TP IV Laboratory report

Eda napari: implementation of a napari plugin

a plugin for visualising framerates and time from timelapse images

Steven Brown
steven.brown@epfl.ch
Supervisor: Willi Stepp

Laboratory of Experimental Biophysics, EPFL

June 15, 2022

Contents

1 Introduction

During biological observations numerous images are captured and the data is stored. The user must then choose a programme for viewing and analysing the data. Napari, a python based open source image viewer, is adapted for biology and facilitates the addition of custom plugins [1]. The aim of this project is to programme, following the napari architecture, a plugin for visualising metadata from image sequences.

More specifically, a plugin that is designed for images captured with an event driven acquisition algorithm. Capturing dynamical processes in biology can be carried out with the aid of time-lapse videos. However, theses processes can take a certain amount of time to occur and the dynamics of theses processes that take place may not be linear in time. This motivates the usage of event driven image acquisition. When activity is detected, then images are taken with a higher frame rate. This has several advantages. Firstly, large data is stored only for relevant time frames. If fluorescent imaging methods are used, this will also have the effect of minimising fluorophore bleaching, and thus enabling longer observations times [2].

The **eda-napari** plugin will be designed for visualising variable frame rates from image sequences. The programme has direct applications at the EPFL experimental laboratory of biophysics. In the laboratory, a neural network for detecting activity and adapting frame rates is used with an iSIM microscope for observing mitochondrial fission [2]. The plugin will allow visualising such data, acquired with adaptive capture times.

2 Napari

Napari is a multi-dimensional image viewing and analysis software [3]. It is an open and free to use software, that started back in 2018. Napari filled the need of a python based n-dimensional viewing software. Previously, users needed time consuming procedures to navigate between non-python based analysis software such as ImageJ and useful python packages [1]. Python has a great deal scientific packages that can be applied to image analysis, for instance deep learning packages scikit-learn, TensorFlow and PyTorch. Napari, being python based, can therefore easily integrate machine learning analysis tools. Napari's graphical interface is written with Qt. All sort of user interaction can thus be implemented within the Qt framework.

Napari itself can directly be used as an image viewer. Images, for example tiff files, with several layers can be displayed and napari holds the basic tools for colouring the image layers. Napari's main display consists of a main window, a scroll bar and side docks (Figure 1). The programme is especially created for allowing the user to install additional plugins. Theses plugins are simple to install and add image analysis tools, for example image segmentation, tracking or specific file readers.

For sharing plugins, napari is linked to **napari-hub**. Napari-hub allows users to have access to a whole environment of plugins. The plugins can be installed into napari and with the possibility of giving feedback. Napari-hub facilitates the search for plugins and improves communication between plugin creators and users.

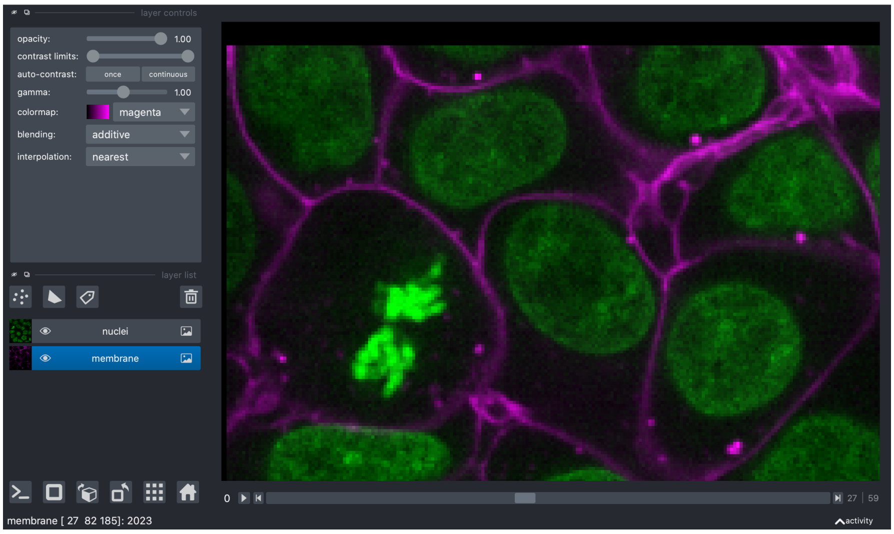

Figure 1: Example of napari's viewer display, visualising 3D + 2 channels cell images [4]

3 Creating a plugin

As previously said, it is possible to install plugins to napari and to add extra features to the base image viewer. This section will describe the main points for creating a plugin and is based on napari's own tutorial[1]. For napari to detect and include a plugin, a specific minimum structure needs to be followed (Figure 2). A napari plugin can be seen simply as a python package and must have certain required files for installation.

```
plugin-name
├── plugin_name
│   ├── __init__.py
│   └── napari.yaml
├── pyproject.toml
└── setup.cfg
```

Figure 2: Starting project directory tree for creating a npe2 napari plugin [4]

The *pyproject.toml* and *setup.cfg* files are specific to the package installation. Pyproject.toml uses setuptools to package the plugin, whereas setup.cfg defines package metadata, install requirements and sets a napari manifest as entry point to allow napari to find the plugin (Code 1). Napari has a second generation plugin engine npe2, with the old one soon to be deprecated[2]. With npe2, a *manifest (.yaml)* file must be defined. This file allows napari to discover the plugin and sets the different contributions. The contributions can be widgets, readers or simply commands. For clarity, plugin classes are often separated into multiple .py files for different types of contributions. For our example, we will consider a *_widget.py* file (Code 1). The widget file would contain classes that define widgets, for example a simple button with interactions.

