears after drawing is finished (left mouse button is released). This function does not affect the saved ROIs and how the exported masks will represent the objects.

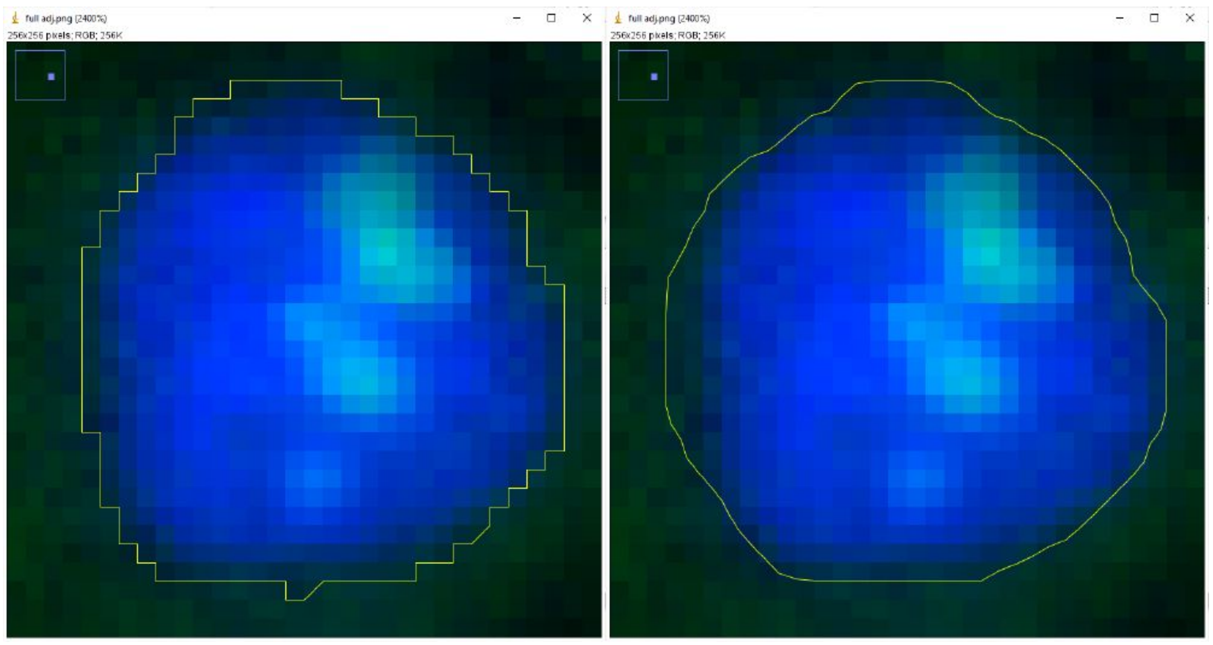


Figure 11. Smoothing option. *An example instance annotation of a nucleus marked in yellow on the left, and its smoothed version on the right.*

Show contours

When selected, objects already added to the ROI Manager are displayed in the currently selected annotation colour. Otherwise no contour of the current annotation is displayed. This function works as *Show labels* in ROI Manager but also affects semantic overlays (that cannot properly be represented in the ROI list).

Contour assist mode

This functionality is currently only available in instance annotation type. By default it is set to "U-Net", a deep learning method. It uses a previously trained U-Net model to help annotate a certain class of objects. The model can be trained in the DL4J framework in Java or in Python. Only Keras models can be loaded in the plugin.
The pre-trained U-Net model is used to pre-segment the image and offer a suggested contour based on an initial contour drawn by the user. The initial contour is expanded by x pixels in each direction to create a rectangle around the current object. x is defined in the options window (Max distance) that can be opened with the "..." button of the main window. Within this rectangle the predicted objects are obtained internally and the largest of them is returned as the suggested contour. The *Brush selection* tool is automatically selected to enable refining (editing) of the contour to the true borders of the object.
This plugin is intended to aid hand-annotation, not to replace it with predictions from existing algorithms. The suggested contour cannot be added automatically to the ROI list, this option is disabled when using *Contour assist* mode. It is possible that the algorithm returns an inverted contour of the object, it can be inverted by pressing the key "u". The suggested contour can be fitted to the object using the active contours algorithm by pressing the key "g". The contour should be revised and corrected where needed with the *Brush selection* tool.

When the refined contour is ready, it can be added to the ROI list by pressing the key "q" or discarded with "Ctrl"+"delete".

The following figure (**Fig.12**) shows the steps in Contour assist mode.

