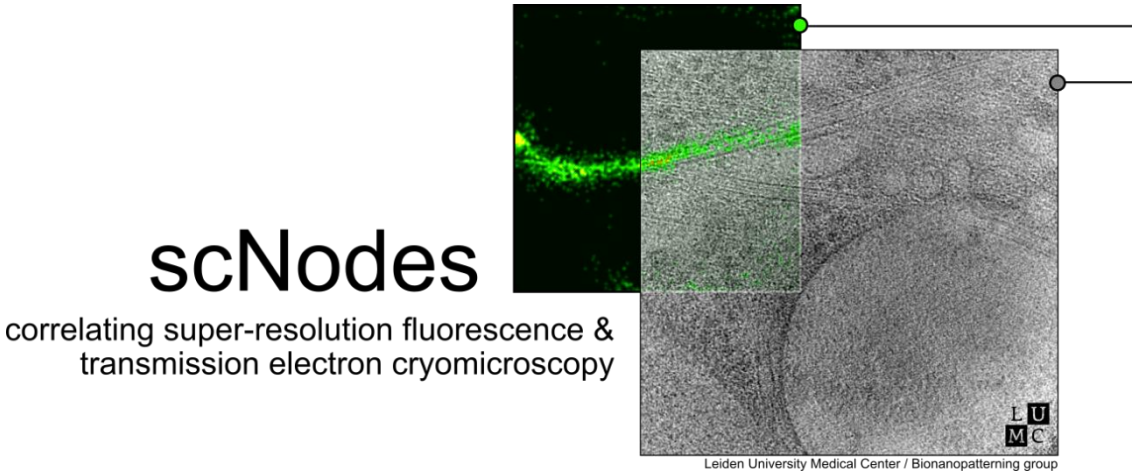
scNodes user manual

Version 1.0.64, January 22nd 2023. m.g.f.last@lumc.nl



Contents

About scNodes

Introduction

scNodes is **a software suite for correlated super-resolution fluorescence and transmission electron cryomicroscopy.** We designed the programme for use in our own 'super-res-cryo-CLEM' workflow – but scNodes as a whole or parts of the software may also find use outside of this field.

scNodes consists of two main parts: the Node Editor, for fluorescence microscopy image processing, and the Correlation Editor, for correlation of images. This manual consists of two corresponding parts: an introduction to the Node Editor, in which we demonstrate **how to create super-resolution reconstructions with scNodes**, and an introduction to the Correlation Editor that outlines a **method to correlate super-resolution data with cryo-electron tomograms.**

While the first public release of scNodes comprises a fully functional programme, there are numerous features, processing modules, or correlation methods that one could think of to improve the software with. For this reason, scNodes was designed to be **extensible**. This manual includes a tutorial on **creating custom nodes for the Node Editor,** as well as on how to create **plugins for the Correlation Editor.** Rather than by creating your own features, **installing extra features** is another way of extending the core functionality of scNodes. The GitHub repository for this project (github.com/bionanopatterning/scnodes) contains a **discussions** section where, it is our hope, you can share or find such plugins.

This discussion board is also a place to ask for help. We hope that others will also find our software of use.

Installation

We have tried to make the installation of scNodes as easy as possible. However, installing software – python in particular – can be a pain. Three ways of installing scNodes are listed below. The first method is the easiest: simply download the release from GitHub, unzip the folder, and launch the scNodes executable. However, as an executable, this installation lacks the versatility of importing modules that the alternative installations with either pip or PyCharm offer.

scNodes was developed on and tested for Windows. A CUDA-compatible GPU is required for the PSF-fitting node; installation can work without such a card, but the PSF-fitting node will not.

Option 1: installation of the scNodes executable

Go to github.com/bionanopatterning/scNodes and download the most recent release, which is listed in the 'Releases' header on the right of the page. Unzip the downloaded folder and move it to a location where you want it to be located, e.g. C:/Program Files/scNodes. Use the launcher called 'scNodes.exe' to start scNodes.

Option 2: installation with pip

Open any terminal window from within which python and pip can be called; e.g. Anaconda Prompt. Open it as an administrator (right-click, 'Run as administrator)'. Run the following command:

```
pip install scNodes
```

With some luck this should install the full package. If succesful, set up scNodes by running the command below. If unsuccesful, read on a bit.

```
python -m scNodes install
```

This will install dependencies that can not reliably be downloaded and installed via PyPi (pyGpuFit, pyimgui-wheels and fill).

If succesful, run scNodes using the following command:

```
python -m scNodes
```

If either the installation or the setup fail, this is typically due to a failure to automatically install some dependency. In such a case an error message will appear, indicating which dependency could not be found/installed. For example:

```
ERROR: Could not find a version that satisfies the requirement glfw>=2.5.5 (from scnodes) (from versions: none)          ERROR: No matching distribution found for glfw>=2.5.5
```

In such a case, manually installing the dependency with pip sometimes solves the issue. In this example:

```
pip install glfw
```

Then try installing scNodes again:

```
pip install scNodes
```

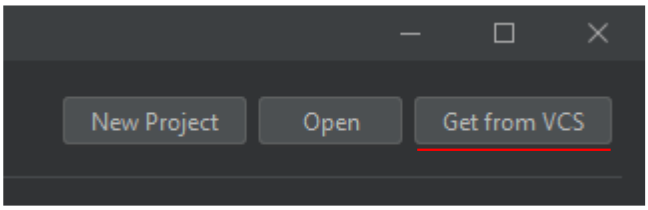
In order to avoid having to manually install multiple dependencies, you can also run the following command to install everything at once:

```
pip install colorcet dill glfw joblib matplotlib mrcfile numpy opencv-python pandas Pillow psutil PyOpenGl PyWavelets pyperclip pystackreg scikit-image tifffile
```

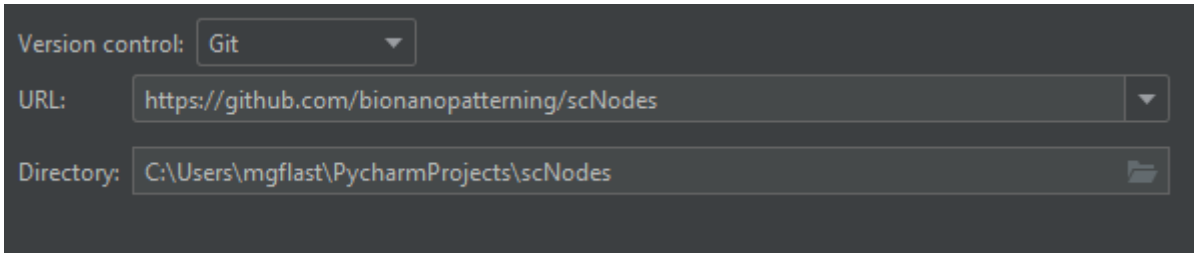
Option 3: clone the project from GitHub

A second way of installing scNodes is to clone the project from GitHub into an editor of your choice. We recommend PyCharm – get it at:
https://www.jetbrains.com/pycharm/download/#section=windows

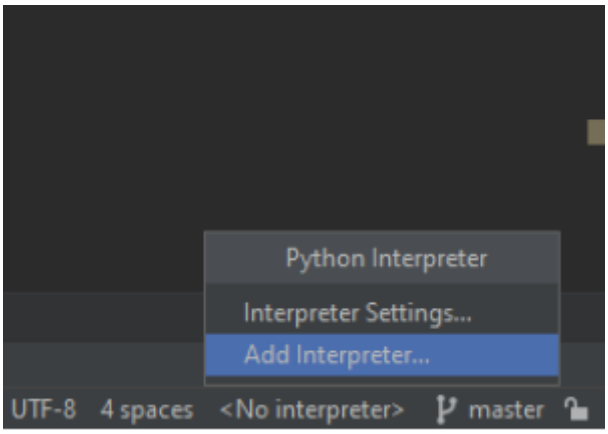
Install and start PyCharm. In the start-up window, select the 'Get from VCS' button in the top-right corner:



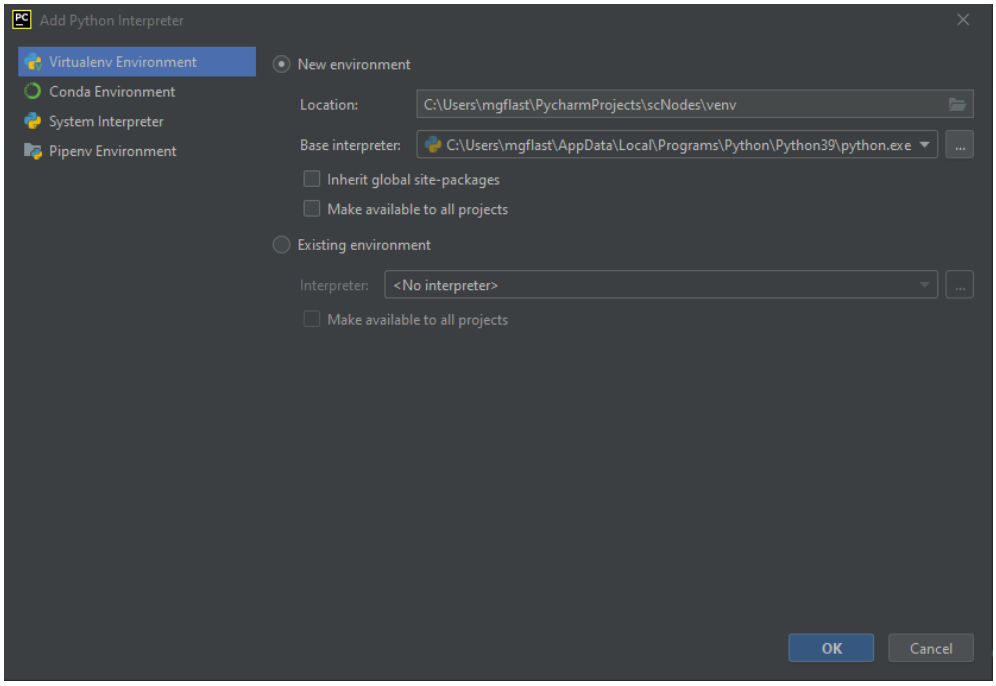
In the subsequent window, enter the GitHub repository url for scNodes: https://github.com/bionanopatterning/scNodes as the URL, and choose a directory to start a new PycharmProject in (in the example below: C:\Users\...)



Pycharm will now get scNodes from GitHub and open the newly created project. Before we can run the programme, a Python interpreter must be set up. In the very bottom right of the Pycharm window, click the '<No interpreter>' icon and select 'Add Interpreter...'.