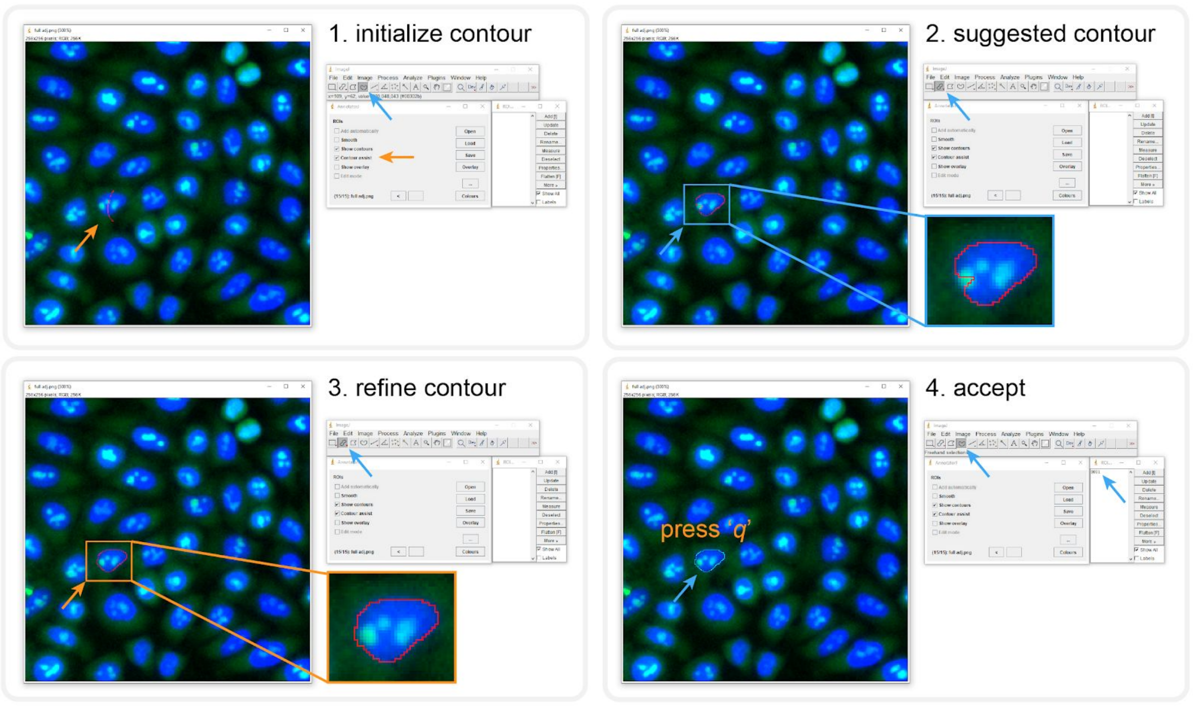


Figure 12. Contour assist mode usage. *Steps are numbered in order of execution. Orange indicates user interaction, blue shows automatic steps. 1) Initialize the contour with a lazily drawn line, 2) wait for the suggested contour to appear (a window is shown until processing completes), brush selection tool is selected automatically, 3) refine the contour as needed, 4) accept it by pressing the key 'q' or reject with 'Ctrl'+'delete'. Alternatively, the suggested contour might be inverted with 'u'. Accepting adds the ROI to ROI Manager with a numbered label.*

To use suggested contours (in U-Net mode), please make sure you have a pre-trained keras U-Net model required by this function of the plugin. The default location AnnotatorJ looks for the model is under the folder \plugins\models\ and the default name of the model files are *model_real.json* and *model_real_weights.h5*. The plugin comes with a pre-trained model for nucleus detection. Such a keras model can be saved from Python with the following commands:

```
# save model as json
model_json=model.to_json()
with open('model_real.json', 'w') as f:
       f.write(model_json)

# save weights too
model.save_weights('model_real_weights.h5')
```

By default, as in the pre-built version, CPU implementation of U-Net prediction is used to support a wider range of potential users' hardware configurations. This can be switched to

GPU implementation in the pom.xml of the Maven project (source code version) in this section of the file regarding the dependency *nd4j*:

```
    <dependency>
           <groupId>org.nd4j</groupId>
           <!-- for CPU + all OS jars-->
           <!-- <artifactId>nd4j-native-platform</artifactId> →
           <!-- for CPU only on current OS-->
           <artifactId>nd4j-native</artifactId>
           <!-- for GPU + all OS jars -->
           <!-- <artifactId>nd4j-cuda-9.2-platform</artifactId> -->
           <version>${nd4j.version}</version>
</dependency>
```

Further options are available in the Options window ("..." button) for this function:

- Max distance: number of pixels to expand the initial contour with to create a bounding box within which the suggestion process is done
- Threshold (gray): floating-point value in the range [0-1] indicating a grayscale threshold to use in region growing. It is only used when the Method is set to "Classic".
- Threshold (RGB): the same as for gray threshold, but 3 individual values can be set for the RGB channels.
- Brush size: *Brush selection* tool size to use when refining the contour.
- Method: can be either "U-Net" or "Classic"; the method to create a suggested contour with.

These can be seen on **Fig.10**. An additional setting in the Options is semantic segmentation brush size, which determines the size of *Brush selection* tool when used in semantic annotation.
Classic method refers to the conventional region growing algorithm that enables fast contour suggestion. This method might not result in satisfying suggested contours.

Show overlay

This options activates the contours in instance annotation mode previously loaded with the Overlay button. This function aims to help distinguish between multiple object classes present on the image. On **Figure 4** an example overlay is displayed in red.

Edit mode

Edit mode is only available in instance annotation and is disabled when using Contour assist. Activating this option disables Contour assist and Add automatically options to ensure that edited contours are stored properly. When selecting this option, the contours already added to the ROI Manager can be selected by clicking on one on the original image. The selected contour's colour is inverted to highlight it. The *Brush selection* tool is automatically selected for editing the contour. When finished editing the edited contour can be saved by pressing the keys "Ctrl"+"q" simultaneously which will update the currently edited contour in the ROI list to this new version of it, highlight it in the colour corresponding to newly added contours and finally the *Freehand selection* tool is reactivated. This is demonstrated on **Figure 13** below.

The edited contour can be discarded by pressing “Esc” and its original version will be restored. The selected contour can be deleted from the ROI list (and from the image) by pressing “Ctrl”+”delete”.

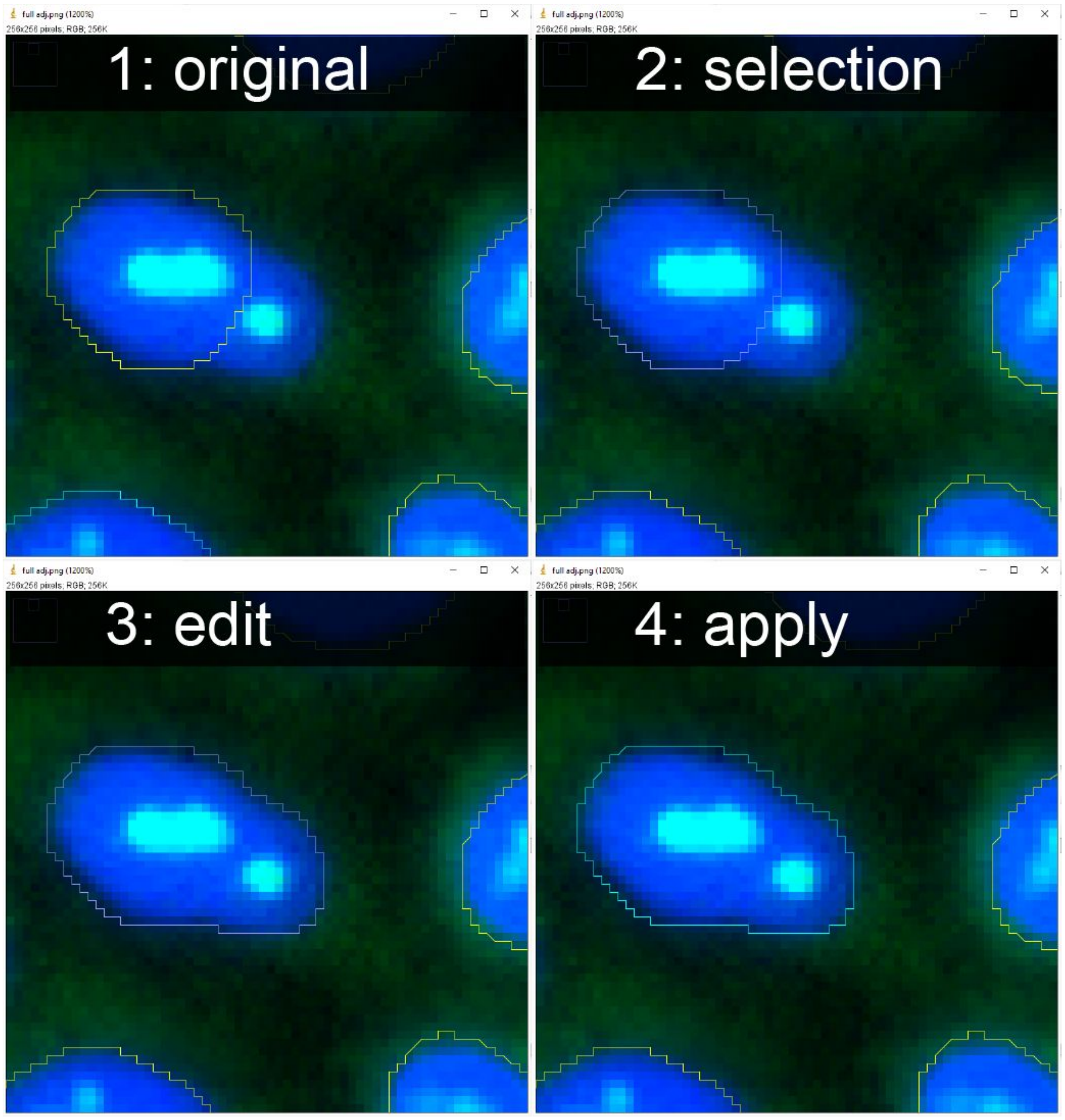


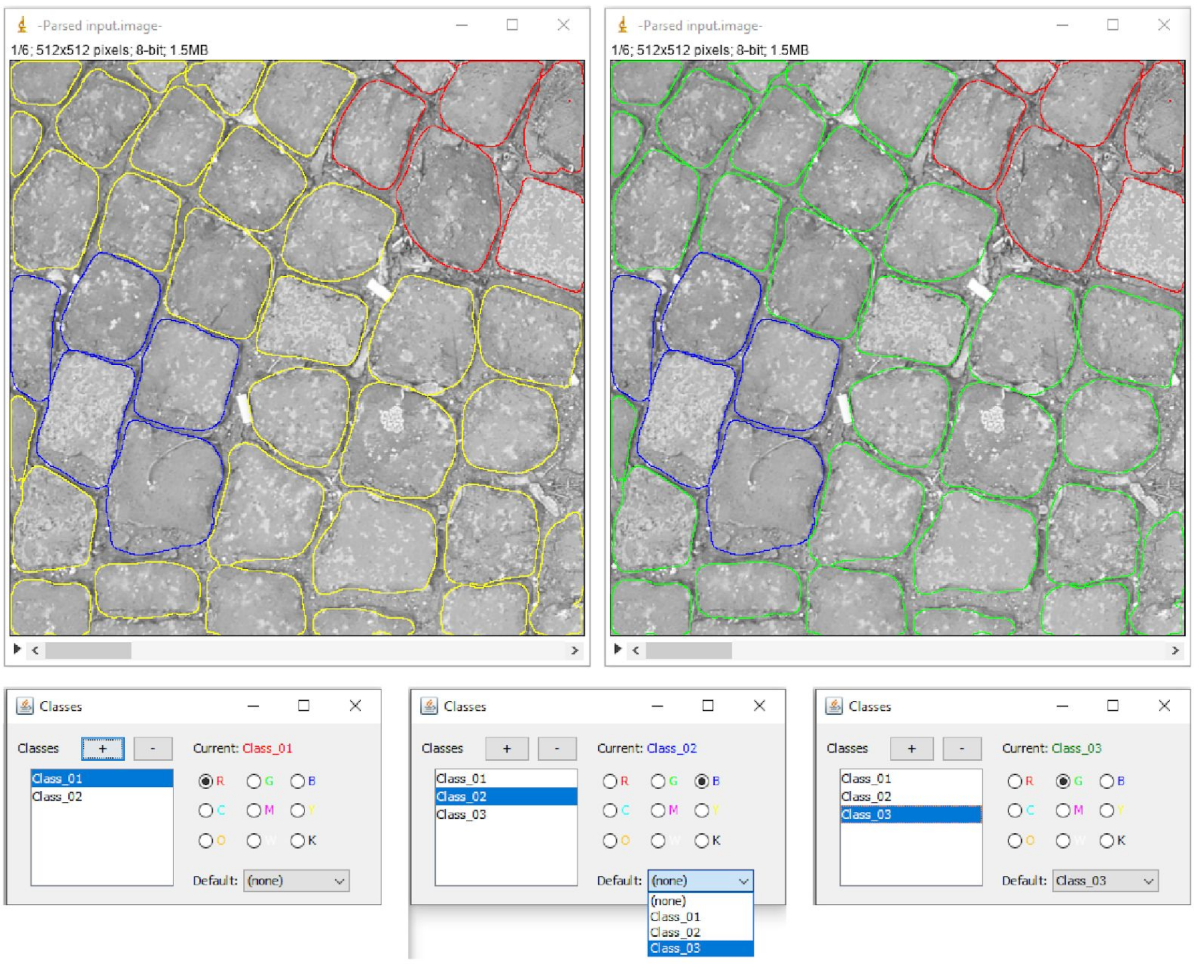
Figure 13. Edit mode. *The order of steps are displayed. The original contours are highlighted in yellow (1), the target object is selected by clicking on it with the left mouse button which changes the highlight colour (2), the contour can be edited with the Brush selection tool (3) then accepted by pressing “Ctrl”+”q” (4).*

Class mode

Class mode is only available in instance annotation and is disabled when using Contour assist or Edit mode. Activating this option disables Contour assist, Edit mode and Add automatically options. As opposed to saving classes, this mode enables object-based class annotation.

It opens a Classes window as seen on **Figure 14** with 2 initial classes by default. New classes can be added with the "+" button or deleted with the "-" button. Deleting a class will unclassify all objects belonging to the class selected for deletion. Newly added classes will have the first unused class colour assigned. Class colour of the currently selected class can be changed by clicking on the desired colour's radio button. Selecting a new colour will update the displayed contour colours on the original image.
Note: if the currently used annotation colour is selected for any class, the class annotation will not be visible! It is advised to use different colours.