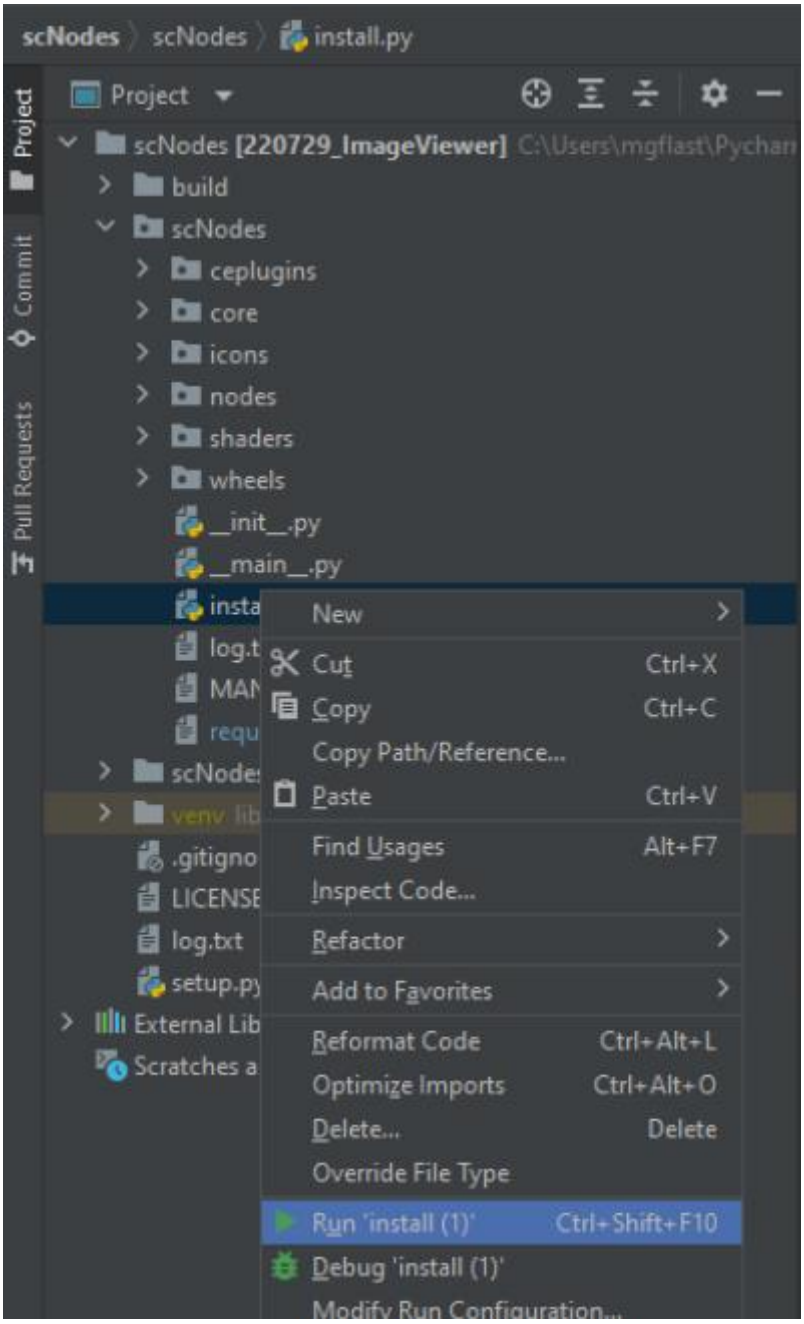


The following menu will open up. The default settings are fine – press OK to continue.



After a couple of seconds, a warning pops up to tell you that various packages are missing. Click 'Install requirements' to automatically download and install these requirements.

Downloading and installing all these packages may take a couple of minutes. Once it is complete, find the 'install.py' file in the project browser (see image below). Right click install.py and select 'Run install', as in the image below.



Extra dependencies that don't automatically install are now installed. Finally, run __main__.py (right click -> Run __main__) to start the software.

Should you want to package your own executable version of scNodes, install 'pyinstaller' (pip install pyinstaller), and in the Pycharm terminal navigate to the folder in which the file 'scNodes.spec' is located. Run:

```
pyinstaller scNodes.spec
```

To package the project. In some cases, modules that are imported by modules that are themselves dynamically imported by the software (e.g. node and ceplugin

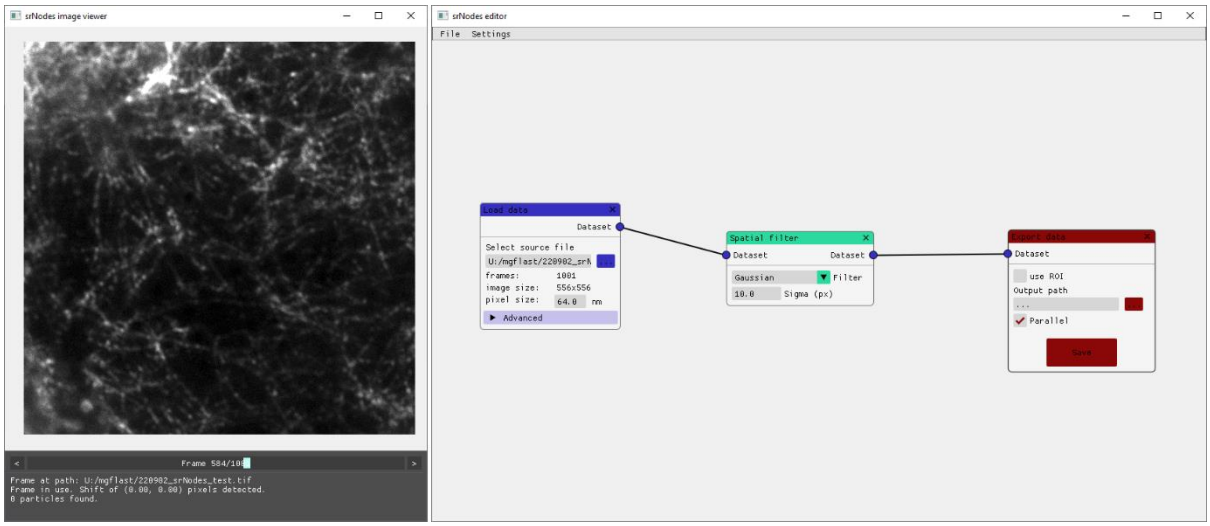
implementations) are not packaged properly. Edit the scNodes.spec file to include these required modules in order to ensure that pyinstaller incorporates them during the build.

User interface - overview

The scNodes user interface comprises two windows: the *image viewer* and the *editor*. In the editor, the user can access either Node Editor or the Correlation Editor.

The Node Editor is used to set up processing pipelines by connecting *nodes*; image processing blocks that take some input, apply a specific operation to that input, and output the result. The image viewer always shows the output of the currently active node (i.e. the node that was last interacted with). The secondary window, the image viewer, continuously displays the result of whatever processing pipeline is set up in the Node Editor.

The Correlation Editor is a tool that enables overlaying, registration, and blending of light and electron microscopy images. Images generated using the Node Editor can be ported into the Correlation Editor with a single click, and external images (.mrc, .tif, and .png) can be imported by simply dropping files onto the editor. Correlated images can be exported as either a .png or a .tif file, at publication-level resolution.



Overview of the Node Editor user interface. Left: the image viewer, right: the editor window showing the Node Editor.

The Node Editor

User interface – context menus

In both windows (i.e. the editor and the image viewer), right clicking anywhere outside of a UI element will open the context menu. In the image viewer, the context menu provides options to change the contrast settings (shortcut ctrl + shift + C), reset the view, and to change the look-up-table (LUT) that is applied to the image data. By default, the option 'always auto' is enabled in the image viewer. When enabled, the contrast settings in the image viewer are automatically set for every new image that is displayed (set such that 3% of the pixels in the image are saturated). The default look-up-table is grayscale.

In the editor, the context menu is the menu via which new nodes are added to the setup.

Right-clicking on a node will open that node's context menu, which provides options to reset, delete, or duplicate the node, as well as to *focus* the node (see below).

User interface – key input
In the image viewer, press 'ctrl + shift + c' to open the contrast settings window, or 'ctrl + s' to save the currently displayed image.
In the editor, holding escape will pause any ongoing processes for as long as the button remains pressed.

Focusing nodes
Focusing a node via the node's right-click context menu fixes that node as the active node. This means that as long as a node is focused, the image viewer will show that node's output – also when the user interacts with other nodes. Focusing a node can be a useful way to inspect how the output of a node changes when the settings of upstream nodes are changed. A focused node is highlighted with a coloured background. Releasing focus of a node is done via either the focused node's context menu, or the editor context menu.

Overview of default nodes
The section below provides an overview of the nodes that are available in the default version of scNodes.

Data IO

- Load dataset: load .tif files from either a single multi-page stack or a folder containing multiple images.
- Load image: load a single image
- Load reconstruction: import a .csv file with particle-based super-resolution reconstruction data (of the type exported by e.g. ThunderStorm).
- Export data: exports full datasets, single images, or reconstructions.

Image processing

- Registration: drift correction / image registration. Two methods are available: full-image grayscale based registration with TurboReg, or feature-detection and alignment based registration using ORB.
- Spatial filter: applies image filters (e.g. gaussian, median, wavelet) to input frames.
- Temporal filter: performs temporal derivatives, grouped difference, and windowed averaging of consecutive frames in a dataset. Note that grouped difference in particular can be useful in photo-activated localization datasets where image background is high: this filter divides the dataset into groups (e.g. groups of 20 frames) and subtracts the same frame (e.g. the 20th, the last frame prior to the next photo-activation pulse) from all images in the group.
- Image calculator: takes two images/datasets as input and performs addition, multiplication, etc.
- Frame shift: shift the dataset frame index by an arbitrary value. E.g., when receiving frame 100 as the input, returns frame 98. Can be useful for building custom temporal filters.

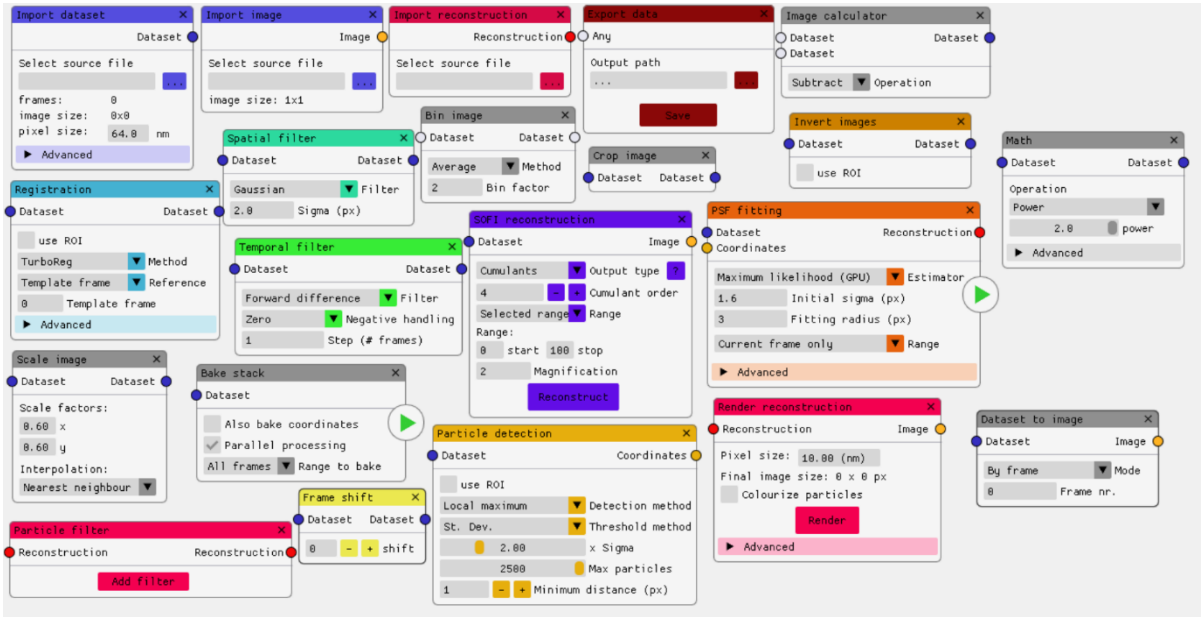
- Bin image: bin images by an integer factor.
- Crop image: crop images to a user-specified region of interest. Can be useful to speed up downstream processing. Note: when the crop image node is active, the image viewer does not show the output of the crop image node, but the input plus the user specified ROI.
- Bake stack: the bake stack node is crucial in optimizing the processing speed of a setup. Once a useful processing pipeline has been found, the final node in that pipeline can be routed in to a bake stack node, which when activated will compute the full output stack in parallel spread over as many CPUs as are available on the PC (or less – the number is set in the editor's settings menu.)

Reconstruction

- Particle detection: detects 'particles': coordinates of significant peaks in the input image. The output is a list of coordinates per frame, which is input for the PSF fitting step.
- PSF fitting: takes a dataset and a coordinate set as the input, and performs per-frame PSF fitting on the GPU. The advanced menu allows the user to set custom bounds for the parameters of the fit (e.g. min/max intensity, sigma, and offset). The node outputs data of type *reconstruction*.
- Render reconstruction: renders the reconstruction. Result is displayed in the image viewer, and can be routed into an export data node to save it to disk.
- Particle filter: filter a reconstruction on the basis of particle parameters, such as the uncertainty, background standard deviation, etc. Multiple filters can be set up in the same node.

Converters:

- Dataset to image: either grabs a single frame from the input dataset and outputs that always, or generates time-projections, such as the sum, standard deviation, or maximum, of the entire dataset.
- Math: perform mathematical operations on input datasets/images, such as threshold, log, power, etc.