Similarly to Edit mode, an object from the current ROI list can be selected by clicking on it on the original image. This will assign the currently selected class to this object and change its contour colour to the class's colour. Selecting an object already classified to one of the classes will unclassify it if it belongs to the currently selected class, or change its class to the currently selected one otherwise.


-Parsed input.image-
1/6; 512x512 pixels; 8-bit; 1.5MB

-Parsed input.image-
1/6; 512x512 pixels; 8-bit; 1.5MB

Classes
Current: Class_01
Class_01
Class_02
R [✓ R] G B C M Y O W K
Default: (none)

Classes
Current: Class_02
Class_01
Class_02
Class_03
[✓ B]
Default: (none)
(none)
Class_01
Class_02
Class_03

Classes
Current: Class_03
Class_01
Class_02
Class_03
[✓ G]
Default: Class_03

Figure 14. Class mode. *Original images before and after default class selection are shown in the upper row. 3 instances of the Classes window show 1) the initial 2-class state, 2) the default class selection after adding an extra class and selecting blue for Class_02, and finally 3) displaying Class_03 as the default class with green colour. The objects marked with red (class 1) and blue (class 2) were clicked to be assigned to their respective classes. The remaining*

objects were automatically added to the default class after selecting it in the drop-down list. Example image shown is a sample from the OpSeF data structure (see later).

Configuration

Options can be saved in the configuration file *AnnotatorJconfig.txt*. This file is created when running the plugin for the first time and is saved in the default model folder. Changing options in the plugin that are listed in the file will be saved and re-applied when restarting ImageJ/Fiji.