2 - Overview of default nodes

Tutorial: registration & reconstruction

This tutorial will demonstrate how to generate a super-resolution reconstruction of a typical STORM dataset acquired by fluorescence cryomicroscopy. These datasets generally suffer from high drift, notable background fluorescence, and relatively low signal-to-noise due to the use of non-immersion objectives. The end result (the node setup) of this tutorial is also available in the default scNodes package – it can be imported by going to *File -> Load node setup* in the top menu bar, and selecting the file 'scNodes_tutorial_setup.srn'.

Step 1: adding a *Load data* node, and importing the dataset. Open the context menu in the editor (right-click) and add a *Load data* node from the submenu *Data IO.* Below the field 'select source file,' either paste the path to the data of interest, or press the adjacent button to open a file browser and select the file. The stack can be browsed using the slider in the image viewer info menu.

Step 2: drift correction using a *Registration* node. Add the node (it is found in the *image processing* sub-menu of the editor's context menu) and connect its input to the *Load data* node's dataset output. To decrease the computational cost of the registration, the option to use a specific region of interest (ROI), rather than the entire image is available. To use it, check 'use ROI'. The selected ROI is highlighted in the image viewer and can be edited by either moving it around (click within the ROI + drag) or drawing a new ROI (click outside of the ROI + drag).

Step 3: image filtering for particle detection. When satisfied with the result of registration, add a *Spatial filter* node and connect it to the *Registration* node. In this tutorial we'll use a difference of gaussians filter for maximum detection. Set the spatial filter to Gaussian and the sigma to 5.0 pixels. Add a second spatial filter (or duplicate the first, via the node's right-click context menu), set the filter to Gaussian and the sigma to 1.0 pixels. Next, add an *Image calculator* node, set the operation to *Subtract,* and connect the outputs of the two spatial

filter nodes to the inputs of the image calculator node. The image calculator performs the following operation:

$$output = input_a \ \Omega \ input_b$$

In which input a is the top and input b the bottom connector, and Ω the selected operator. Alternatively, use a spatial filter node and apply a 'difference of Gaussians' to achieve the same result.

Step 4: particle detection. Add a *Particle detection* node from the *Reconstruction* submenu and connect it to the output of the image calculator. Using a ROI in the particle detection step can help limit the total amount of particles found, and thus speed up processing. Set the threshold method to *St. Dev.* and the factor to 1.0 – meaning: the intensity threshold for every frame will be 1.0 times the standard deviation of the pixel intensity in that frame. Note: when using a ROI, the threshold will be based on the ROI only. Adjust the maximum number of particles and the minimum distance between particles if necessary. Inspect the image viewer to optimize the settings.

Step 5: baking the stack. The next step in the PSF fitting super-resolution reconstruction pipeline is performing the PSF fitting for all of the detected particles. In scNodes, this process is optimized by performing it on the GPU on a frame-by-frame basis. The processing of frames prior to the GPU-based computation, however, can be optimized by using parallel computation on multiple CPUs. Thus, it is practical to separate the preparation and the PSF fitting steps, by generating, or 'baking', a final stack, for input to the PSF fitting node. We use a *Bake stack* node to perform this CPU-based parallel computation.

Add a bake stack node and connect it to the output of the particle detection node as well as to the output of the registration node. Note that it is possible to use any output of type *dataset*, but for now we simply want to perform the PSF fitting on the original data. Once set up, press the start button on the bake stack node to initiate processing.

Note: it is not required to use a bake stack node. A PSF fitting node (see below) can also be directly connected to any of the previous dataset-outputting node and the particle detection node.

Step 6: PSF fitting. When the stack is fully baked, add a *PSF fitting* node and connect it to the output of the bake stack node. Set the estimator to 'Maximum likelihood,' the initial sigma to 1.6 pixels, and the fitting radius to 3 pixels, and the range to 'All frames,' then press the start button. Alternatively, use a 'Custom range' for faster feedback on the quality of the final result, if you are still exploring the processing settings.

Step 7: rendering the final result. Once the PSF fitting node is finished processing, add a *Render reconstruction* node and connect it to the PSF fitting node. Press the 'Render' button to render the final result, and inspect it in the image viewer.

Step 8: exporting the results. Add two *Export data* nodes. Connect one to the output of the Render reconstruction node and the other to the output of the PSF fitting node. Change the output path to the desired location and press 'Save' to save the file (an image in the one case, and a .csv file with particle data in the other) to the disk. Optionally, add a third export data node (or re-use another) and connect it to the dataset output of the bake stack node, to save the final stack that was used for PSF fitting.

Step 9 - optional: particle filtering and colouring. Remove the link between the reconstruction renderer and the PSF fitting node (right-click on the connector blob of either node). Add a *Particle filter* node. Connect the output of the PSF fitting node to the particle filter and connect the output of the particle filter to the render reconstruction node. In the reconstruction renderer, activate the 'Colourize particles' checkbox and select a parameter to colourize the particles by. For an intuitive example of this rendering method, choose 'frame' as the parameter and set the LUT to 'Over/under B'. If there is any remaining drift in the dataset that the reconstruction was made with, this rendering method will make the drift very apparent.

In the particle filter node, click the '+' button to add a new filter. Select a parameter of interest, e.g. 'offset', and drag the 'min' and 'max' sliders to select a range of allowed offset values. Any particle with an offset value outside of this range will not be incorporated in the final reconstruction. Connect the output of the particle filter to the input of the reconstruction renderer. Press 'Render' again to see how the particle filter and particle painter nodes affect the result.

Step 10 – optional: SOFI reconstruction. For datasets where the background fluorescence intensity is relatively high, or the labelling density is high, PSF-fitting approaches to super-resolution image reconstruction can be impractical. One alternative to PSF-fitting reconstruction is stochastical optical fluctuation imaging. A node for SOFI reconstruction is found under the header 'alternative reconstructions' and can be used for an alternative reconstruction method.

Downloading and installing new nodes

Tutorial: downloading and installing a node

scNodes was designed to be easily extensible. Installing extra nodes is only a matter of downloading the right file, and either: i) copying the file into the 'nodes' folder, found in the scNodes programme directory, or ii) installing it from within the Node Editor. For this brief tutorial, a 'Fourier transform' node is available here, on the scNodes GitHub Discussions page: https://github.com/bionanopatterning/scNodes/discussions/3

Download the 'fftnode.zip' file provided via the above link and extract the .zip folder to find a file called 'fftnode.py'. If you can't locate the folder where scNodes was installed, simply start scNodes and open the 'Settings' menu in the editor's top menu bar. Select the option 'Install a node', select the file 'fftnode.py', and click Open. The node is now installed – the software must be re-started for it to become available.

Upon starting scNodes, an error will be displayed in case a '.py' file is found in the nodes folder that does not define a proper node (see also the tutorial below).

Creating a custom node

Tutorial: creating a custom node that inverts images

The directory *'nodes'* found in the project folder, contains the source code for all of the default nodes, as well as a file called '*custom_node_template.py*.' All of the .py files in the *nodes* folder that properly define a node will be loaded upon starting the software, so that these nodes are available in the editor (an exception is the aforementioned template file which is explicitly ignored).

Step 1: open the file 'custom_node_template.py' and read through it. For a custom node to be registered by the software, the .py file that defines it must minimally contain the following two definitions:

- A function called 'create' that takes no arguments and returns an object of your custom node type
- A class with any name, e.g. InvertNode, that inherits from the Node base class and calls its __init__ function upon initialization.

Thus, the minimal code to define a node that is available in the editor would be as follows:

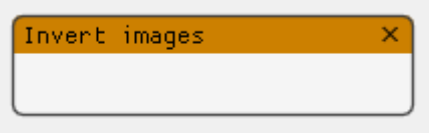
```
from node import *


def create():
    return InvertNode()


class InvertNode(Node):
    def __init__(self):
        super().__init__()
```

Step 2: create a new file in the /nodes folder, give it a name, and copy the above code into it. So far, the node would look as follows:

The while colour and title 'NullNode' are defaults and the group in which the node is listed in the editor's context menu is called 'Ungrouped'. These parameters can be changed by editing the variables *title, colour,* and *group,* as follows:

```
class InvertNode(Node):
    title = "Invert images"
    colour = (0.8, 0.5, 0.0, 1.0)
    group = "Tutorial"
```

Which changes the look to this:



Step 3: define input and output attributes. Since we want this node to invert the contrast of incoming frames, we need an input as well as an output dataset attribute. These can be set up by adding the following code to the node's __init__ method:

```
def __init__(self):
    super().__init__()

    self.connectable_attributes["dataset_in"] =
ConnectableAttribute(ConnectableAttribute.TYPE_DATASET,
ConnectableAttribute.INPUT, parent=self)
    self.connectable_attributes["dataset_out"] =
ConnectableAttribute(ConnectableAttribute.TYPE_DATASET,
ConnectableAttribute.OUTPUT, parent=self)
```

ConnectableAttribute (attribute, for short) is a class that defines the behaviour of how nodes are connected. Connections are made between two attributes – and thus only implicitly between nodes – of the same attribute type, and only for a pair where one of the attributes' direction is input and that of the other output. These types are defined in the ConnectableAttribute class definition (see node.py). ConnectableAttributes are defined within a Node's __init__ method, and are parented to that node. The connection logic is internally handled, and after defining a node's attributes, the only other requirement is that the attributes are rendered – see next section.

Note that the ConnectableAttribute members are stored in a *dict* called *self.connectable_attributes.* This is done to ensure that all ConnectableAttributes are accessed using the same default named dictionary, which facilitates rendering and saving node setups for re-use. There is no need to handle rendering or saving yourself, as long as any ConnectableAttribute is stored in this dict.

Step 4: overriding the base Node class' *render* methods in order to define a custom GUI for the new node. The Node base class has a number of methods that are automatically called by the editor. Examples are render(), on_update(), and importantly .get_image_impl() – for a full list and details, see the next section. For now it is important to know that the user interface is defined in the render() method, and that the function get_image_impl() is responsible for outputting images. (Note: some nodes do not output images; these nodes will implement methods such as get_particledata_impl() and get_colour_impl() instead).

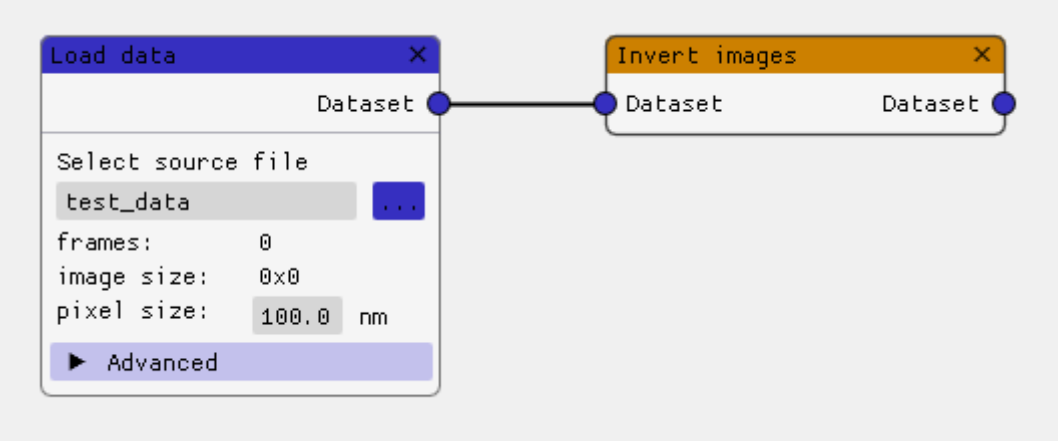
First, the render() method. The default pattern for a custom node's render method is as follows:

```
def render(self):
    if super().render_start():
        # custom render code here
        super().render_end()
```

As the goal of this tutorial is to make a node that mirrors the incoming images, we don't need any settings (the GUI which would typically be defined in the render method). What we do need to do is tell the node to display (render) its ConnectableAttributes:

```
def render(self):
    if super().render_start():
        self.connectable_attributes["dataset_in"].render_start()
        self.connectable_attributes["dataset_out"].render_start()
        self.connectable_attributes["dataset_in"].render_end()
        self.connectable_attributes["dataset_out"].render_end()

        super().render_end()
```

We can now set up the following processing pipeline:



Step 5: overriding the *get_image_impl* method. Another method of the Node base class is called *get_image.* Externally, image output can be requested from nodes by calling their get_image method. Internally though, every node that outputs images must define its own get_image implementation, in a function called *get_image_impl*. Take a look at the base Node class' get_image method to see the difference between this implementation and the base function (the reason for this structure is: get_image wraps get_image_impl in try/catch clauses in order to inform the user on possible errors)

```
def get_image_impl(self, idx=None):
    incoming_node =
self.connectable_attributes["dataset_in"].get_incoming_node()

    if incoming_node:
        incoming_frame = incoming_node.get_image(idx)
        data = incoming_frame.load()
        min_val = np.amin(data)
        max_val = np.amax(data)

        inverted_data = max_val + min_val - data
        output_frame = incoming_frame.clone()
        output_frame.data = data
        return output_frame
```

In the above code, we do the following:

- Find which node is incoming, by calling the *get_incoming_node* method on the node's dataset_in attribute. If this attribute has a connection, the method returns the parent node of the attribute that dataset_in is connected to. In the above image example that would be the Load data node.
- If an incoming node was found, process the image. If not, there is no definition to go to – the method will return None by default, which tells the image viewer not to display any image.
- Get the incoming frame by calling get_image() on the incoming node.
- Get the pixel data for that frame by calling load() on the incoming frame.
- Calculate the pixel data for the inverted-contrast frame.
- Clone the incoming frame. By calling clone() on a frame object, a copy of the object is returned. This way all of the metadata of the original frame is retained.
- Set the data of the cloned frame as the inverted-contrast version of the original frame.
- Finally, return the frame.