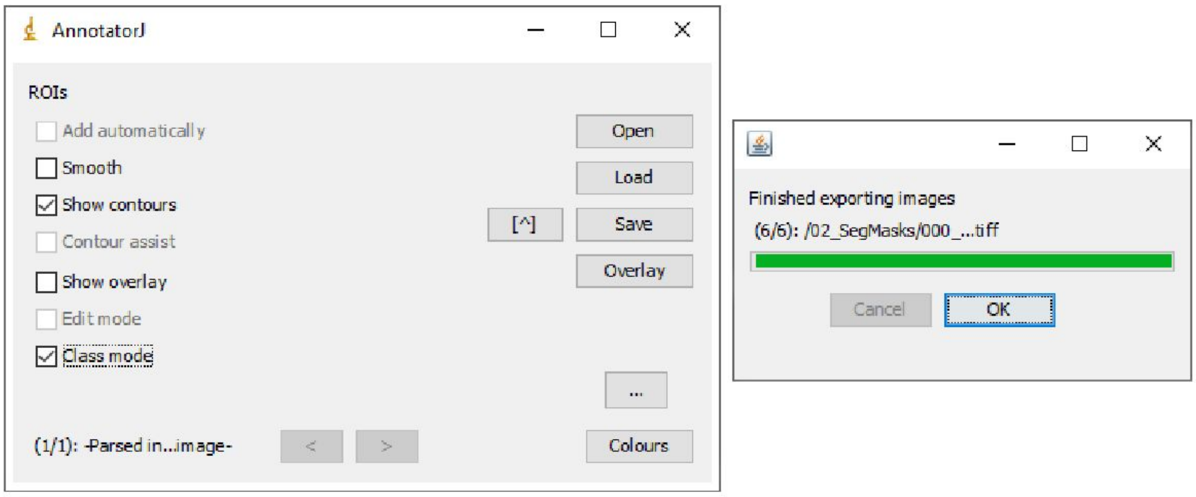
An example config file contains the following lines:

```
#Fri Jun 12 12:03:08 CEST 2020
contourAssistThresholdGray=0.1
contourAssistThresholdG=0.4
contourAssistThresholdB=0.2
autoMaskLoad=yes
classes=normal,cancerous,cyto
semanticBrushSize=25
rememberAnnotType=yes
overlayColor=yellow
contourAssistThresholdR=0.2
saveAnnotTimes=no
saveOutlines=no
enableMaskLoad=yes
annotationColor=green
modelJsonFile=model_real.json
defaultAnnotType=instance
modelFolder=
contourAssistMethod=UNet
contourAssistBrushsize=5
modelFullFile=model_real.hdf5
contourAssistMaxDistance=17
modelWeightsFile=model_real_weights.h5
```

The first line starting with '#' marks a comment that states when the file was last modified. Custom classes can be added to the list 'classes' for convenience. The model used in *Contour assist* mode can be set here (file name and path). Further settings available in '...' (options) or the default annotation colour can also be set. Default annotation type to use can also be set for convenience.

Quick export - "[^]" button

The main AnnotatorJ window has been extended with an export button ("[^]") that exports the currently opened annotations to multi-labelled mask format (see in Export below).
A progress bar window is displayed when exporting annotations in the OpSeF data structure (see later), similar to the that of the exporter plugin.



Finished exporting images
(6/6): /02_SegMasks/000_...tiff

Figure 15. AnnotatorJ main window with the quick export ("[^]") button.

Batch mask load - Load button *(must be configured)*

In the file *AnnotatorJconfig.txt* `enableMaskLoad=yes` and `autoMaskLoad=yes` options must be configured for this functionality. Only instance annotation type is supported. After opening an image the Load button will trigger a folder selection dialog box for the user to select a folder of multi-labelled .tiff mask images (see in Export) with the same name as images found in the currently selected original image folder (also see in Export). If the mask image exists, ROIs are imported for the currently opened image and can be edited. When opening another image in the same original image folder (or stepping with the buttons "<" and ">") the plugin tries to automatically load the mask image with the same name; if the file does not exist a message is printed to the Log window. Upon successful import of the first mask image, the mask folder is saved for the given session of AnnotatorJ. Restart of the plugin is needed to load masks from a different folder.

Export

How to run the exporter plugin, AnnotatorJExporter:

1. Start Fiji
2. Navigate to Plugins → AnnotatorJExporter

How to run AnnotatorJExporter from the pre-built version:

1. Locate *annotator_Project-0.0.1-SNAPSHOT.jar* in your file system
2. Run it
3. Close automatically opened window of AnnotatorJ
4. Navigate to Plugins → AnnotatorJExporter