For more details on the methods and variables of the Frame and ConnectableAttribute classes, see the next section.

Step 6 - optional: with the above code, the invert node is complete. This next section will demonstrate how to customize the render() method further, in order to add settings to the node. For a more detailed example, see the file 'custom_node_template.py'. Copying the contents of this file to a new .py file in the /nodes folder will make the node available in the editor. Should you want to disable a node, the variable 'enabled' can be set to False (see the custom_node_template.py file).

Most of the GUI in scNodes is built using *imgui*: https://github.com/ocornut/imgui – a simple and user friendly library that makes it very easy to set up graphical user interfaces. Specifically, scNodes uses pyimgui: https://pyimgui.readthedocs.io/en/latest/ - creating nodes will thus require some knowledge of pyimgui. For this we refer to the documentation (see previous link), which is very straightforward.

Here we will add an option to the node to use region of interest so that the contrast will only be inverted in part of the image. The base Node class defines a flag *use_roi* which, when True, tells the image viewer to allow user input for drawing a roi. Let's make the default option to not use a ROI, but add a checkbox to the node GUI to allow the user to decide whether to use a ROI or not.

Add the following line to the __init__ code block:

```
self.use_roi = False
```

Variables .use_roi and .roi are default member variables of the Node base class. The image viewer will automatically allow drawing a ROI when .use_roi is set to True, and store the ROI

coordinates in the .roi var – see the final .render() function in this tutorial for a usage example.

And change the render function to:

```
def render(self):
    if super().render_start():
        self.connectable_attributes["dataset_in"].render_start()
        self.connectable_attributes["dataset_out"].render_start()
        self.connectable_attributes["dataset_in"].render_end()
        self.connectable_attributes["dataset_out"].render_end()

        imgui.spacing()
        imgui.separator()
        imgui.spacing()

        _changed, self.use_roi = imgui.checkbox("use ROI", self.use_roi)

        super().render_end()
```

The resulting node will look as follows when the use ROI checkbox is checked:

When the user clicks the checkbox, the value of the variable use_roi will change accordingly. Next, we need to alter get_image_impl to take in to account the fact that the user may optionally have requested to use a ROI. Frame objects have a method that makes this easy: Frame.load_roi. To write the inverted pixel data back in to the right section of the image, we have to manually index the original pixel data.

```
def get_image_impl(self, idx=None):
    incoming_node =
self.connectable_attributes["dataset_in"].get_incoming_node()
    if incoming_node:
        incoming_frame = incoming_node.get_image(idx)
        data = incoming_frame.load()
        _roi = self.roi
        if not self.use_roi:
            _roi = [0, 0, incoming_frame.width, incoming_frame.height]
        roi_data = incoming_frame.load_roi(_roi)
        min_val = np.amin(roi_data)
        max_val = np.amax(roi_data)

        inverted_data = max_val + min_val - roi_data
        output_frame = incoming_frame.clone()
        output_frame.data = data
        output_frame.data[_roi[1]:_roi[3], _roi[0]:_roi[2]] = inverted_data
        return output_frame
```

The resulting node and its output will look as follows:

When testing this node, you may notice when you i) use the Invert images node to invert the contrast of a ROI and then ii) deselect the 'use roi' checkbox, that the image viewer still shows the previous image. To change this behaviour, set the Node base class' *any_change* flag to True. This will trigger the image viewer to request an update image from the node. Most imgui functions return a tuple of two values: (changed, state) (see for example the line: _changed, self.use_roi = imgui.checkbox(...) above). The first element in this tuple, '*changed*', is a bool that indicates whether any change occurred to that UI element.

```
        _changed, self.use_roi = imgui.checkbox("use ROI", self.use_roi)
        self.any_change = self.any_change or _changed
```

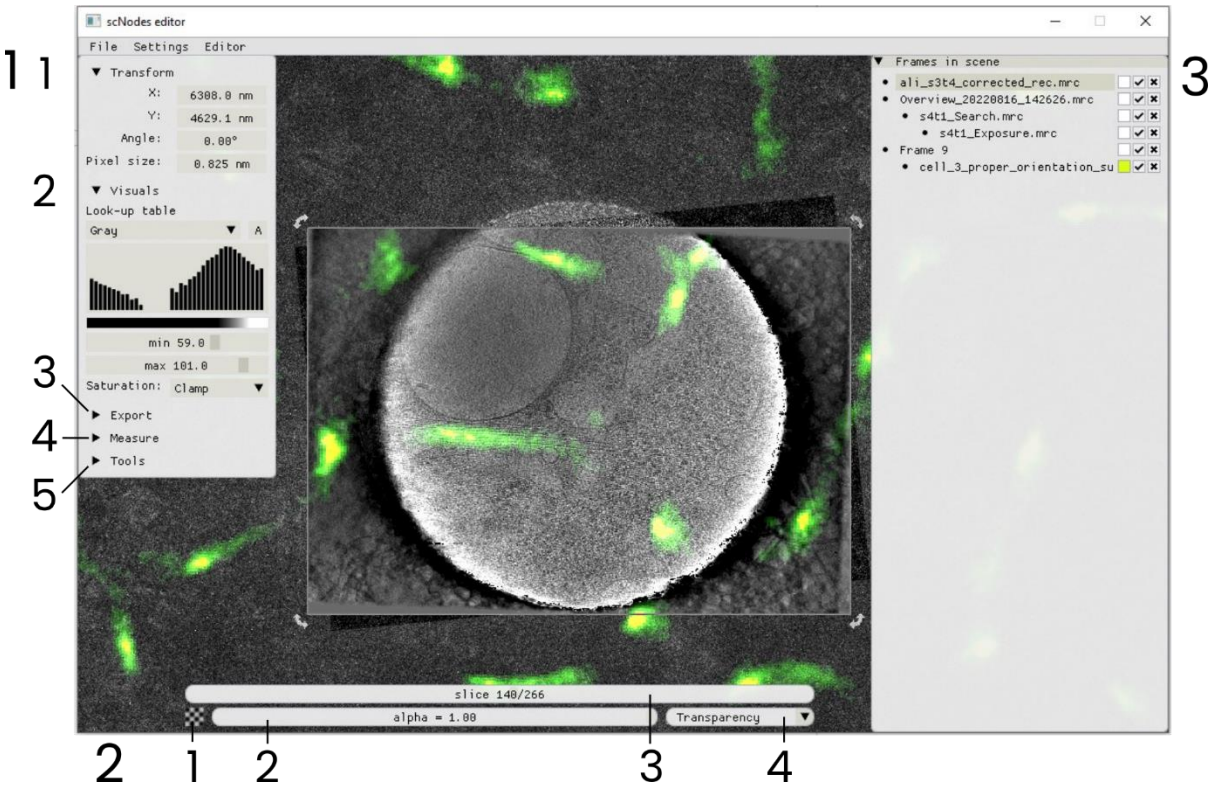
With this line added, the image viewer will immediately show an updated output when the use_roi checkbox is (de)selected.

The final code for the invert image node is pasted below.

```
from scNodes.core.node import *


def create():
    return InvertNode()


class InvertNode(Node):
    title = "Invert images"
    colour = (0.8, 0.5, 0.0, 1.0)
    group = "Image processing"
    sortid = 110

    def __init__(self):
        super().__init__()

        self.connectable_attributes["dataset_in"] =
ConnectableAttribute(ConnectableAttribute.TYPE_DATASET,
ConnectableAttribute.INPUT, parent=self)
        self.connectable_attributes["dataset_out"] =
ConnectableAttribute(ConnectableAttribute.TYPE_DATASET,
ConnectableAttribute.OUTPUT, parent=self)

        self.use_roi = False
```

```
    def render(self):
        if super().render_start():
            self.connectable_attributes["dataset_in"].render_start()
            self.connectable_attributes["dataset_out"].render_start()
            self.connectable_attributes["dataset_in"].render_end()
            self.connectable_attributes["dataset_out"].render_end()

            imgui.spacing()
            imgui.separator()
            imgui.spacing()

            _changed, self.use_roi = imgui.checkbox("use ROI",
self.use_roi)
            self.any_change = self.any_change or _changed

            super().render_end()

    def get_image_impl(self, idx=None):
        incoming_node =
self.connectable_attributes["dataset_in"].get_incoming_node()
        if incoming_node:
            incoming_frame = incoming_node.get_image(idx)
            data = incoming_frame.load()
            _roi = self.roi
            if not self.use_roi:
                _roi = [0, 0, incoming_frame.width, incoming_frame.height]
            roi_data = incoming_frame.load_roi(_roi)
            min_val = np.amin(roi_data)
            max_val = np.amax(roi_data)

            inverted_data = max_val + min_val - roi_data
            output_frame = incoming_frame.clone()
            output_frame.data = data
            output_frame.data[_roi[1]:_roi[3], _roi[0]:_roi[2]] =
inverted_data
            return output_frame
```

The Correlation Editor



Overview of the Correlation Editor interface. For information on which GUI elements do what, see the section below. In this example a cryo-TEM tomogram is overlayed with super-resolution fluorescence data.

The Correlation Editor is used to align, overlay, and blend images. Images can be added to the Correlation Editor by either i) dragging .mrc, .tif, or .png files onto the window, or ii) right-clicking an image in the image viewer window and selecting 'Add to Correlation Editor'. The editor's behaviour is similar to that of common image editing software packages such as Inkscape or Adobe Illustrator. Clicking and dragging a frame moves it in the scene, and clicking and dragging the handles visible at the corner of a selected image will scale or rotate that image. The easiest way to get a feel for it is to just play around with some data.

The numbers in the above overview indicate the following GUI items:

Menu 1 – the tools window

1.1: Transform: displays and allow editing of the selected image's transform; i.e. its position, rotation angle, and pixel size. Dragging left or right on the value fields changes the value. Use control + click to manually enter a value. Upon importing images from the Image Viewer or loading .mrc files, the pixel size is automatically set to what is found in the file header.