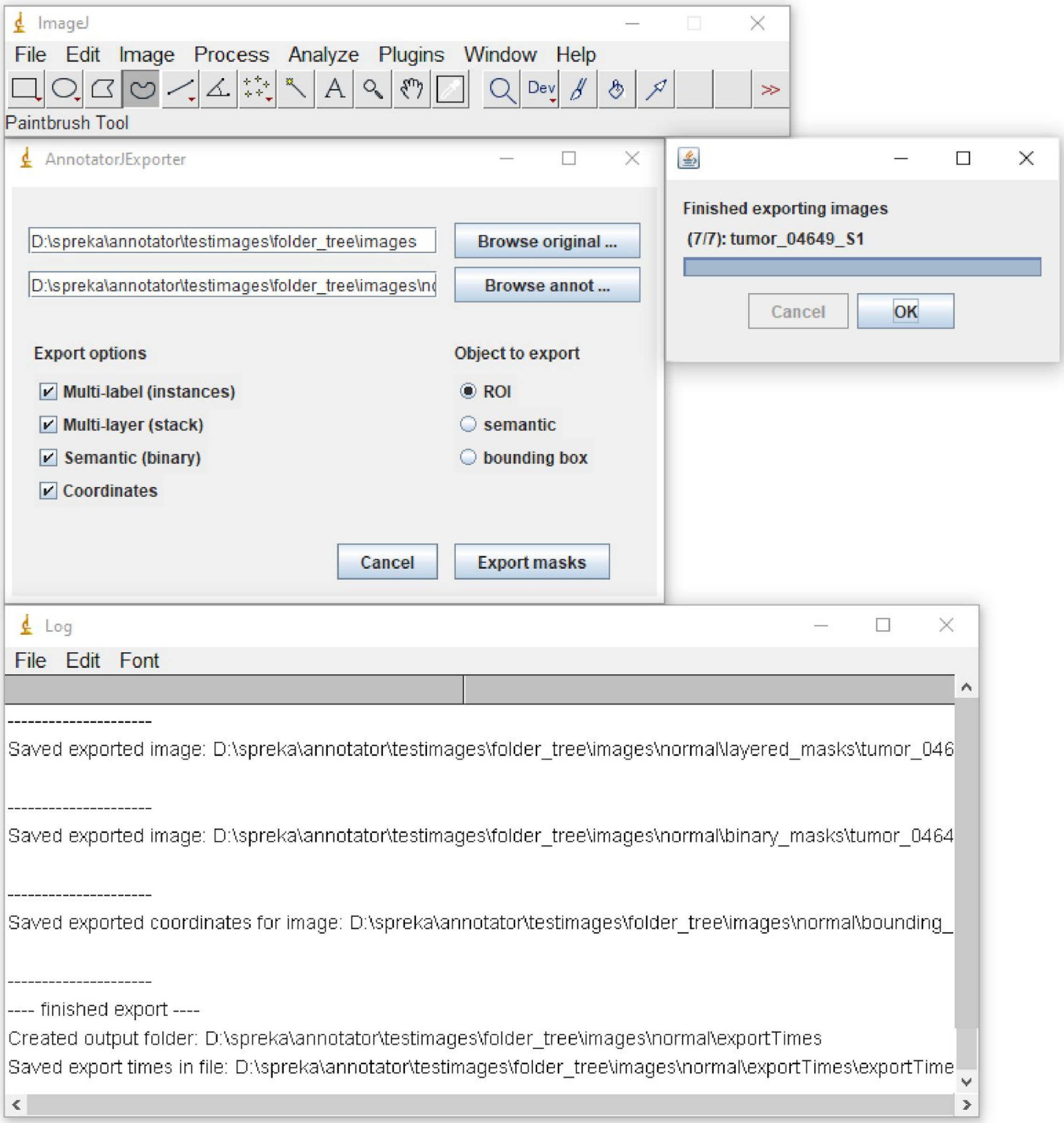
When exporting the annotations, a class folder must be selected and all annotations in that class will be exported in one batch for all images found in the provided original image folder. Only annotations that have corresponding original image files will be exported, correspondence is based on the naming convention used in the annotation plugin.
Selecting an annotation option will determine the type of annotation files the exporter looks for in the annotation folder. For example ROI corresponds to instance annotation, in which case annotation files ending in "_ROIs.zip" or [ROIs]_[number].zip will be listed for export. If an original image has multiple corresponding annotation files, a popup window will ask the user to select which annotation file to use for the given image.
Exported files also follow a naming convention similar to annotation files, and they are saved in separate folders by export types as detailed in **Table 1**.

export option	exported file name	exported folder
multi-labelled masks	[original image name].tiff	labelled_masks
multi-layered masks	[original image name].tiff	layered_masks
semantic masks	[original image name].tiff	binary_masks
coordinates	[original image name].csv	bounding_box_coordinates
overlay	[original image name].tiff	outlined_images

Table 1. Export options and names.

The available export options are visually represented on **Fig.3** and the folder structure created by the exporter follows that of the annotation plugin, as shown on **Fig.7** in green.
Figure 16 below demonstrates the exporter windows. After selecting the folders for original images and annotations in their respective text boxes or browse buttons, the object type can be selected on the right with radio buttons corresponding to the annotation types offered in AnnotatorJ. Multiple export options can be used at once which will create all selected export files for each annotation file. Progress can be tracked in a separate progress bar window or in the Log window of ImageJ that also displays the full path of the saved exported files.

As **Fig.3** suggests, all export options are available for both instance and semantic annotation type, however, when exporting semantic annotations as multi-labelled or multi-layered masks or coordinate files please note that only those objects will be separated to individual objects that do not touch each other on the annotation image.