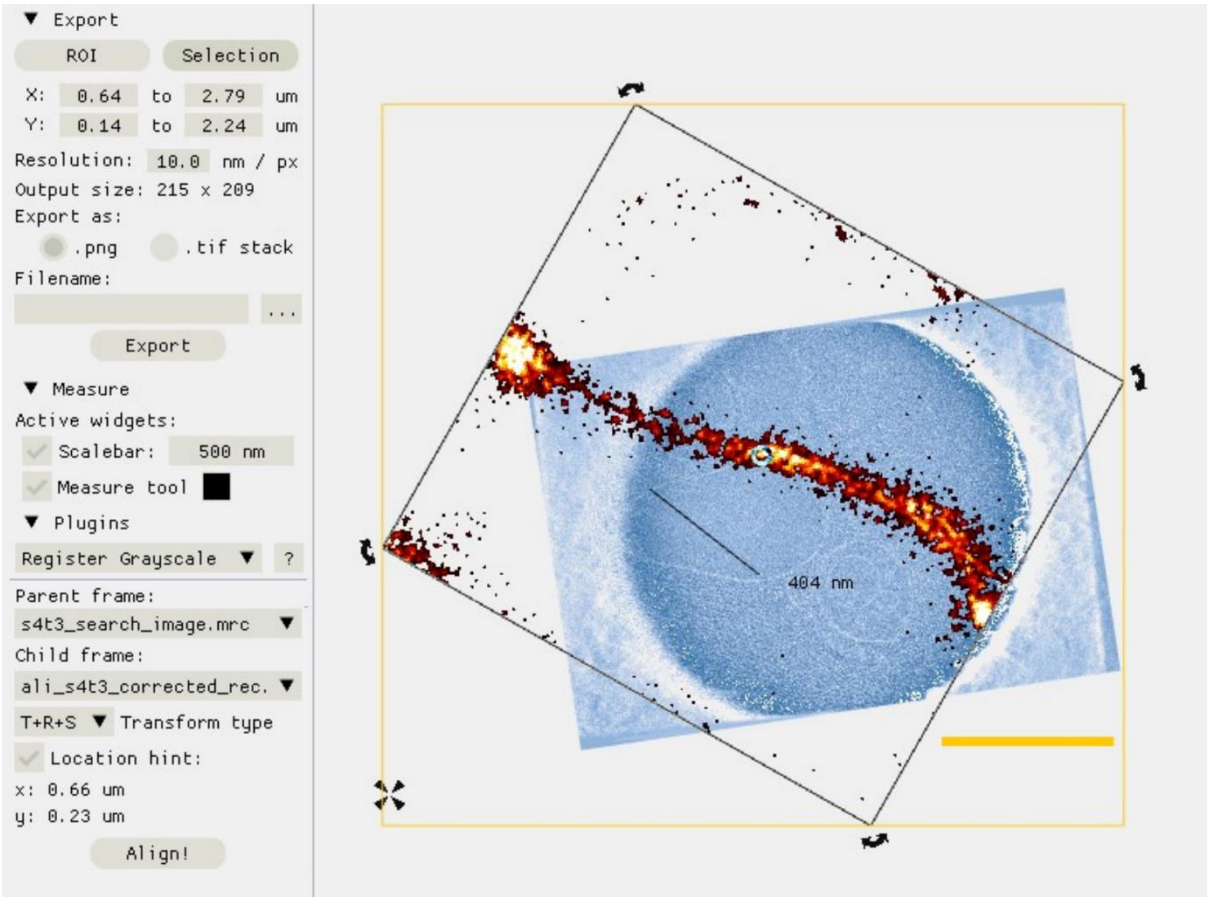
1.2: Visuals: change the look-up table and contrast settings of the selected image. A histogram of the pixel intensity distribution is shown (note: the vertical axis is scaled logarithmically). The 'saturation' setting dictates how pixel intensities below the minimum or higher than the maximum contrast setting are rendered. In *clamp* mode, the low/high values are clamped to the minimum and maximum colour of the LUT. In *discard* mode,

pixels outside of the selected range are discarded. In *discard min* mode, the low values are discarded, while high values are clamped.

1.3 Export: use this menu to export data as either a .png file or as a .tif stack. The region of interest for export can be set either i) manually (toggle the 'ROI' button), or ii) automatically snap to the selected image (toggle the 'Selection' button). The output resolution is limited only by the resolution of the data – the default export resolution is at 10 nm/pixel, but this value can be adjusted to any value (but the minimum is 0.001 nm/pixel). *Note: setting the resolution to a very high value (e.g. 0.1 nm/pixel) can result in large output image sizes that can easily exceed available RAM and freeze the software.*

1.4 Measure: toggle the scale bar on/off and the measure tool on/off. The scale bar can be positioned anywhere within the scene (click + drag) and is automatically adjusted in size when the camera zoom is changed. Change the colour of the scale bar by right-clicking the scalebar and editing the colour in the newly opened popup. When the measure tool is activated, clicking anywhere in the scene will spawn a new distance measurement. Clicking a second time will set the second measurement point.

1.5 Plugins: this menu provides access to the Correlation Editor plugins. There are two default plugins: 1) Register grayscale, and 2) Split RGB. Select a plugin and hover the "?" icon to find information on how to use it.



Overview of the Export, Measure, and Plugins submenus of the tools window. In this example, the scalebar (orange bar, set to 500 nm), distance measure tool (the line indicating 404 nm), and

the export ROI indicator (orange box) are visible in the scene. The scene contains a cryo-TEM tomogram overlayed with a super-resolution fluorescence image. The fluorescence image is rendered using the 'discard min' saturation mode.

Menu 2 – rendering

2.1 Interpolation - press this button to toggle between interpolation mode 1) nearest neighbour, and 2) linear interpolation

2.2 Alpha slider - adjust the alpha of the selected image. In the default blending mode (transparency), adjusting the alpha will change the transparency/opaqueness of the image. In other blending modes, alpha has a similar effect.

2.3 Slicer - for multi-slice images, such as tiffstacks or volume .mrc files, the slicer can be used to scroll through the volume.

2.4 Blend mode - set the blending mode for the selected image. Default is transparency blending, but other modes such as *multiply* or *subtract* can help enhance contrast when manually aligning similar frames, while other modes such as *sum* and *retain* can help blend fluorescence images with TEM or other fluorescence images.

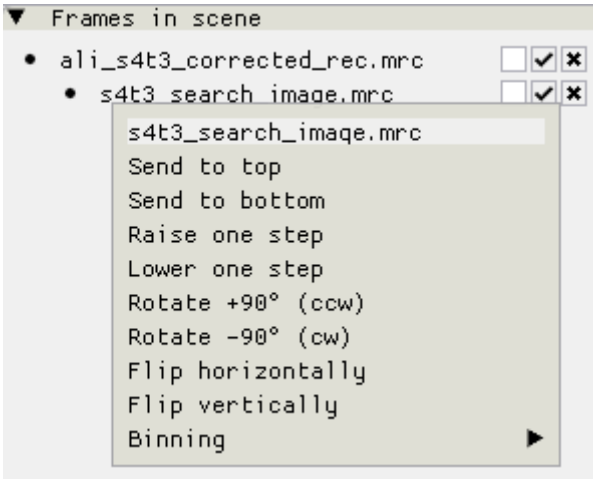
Menu 3 – frames in scene

This window provides an overview of the frames currently in the scene. Right-click an item to open a menu that holds options to change the title of the frame, or to rotate, flip, bin, etc., and move the frame up or down in the rendering order. Right clicking an image in the scene will open the same menu.

Dragging one item onto another will parent the drop target to the dragged frame. When a frame is the parent of another frame, moving, scaling, or rotating the parent frame will affect all child frames too. This can be useful when one frame is registered to another: setting the template frame to be the parent of the registered frame will keep the relative position of the child and parent frame the same, even when the parent frame is moved (but not when the child is moved!)

Note: binning and flipping affects the frame in the rendered view, but not the underlying pixel data. It is important to be aware of this to avoid confusion when attempting to register a flipped frame to a non-flipped frame – the registration tool works on the original pixel data, i.e., with neither frame flipped.

From left to right, the buttons alongside each title can be use to: i) change LUT to 'Custom colour' and edit the colour, ii) hide/unhide the frame, and iii) delete the frame.



The 'Frames in scene' window with a view of the context menu opened for the frame titles 's4t3_search_image.mrc'

Controls

Mouse

- Drag files (.png, .tif / .tiff, or .mrc) onto the Correlation Editor to add them to the scene.
- Click a frame once to select it. The editor gizmos (alongside the corners of the frame) will be in 'scale' mode.
- Click a selected frame again to toggle between the 'scaling' and the 'rotation' gizmos. When in 'rotation' mode, the pivot point of the frame is visible as a circle icon. Click and drag the pivot point to move it.
- Clicking and dragging the editor gizmos will scale/rotate the frame relative to the pivot point. Holding shift will override the pivot point and will rotate and scale the frame in local coordinates instead (i.e., with the centre of the image as the pivot point)
- Clicking and dragging a frame will move it.
- Pressing the middle mouse button and dragging the mouse moves the camera.
- Many of the frames with numerical text in the tools window can be dragged to change their value.
- Right-click a frame in the scene or in the 'Frames in scene' window to access a context menu, with options for rotating, mirroring, and moving the frame to the back-/foreground.

Keys

- The arrow keys can be used to move a frame in the scene. The default step size for a button press is 100 nm (can be changed in Settings). Hold Shift to increase the step size a factor 10, or Control to reduce is a factor 10.
- Press Delete to remove a frame from the scene.
- Hold Shift and scroll the middle mouse button to zoom in/out on the scene.
- Number keys (1 to 9) are a shortcut to change the blend mode of the selected frame.
- Press the space bar to center the view on the selected frame.

Correlating TEM and fluorescence microscopy data

Tutorial: correlating a super-resolution fluorescence microscopy dataset with a cryo-electron tomogram.

In this tutorial, the general workflow of correlating TEM and fluorescence microscopy datasets is demonstrated. We will use cryo-electron tomography data, acquired with a Talos Artica 200keV microscope and using the TFS Tomography software. The EM dataset comprises the following images:

- An 'Overview' image, at 15.375 nm/pixel.
- A 'Search' image, at 1.826 nm/pixel.
- An 'Exposure' image, at 0.549 nm/pixel.
- A (binned) reconstructed Tomogram, at 0.835 nm/pixel.

Note that these Overview, Search, and Exposure images are the images saved by the TFS Tomography software upon adding a site to the batch processing list and refining it.

The light microscopy data consists of the following:

- A reflected light image that is used to register to the TEM overview with. (64 nm/px.)
- A PSF-fitting based super-resolution reconstruction generated using scNodes. (6.4 nm/px)

Step 0: before entering the Correlation Editor, it is important to outline exactly how the final correlation between the super-resolution fluorescence data and the cryo-electron tomogram is performed. To enable a correlation between fluorescence and TEM, we have to use the reflection image as a 'bridge' between the two different types of microscopy data: the reflection image is 1) aligned with the fluorescence data by virtue of being acquired in the same microscope, and within a short enough timeframe to avoid drift, while it also 2) contains features that are highly recognizable in the TEM Overview image as well. In particular, the holes in the holey gold support film are easily spotted in both the reflection and the TEM overview image.

While it is straightforward to acquire a reflection and fluorescence image within a short timeframe, thus ensuring spatial correlation between these two light microscopy images, the drift correction step that is typically required in cryo-fluorescence super-resolution microscopy can undo the fluorescence-to-reflection registration. To avoid this problem, it is important *only to register fluorescence timelapses to a template frame that is itself registered to a reflection frame*. This was done for the data used in this tutorial.

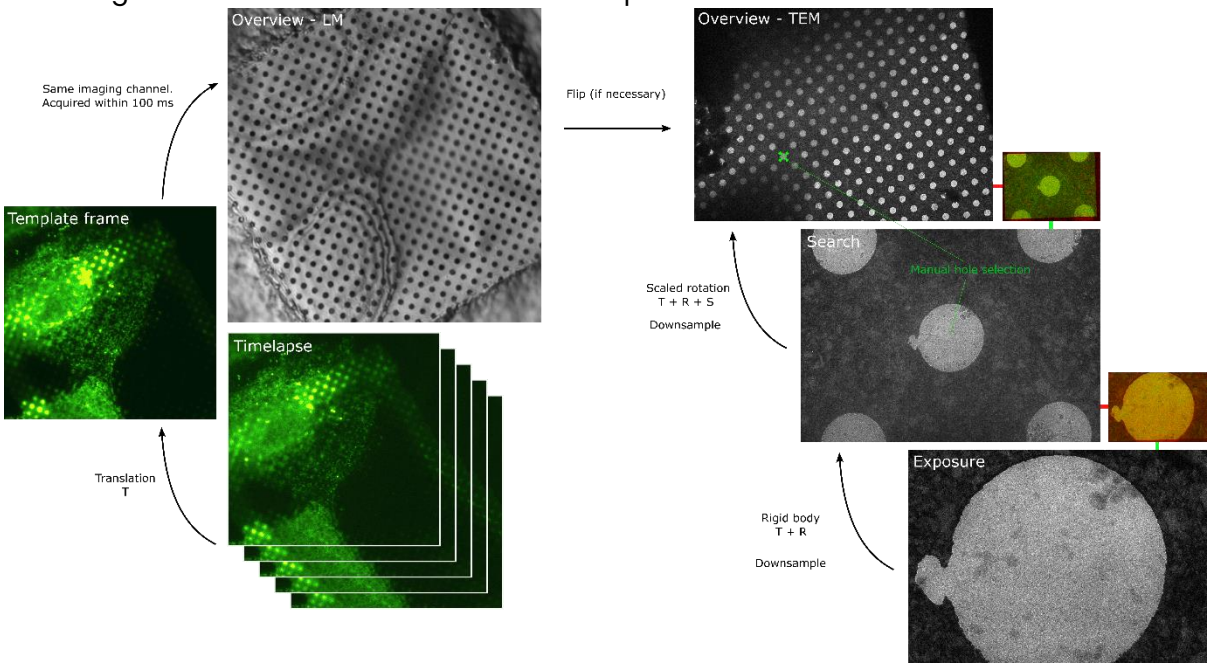
Note that the above is only one way of registering and correlating reflection/fluorescence/electron microscopy images – you may find other ways of achieving the same result that are better suited for your experiments.

Since the fluorescence timelapse that was used to generate the super-resolution fluorescence image used in this tutorial was registered with a fluorescence template frame that was itself registered to a reflection image by means of fast acquisition, the super-resolution image is also aligned with the reflection image. (Note: when rendering reconstructions, scNodes always uses the size of the (ROI in the) input images as the ROI for

rendering; this ensures that the alignment between and aspect ratio of the input dataset and output image is not changed).

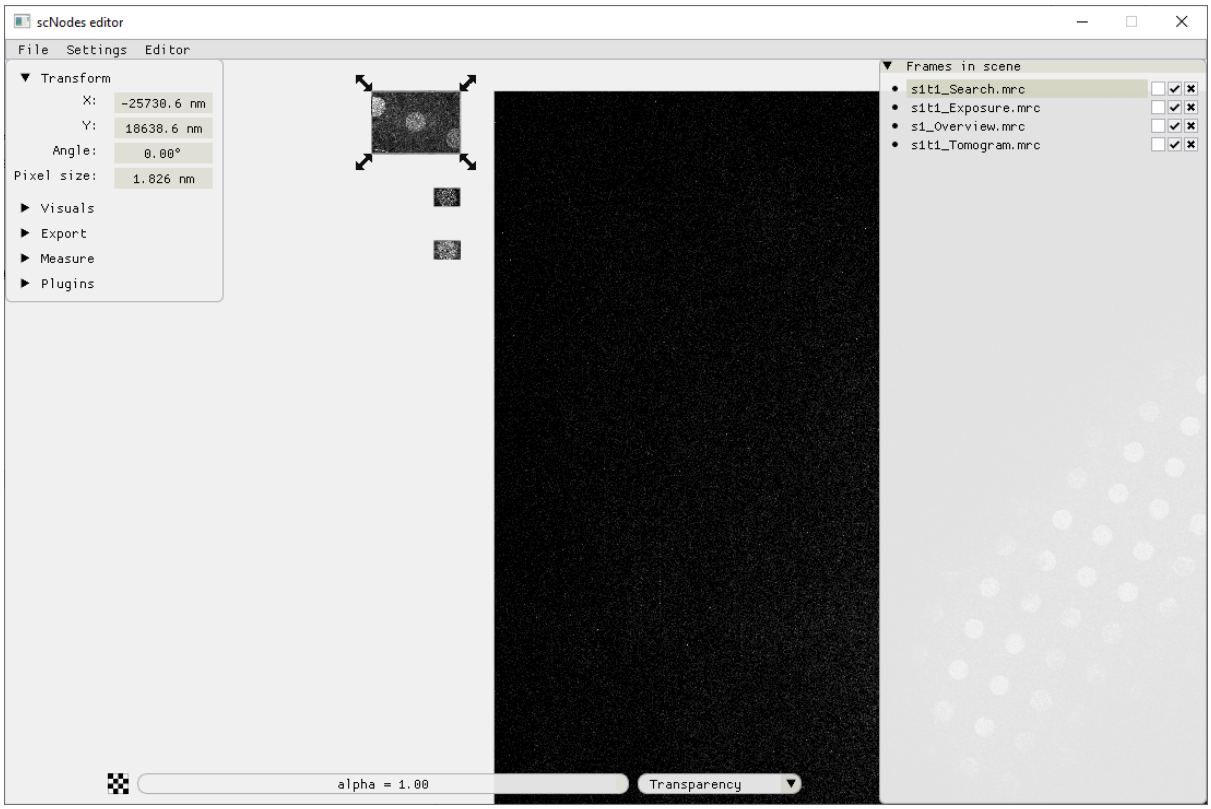
Finally, we note that the images used in this tutorial were loaded into the Correlation Editor from the PC memory, and *not* by porting them into the Correlation Editor via the Image Viewer. Typically, one would be generating images in the Node Editor, right-click the output in the Image Viewer, and select 'Add to correlation editor,' to import fluorescence microscopy data.

The image below illustrates the full correlation procedure.



Overview of the procedure to correlate fluorescence & electron microscopy images. The final goal is to correlate the TEM 'Exposure' image with the full fluorescence timelapse. The timelapse can be used to generate a super-resolution fluorescence image, which is then also correlated to the TEM Exposure image. Similarly, for electron tomography datasets, the final tomogram can be aligned with the TEM Exposure image to create a correlated super-resolution fluorescence and electron tomography image.

Step 1: import the TEM data by dragging the files onto the Correlation Editor. The frames will be listed in the 'frames in scene' – see the image below.

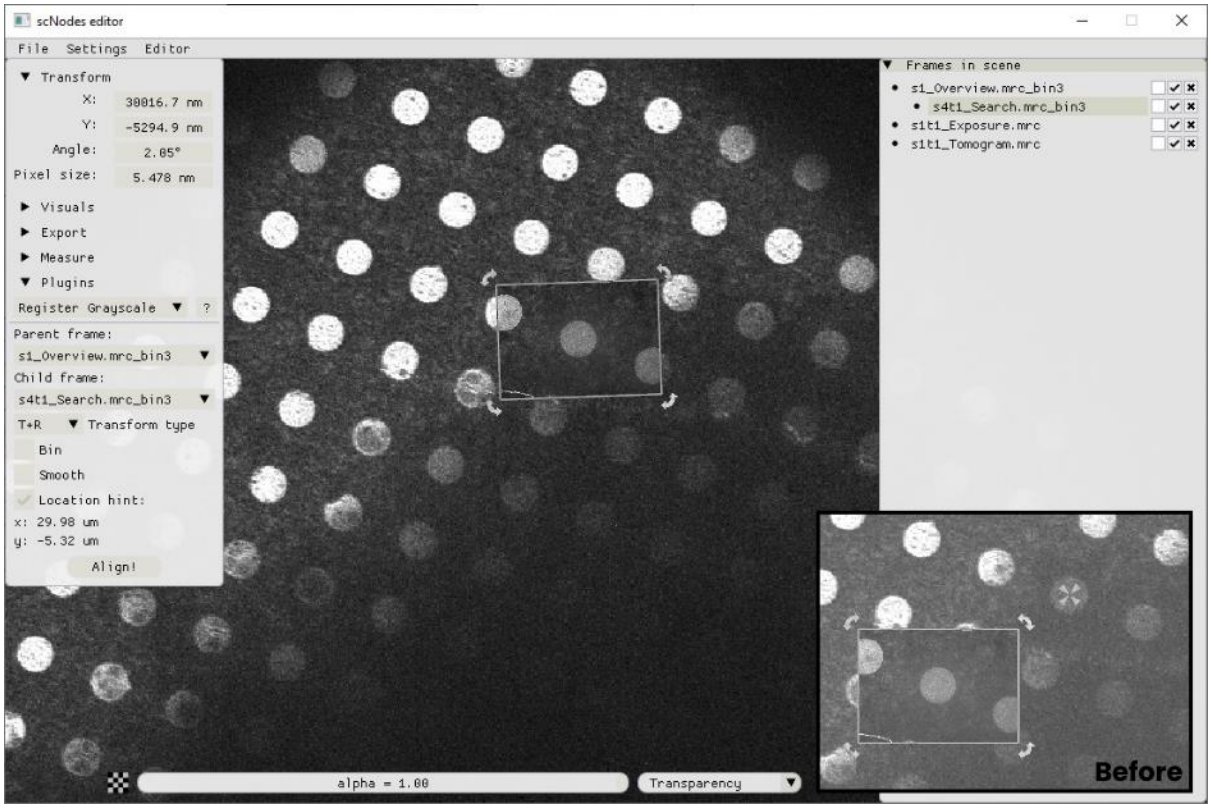


Step 2: align the Overview and Search images. The data can appear clearer to the human eye when it is binned (Plugins -> Apply binning) somewhat – a factor 2 or 3 should suffice – but this is not necessary for the alignment. Since it helps with the visualization, we will use a bin factor of 3 for the tutorial.

As our Overview image contains many similar holes, fully automatic alignment of the Search and Overview is not possible; we need to provide an approximate coordinate for where the Search image would be positioned within the Overview image.

In the Plugins menu, choose 'Register Grayscale', select the Overview image as the parent frame and the Search image as the child frame. Choose transform type 'T + R' (translation plus rotation). The checkboxes 'Bin' and 'Smooth' can be activated to perform binning and/or gaussian blurring on the images-to-be-registered. This is internal to the plugin only, meaning the pixel data of images in the scene will not be affected.

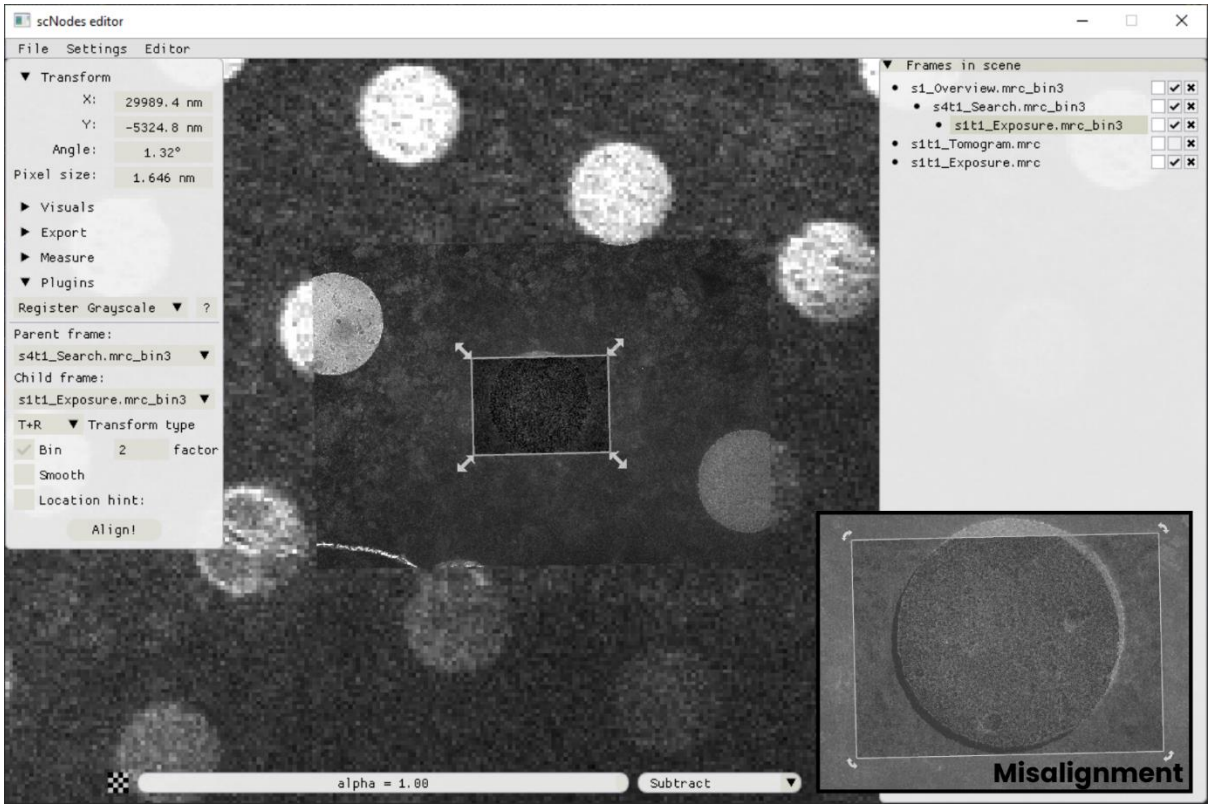
Next, check the 'Location hint' box. A marker will appear in the scene. Drag this marker onto the Overview image and place it at the approximate location of where the Search image should end up. Finally, press 'Align'. Note that the Search image is now a child of the Overview image.