D:\spreka\annotator\testimages\folder_tree\images

D:\spreka\annotator\testimages\folder_tree\images\n

Export options

- [✓] Multi-label (instances)
- [✓] Multi-layer (stack)
- [✓] Semantic (binary)
- [✓] Coordinates

Object to export

[✓ ROI] semantic bounding box

Finished exporting images
(7/7): tumor_04649_S1

```
---------------------
Saved exported image: D:\spreka\annotator\testimages\folder_tree\images\normal\layered_masks\tumor_046

---------------------
Saved exported image: D:\spreka\annotator\testimages\folder_tree\images\normal\binary_masks\tumor_0464

---------------------
Saved exported coordinates for image: D:\spreka\annotator\testimages\folder_tree\images\normal\bounding_

---------------------
---- finished export ----
Created output folder: D:\spreka\annotator\testimages\folder_tree\images\normal\exportTimes
Saved export times in file: D:\spreka\annotator\testimages\folder_tree\images\normal\exportTimes\exportTime
```

Figure 16. AnnotatorJExporter. *The main window shows options for annotation type to filter the annotation files by and export options that determine the output type. Multiple export options can be selected at once. A progress window appears when starting to export.*

The Overlay export option saves ROIs outlined on the original image (in the annotation colour used to annotate the image).

OpSeF compatibility

How to run the OpSeF importer plugin, AnnotatorJImportOpSef:

1. Start Fiji
2. Navigate to Plugins → *AnnotatorJImportOpSef*

How to run AnnotatorJExporter from the pre-built version:

1. Locate *annotator_Project-0.0.1-SNAPSHOT.jar* in your file system
2. Run it

AnnotatorJ has been extended to support the data structure used in OpSeF[1] by the *AnnotatorJImportOpSef* plugin. In contrast to the usual folder structure of AnnotatorJ, input images and initial segmentation masks are located as in **Table 2**.

folder	usage	comment
00_InputRaw	original image to open	opened as a “fake” z-stack input from OpSeF
01_Input	(OpSeF only)	
02_SegMasks	segmentation masks	multi-labelled, 16-bit masks input from OpSeF
03_SegOverlays	(OpSeF only)	
04_BasicQuantification	(OpSeF only)	
05_AdditionalChannel	(OpSeF only)	
06_AdditionalMask	(OpSeF only)	
07_ClassifiedSegMasks	classified segmentation masks	separate masks for each class optional input from OpSeF
08_ClassifiedSegOverlays	(OpSeF only)	
09_ClassifiedQuantification	(OpSeF only)	
10_ImportExport	input/output file names in .txt	for internal/OpSeF usage
11_SegMasksFromFiji	modified segmentation masks	multi-labelled, 16-bit masks contains all objects of all classes
12_TmpRoisFromFiji	ROI sets of modified contours	like basic AnnotatorJ saved ROIs.zip loaded at startup if files exist
13_ClassifiedSegMasksFromFiji	modified, classified masks	multi-labelled, 16-bit masks separate masks for each class labels remain the same as in 11_SegMasksFromFiji

Table 2. OpSeF folder structure. *Folders marked as (OpSeF only) do not need to exist for the plugin to work correctly.*

Folders *00_InputRaw* and *02_SegMasks* are inputs from OpSeF, *07_ClassifiedSegMasks* is an optional input containing only those segmented objects that correspond to a given class, for

[1] Rasse TM, Hollandi R and Horvath P (2020) OpSeF: Open Source Python Framework for Collaborative Instance Segmentation of Bioimages. Front. Bioeng. Biotechnol. 8:558880. doi: 10.3389/fbioe.2020.558880

each class. Folder *10_ImportExport* stores text files used to import images and masks to AnnotatorJ and marks masks exported from it for later use in OpSeF. Folders *11_SegMasksFromFiji* and *13_ClassifiedSegMasksFromFiji* contain the modified and optionally classified masks created in AnnotatorJ. Current work can be saved to folder *12_TmpRoisFromFiji* (selected automatically) so you can continue working on the project later. At import this folder is searched for "_ROIs.zip" files matching the input mask file names and loaded if exist by default. If you do not want your previously saved ROIs to be loaded at startup, you have to manually delete this folder (or rename either the folder or specific ROI zip files to omit).

When this plugin is started, some functions of AnnotatorJ are modified to help and fit to OpSeF as follows.