Aligning the Search to the Overview image. Main view: the result of the 'Register Grayscale' plugin. Inset (bottom right): the scene before pressing 'Align' in the 'Register Grayscale' plugin window. Note the crosshairs icon in the hole in the overview image – this is the 'Location hint' marker.

Step 3: aligning the Exposure image onto the Search image. Again, use the 'Register Grayscale' plugin. This time a location hint is not required – uncheck the box. To aid visualization, we've also binned the Exposure frame a factor 3 prior to alignment.

The result of this second registration step is shown in the image below. To illustrate the use of using different blending modes, the Exposure image has been set to blend mode 'Subtract'. Note the difference between the main image and the inset in the bottom right, where the alignment is intentionally erroneous.



Step 4: align the Tomogram with the Exposure image. Select the Tomogram frame. The slicer will pop up, allowing you to scroll through the volume. When using the Register Grayscale plugin on a multi-slice frame, the currently shown frame will be used for the alignment. Similarly, the 'Apply binning' plugin will output a binned copy of the currently selected frame only.

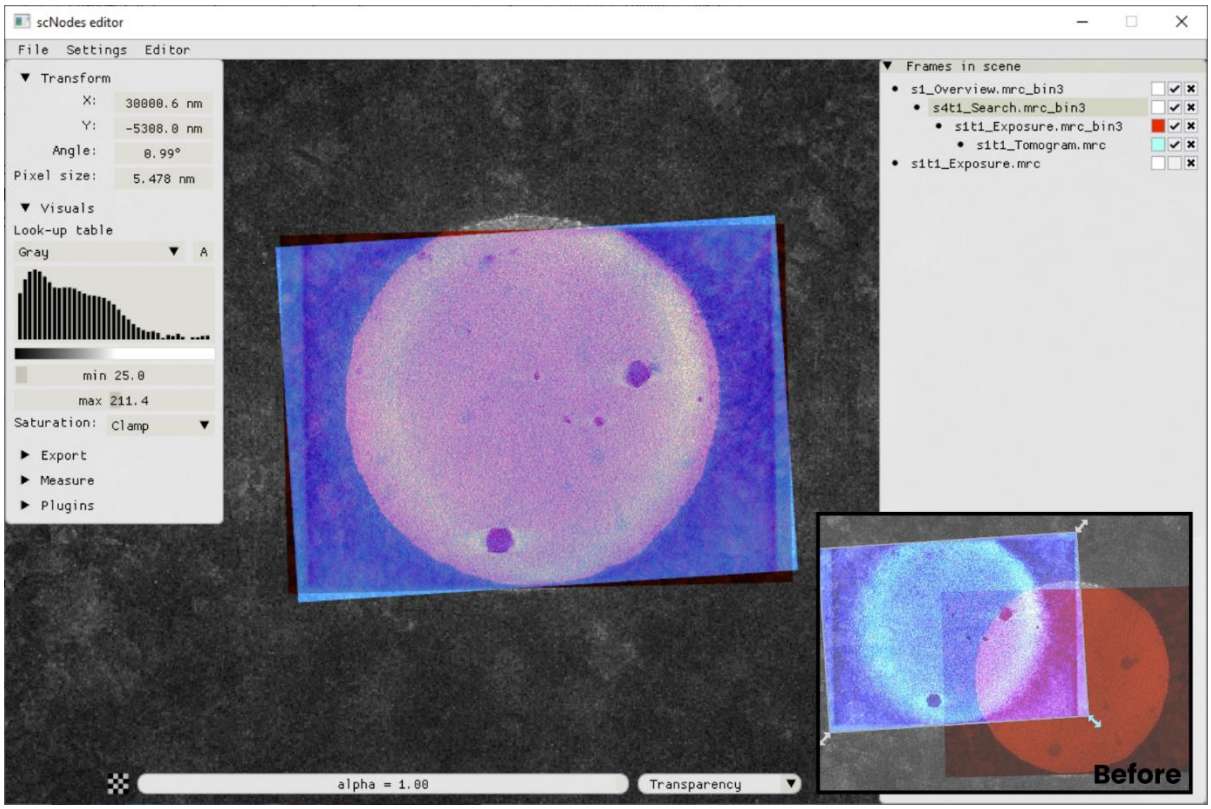
The Register Grayscale tool can sometimes fail to properly register Tomogram slices to Exposure images, e.g. when high-contrast features such as the edges of a hole are distorted due to flattening of the Tomogram. In such a case, manual registration may be required.

In this example, we will manually align the Tomogram with the Exposure image. To do so, we move the Tomogram frame to the top of the frame stack (right click -> send to top), and move it onto the Exposure image. Toggling the interpolation mode (checkerboard button in the bottom control panel) can aid visual inspection.

Next, set the LUT for the Exposure image to red (in the 'Visuals' drop-down menu of the Tools window), and to blue for the Tomogram. Select a slice in the Tomogram in which multiple features are visible that are also easy to spot in the Exposure image, such as ice particles. Set the blend mode for the Tomogram frame to 'Sum'. In the 'Frames in scene' window, drag the Tomogram frame onto the Exposure frame to make it a child of the Exposure frame.

Move the Tomogram frame such that one of the features overlaps with the corresponding feature in the Exposure image. Click the Tomogram image to toggle the editing move from 'scale' to 'rotate'. The pivot point marker will appear – move it onto the overlapped feature.

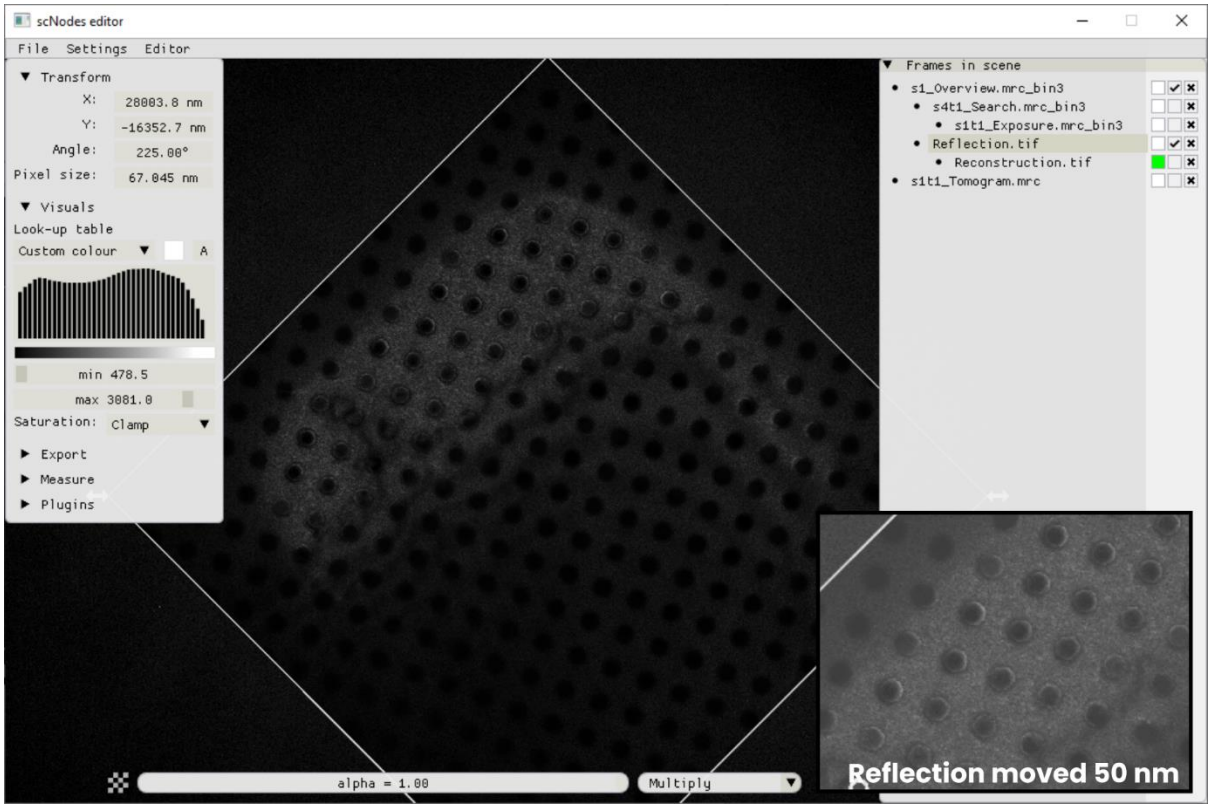
This way, rotating (and scaling) the image will keep this initially registered feature in place, while allowing you to align the remaining features.



Step 5: importing the fluorescence data. For convenience, the Image Viewer window has an option to send frames to the Correlation Editor: right-click in the Image Viewer and select 'Add to Correlation Editor.' Use it if your fluorescence microscopy data is loaded in the Node Editor.

After importing the Reflection and super-resolution reconstruction (Reconstruction) images, make the Reconstruction a child of the Reflection and hide it (checkbox in the 'Frames in scene' window). Manually register the Reflection to the TEM Overview image. This can be done in similar fashion to the previous alignment of the Exposure and the Tomogram frames. Alternatively, use blend mode 'Multiply' with both images' LUT set to 'Gray'. The trick is to find the same feature/hole in both the Overview and the Reflection image, overlay those, move the pivot onto it, and then rotate and scale (when image sizes do not match up, the calibration of the light microscope pixel size is perhaps a bit off. The calibration of the TEM pixel size can also be off by a few percentage points.) the Reflection image in order to align the frames.

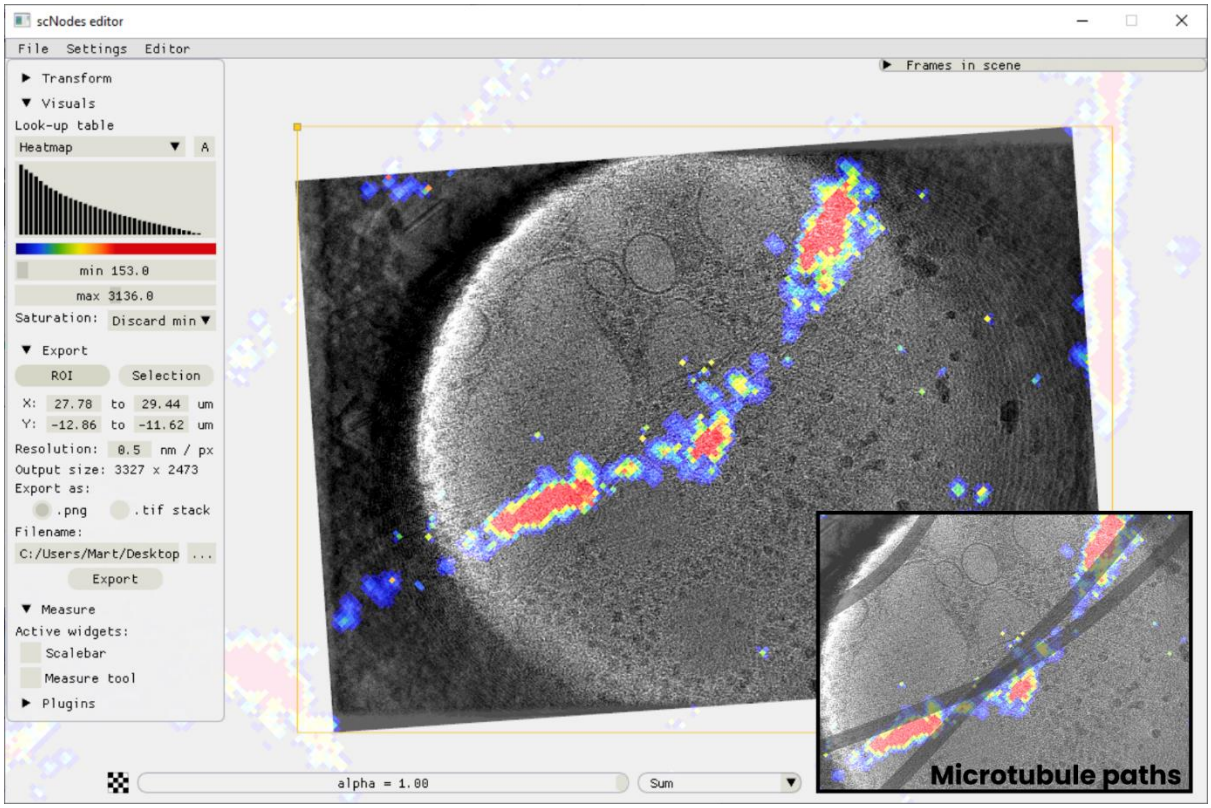
The result is shown in the image below. The inset shows the same scene, but with the Reflection image moved 50 nm to the left; clearly, this alignment is worse.



Step 6: because we set the Reconstruction frame to be a child of the Reflection frame, once the Reflection frame is in place the Reconstruction is aligned as well.

Hide the Reflection frame and make the Reconstruction frame visible. Set the blend mode to ‘Sum’ and the LUT to, e.g., ‘Green’. Clamping mode ‘Discard min’ can also help increase visual clarity.

In this example, the protein of interest was MAP2 – microtubule associated protein 2, which binds to tubulin – labelled with rsEGFP2, a reversibly photo-switchable green fluorescent protein. In inset in the image below two microtubules that are found in the tomogram are annotated.



Creating a custom Correlation Editor plugin

Similar to Nodes in the Node Editor, plugins for the Correlation Editor are each separate modules that are dynamically loaded into the software. Each plugin is a single python file located in the 'ceplugins' subdirectory found in the main programme directory.

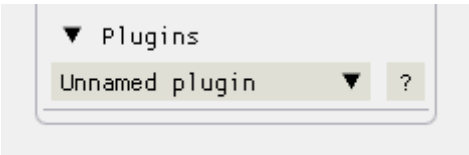
Creating and installing custom plugin is thus only a matter of: i) writing code that satisfies the minimum requirements for a Correlation Editor plugin, and ii) placing the file in the *ceplugins* folder.

Tutorial: creating a Correlation Editor plugin that generates a blurred version of a frame.

Step 1: open the 'custom_plugin_template.py' file found in the *ceplugins* folder and browse through it. Create a new .py file in the same folder, give a name (e.g. blur_frame.py), and copy the following code into the file in order to satisfy the minimum requirements for a Correlation Editor plugin:

```
from ceplugin import *


def create():
    return BlurPlugin()


class BlurPlugin(CEPlugin):
    pass
```

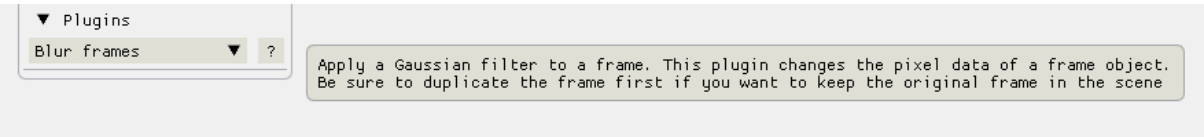
In the software, the plugin would now be available in the tools window -> Plugins menu -> plugin selection drop down menu, as a plugin called 'Unnamed plugin'. Upon selection the window would look as below:



Step 2: name the plugin and write a description. To do so, override the CEPlugin class' default member variables *title* and *description*:

```
class BlurPlugin(CEPlugin):
    title = "Blur frames"
    description = "Apply a Gaussian filter to a frame. This plugin changes
the pixel data of a frame object.\n" \
                  "Be sure to duplicate the frame first if you want to keep
the original frame in the scene"
```

Now, when the '?' icon is hovered, the plugin appears as follows:

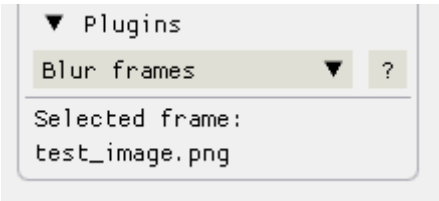


Step 3: adding a way to select which frame to process. There are a number of default widgets that can be used to select a frame (see the custom_plugin_template.py). For this plugin, let's use whichever frame is selected in the editor. The default function 'CEPlugin.widget_show_active_frame_title()' does exactly this: it adds a text field displaying the title of the selected frame to the interface, and the function returns the corresponding CLEMFrame object.

Define a variable 'selected_frame' in the __init__(self) method of the BlurPlugin class, and override the CEPlugin.render() method to create a custom interface for the BlurPlugin class. In it, we call the show active frame widget mentioned above:

```
def __init__(self):
    self.selected_frame = None

def render(self):
    self.selected_frame =
CEPlugin.widget_show_active_frame_title(label="Selected frame:")
```

The plugin now appears as:



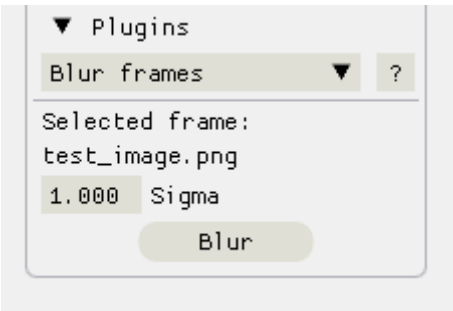
Step 4: adding settings and a button to activate the plugin. For the blur we can use the *gaussian_filter* function from the scipy.ndimage module – this is a dependency of the scNodes core, so we're not adding any new dependencies. The *gaussian_filter* function takes two inputs: the image pixel data array, and a value for the sigma of the gaussian kernel. Let's add that sigma as a setting to the plugin. For this we will use an imgui input_float field.

```
def __init__(self):
    self.selected_frame = None
    self.sigma = 1.0  # blur kernel standard deviation in pixels.

def render(self):
    self.selected_frame =
CEPlugin.widget_show_active_frame_title(label="Selected frame:")

    imgui.set_next_item_width(50)
    _, self.sigma = imgui.input_float("Sigma", self.sigma)
```

Next, add a button, using the default 'centred_button' widget. We'll implement the behaviour of a button click in the next step.

```
def render(self):
    self.selected_frame =
CEPlugin.widget_show_active_frame_title(label="Selected frame:")

    _, self.sigma = imgui.input_float("Sigma", self.sigma)

    if CEPlugin.widget_centred_button("Blur"):
        pass
```

The plugin now looks like this:



Step 5: processing an image. When the button is clicked, we want to change the pixel data of the selected frame and have that change be visible in the scene. The pixel data of a frame (i.e. of a CLEMFrame type object) is stored in the .data member variable. Every

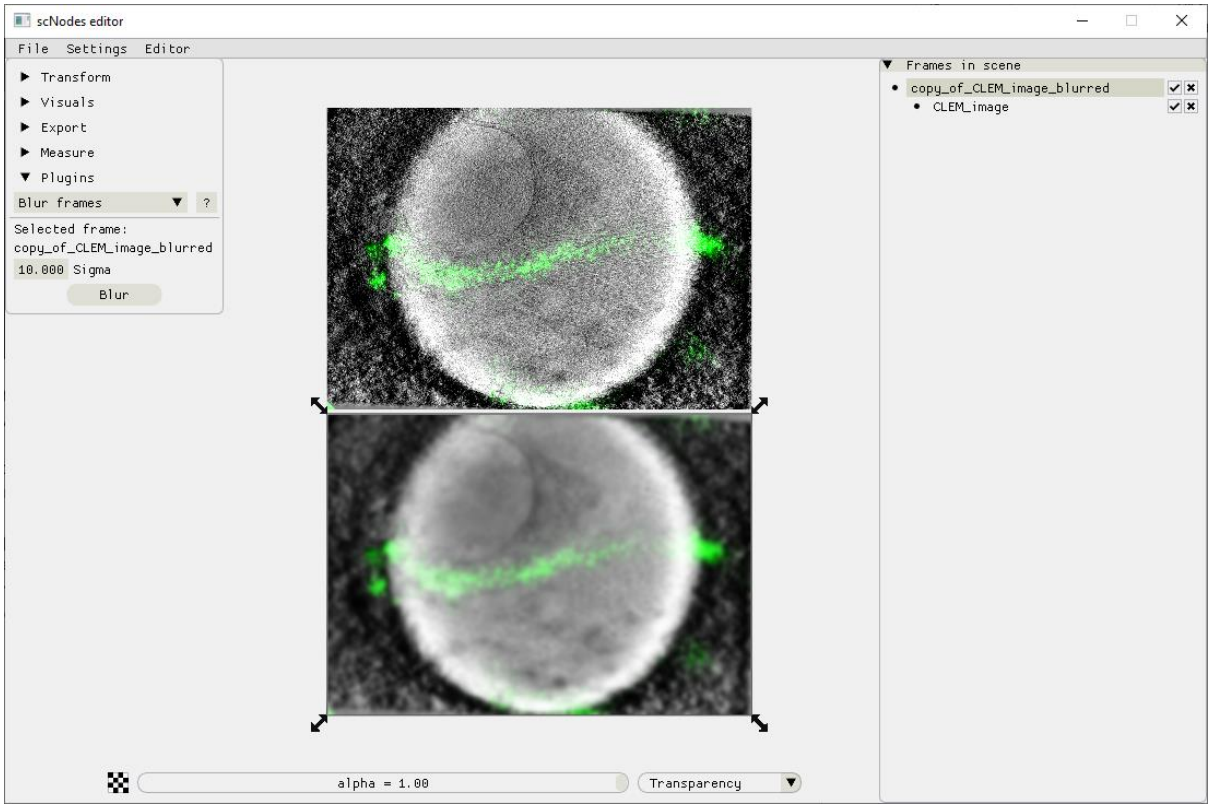
frame has a *texture* (handled internally), which is the GPU-side storage of the pixel data. Whenever the pixel data of a frame is edited (this happens on the CPU), this texture has to be updated. This can be done by calling the .update() method of a CLEMFrame object.

We can add a new method to the BlurPlugin class to take care of processing the images, e.g. 'process_image()'. CLEMFrames can either be grayscale (for .tif and .mrc files) or RGB (for .png files). The .is_rgb variable tells us whether a frame is RGB: it is True when so, and False otherwise. For this plugin we only want to blur the image spatially. So, for RGB frames, the code is a little different than for grayscale frames – see the code block below.

```
    if CEPlugin.widget_centred_button("Blur"):
        self.process_frame()

def process_frame(self):
    out_frame = self.selected_frame.duplicate()
    img_data = out_frame.data

    if self.selected_frame.is_rgb:
        img_data[:, :, 0] = gaussian_filter(img_data[:, :, 0], self.sigma)
        img_data[:, :, 1] = gaussian_filter(img_data[:, :, 1], self.sigma)
        img_data[:, :, 2] = gaussian_filter(img_data[:, :, 2], self.sigma)
    else:
        img_data[:, :] = gaussian_filter(img_data, self.sigma)
    out_frame.data = img_data

    cfg.ce_frames.append(out_frame)
    cfg.ce_frames.remove(self.selected_frame)
    out_frame.update()
```

Note that the if clause belonging to the 'Blur' button has also been edited: it now calls the new process_frame method.

In the first line of the process_frame method, we create a local duplicate of the input frame by calling .duplicate() on the CLEMFrame object stored in self.selected_frame. Next, we edit the pixel data of this frame. Then we add the duplicate frame to the scene by appending out_frame to cfg.ce_frames, the list of all frames the scene. We also remove the original frame. Finally, we call .update() on the new frame. The end result is that there is now a new frame in the scene which identical to the input frame in all respects but the pixel data, which is blurred.



Installing a plugin

Installing a plugin is as easy as installing a custom node: copy the .py file that defines the plugin into the *ceplugins* directory of the main programme folder and reboot the software.