Data import

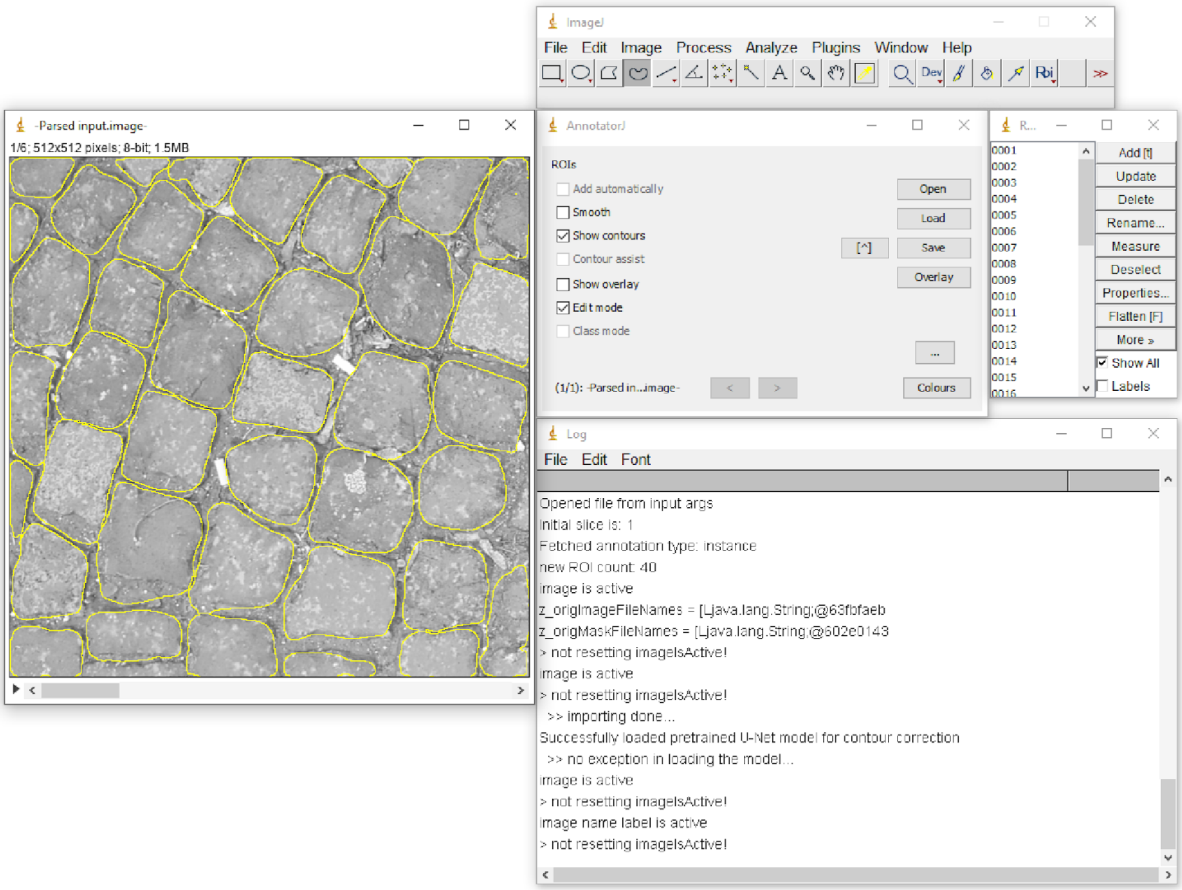
In AnnotatorJOpSef, a *FilePairList* text file must be located in the file open dialog displayed at startup, in the desired OpSeF project folder. It determines which original images and their corresponding masks are opened.
The text file has at least 2 columns and as many lines as image/mask pairs exist in the input folders (*00_InputRaw* and either *02_SegMasks* or *07_ClassifiedSegMasks*).

An example FilePairList .txt file looks as below:

```
/00_InputRaw/8bitSum_Train_20200304_080724.tif;/02_SegMasks/000_CP_Mask_0.4_R
      GBInput_000_Train_20200304_080724.tif
/00_InputRaw/8bitSum_Test_20200304_081408.tif;/02_SegMasks/000_CP_Mask_0.4_RG
      BInput_000_Test_20200304_081408.tif
/00_InputRaw/8bitSum_Test_20200304_080807.tif;/02_SegMasks/000_CP_Mask_0.4_RG
      BInput_000_Test_20200304_080807.tif
/00_InputRaw/8bitSum_Test_20200304_080748.tif;/02_SegMasks/000_CP_Mask_0.4_RG
      BInput_000_Test_20200304_080748.tif
/00_InputRaw/8bitSum_Train_20200304_081404.tif;/02_SegMasks/000_CP_Mask_0.4_R
      GBInput_000_Train_20200304_081404.tif
/00_InputRaw/8bitSum_Test_20200304_080730.tif;/02_SegMasks/000_CP_Mask_0.4_RG
      BInput_000_Test_20200304_080730.tif
```

As opposed to usual initialization with the Open button, all images listed in the text file are opened as a fake z-stack, while their masks are imported as ROI sets (see **Figure 17** below). After import, "slices", i.e. the original images can be stepped through with mouse scrolling. Synchronously, ROI sets corresponding to each image (imported from their mask) are being stepped with the images and displayed accordingly. On a given slice (layer of the stack) annotation functions work as usual.

Edit mode is automatically selected at startup.



```
Opened file from input args
Initial slice is: 1
Fetched annotation type: instance
new ROI count: 40
image is active
z_origImageFileNames = [Ljava.lang.String;@63fbfaeb
z_origMaskFileNames = [Ljava.lang.String;@602e0143
> not resetting imageIsActive!
image is active
> not resetting imageIsActive!
  >> importing done...
Successfully loaded pretrained U-Net model for contour correction
  >> no exception in loading the model...
image is active
> not resetting imageIsActive!
image name label is active
> not resetting imageIsActive!
```

Figure 17. Imported OpSeF-structured images and ROIs. Images are collected in a z-stack that can be stepped with mouse scroll. The corresponding ROI set is also stepped and displayed.

Saving annotations (OpSeF structure) - Save button

When images (and masks) are imported in AnnotatorJOpSef the Save button saves “_ROIs.zip” files to the *12_TmpRoisFromFiji* folder in the OpSeF folder structure. The usual class selection window will not open, and already existing files in this folder will be overwritten without a renaming option - because this is a temporary folder.
This function is also executed when clicking on the “[^]” (Export) button (see **Fig.15**) if input data was imported from an OpSeF project.