[1] Your First Plugin: `https://napari.org/plugins/first_plugin.html`

[2] napari-plugin-engine was the first generation version. It used hook implementations "@napari_hook_implementation" to allow napari to detect the plugin.

Code 1: Example code of pyproject.toml, napari.yaml and setup.cfg, inspired from napari [4]

/pyproject.toml

```
[build-system]
requires = ["setuptools", "wheel"]
build-backend = "setuptools.build_meta"
```

/napari.yaml (manifest)

```
name: plugin-name
contributions:
  commands:
    - id: plugin-name.command_name
      title: command title
      python_name: plugin_name._widget:widget_class_name #path of class or function
  widgets:
    - command: plugin-name.command_name
      display_name: Title in napari tab window
      autogenerate: true
```

/setup.cfg

```
[metadata]
name = plugin-name
version = 0.0.1
classifiers =
    Framework :: napari

[options]
packages = find:
include_package_data = True
install_requires =
    napari
    #add all the other packages needed for programming the plugin
[options.entry_points]
napari.manifest =#tell napari where to find to manifest
    plugin-name = plugin_name:napari.yaml
```

Once the structure is in place, the plugin can be pip installed and napari will detect and encapsulate the plugin. The plugin will be visible in the plugin's window tab of the napari programme.

4 Interactions between plugin and napari

The plugin can only interact with the napari viewer through the *napari.Viewer* class. This class is the napari viewer instance and contains the window, widgets, layers, scroll bars and more. All of which are visible in the viewer. Understanding how to access the different sub-classes and information from this class is the key to coding the plugin interactions. To give the reader a better intuition on how the information is stored, some important sub-classes are made explicit (Code 2). Once the plugin has access, the plugin can also update attributes seen in the viewer. This interaction is bidirectional as plugin functions can connect to viewer *events*. Some of napari's modules, for example *layers*, *Window* and *dims* have events. A module can have several events and each event can be reacted to. Qt implements events with what one calls signals and slots [3]. The event emits a signal that can then be connected to slot function that executes an operation.

Code 2: Some basic napari.Viewer python access commands

```
napari.Viewer.window #the window widget that contains all dock widgets
napari.Viewer.layers #list of image layers
napari.Viewer.layers[0].source.path #path of the source of first layer
napari.Viewer.layers.events#events class of layers containing all the events a layer can have
napari.Viewer.dims #the napari scroll bar widget
napari.Viewer.dims.events.current_step #the current step event of the scroller
napari.Viewer.dims.events.current_step.connect(function)#connect a slot function to an event
```

5 Eda-napari

The plugin programmed for this project is named eda-napari, with eda for event driven acquisition. The whole plugin code can be accessed on GitHub via: `https://github.com/LEB-EPFL/eda-napari`. The main structure of the plugin is based on section 1. The widget code is placed in a separate file *_widget.py*. The structure of the final code is shown in the annex 7.1. The full code of the *_widget.py* file is also available in annex 7.2.
The plugin has two main features: *Frame rate widget* and *Time scroller widget*. Each feature is implemented as a python class that inherits from QWidget. The class is linked to the napari viewer during initialisation by passing as argument *napari_viewer* [8] (Code 3). During the instantiation, napari replaces arguments with values equal to "napari_viewer" with the viewer class.
The graphical interface is coded in qtpy which is the binding language between Qt (Framework coded in c++) and Python. The main structure consists of creating a Qt layout and then populating it with widgets, for example buttons, labels or figures.
The details of eda-napari's whole class methods will not be explained since the fully annotated code is in the annex and on the Github. However, important implementation choices and references to code extracts will be presented in this section.
The plugin installs automatically napari-aicsimageio [5]. AICSImageIO is an image reader specialised for biological image formats. It is useful for visualising multiple layers in napari. This reader is not necessary for the plugin's functionality. It is also possible to use the builtin reader but it is advantageous to use AICSImageIO. It can be selected as the default reader, when opening a tif file for the first time.

5.1 Frame rate widget

Frame rate widget (Code 3) is the major feature of this project. The widget reads metadata from tif files originating from microscope imaging. From a time-lapse image stack, the widget reads the capture time of each individual image. This time data is stored and the frame rate is approximated from this data and then ploted in a dock widget in the napari viewer window. The widget is implemented as a class with an initialisation, class attributes, connections to napari events and class methods.

Adding a plot to the widget

It is possible to embed a matplotib canvases in the Qt interface [6]. A Figure canvas is created with axes programmed, as for a typical matplotlib plot. The frame rate is plotted with respect to

[3] Qt Signals and slots documentation: `https://doc.qt.io/qt-6/signalsandslots.html`

time or frame number, with a Qpushbutton allowing to switch between the two. The evolution of the time is also plotted versus the frame number. A vertical line, showing the current frame, is linked to the napari's scroll bar (*dims*) and is updated if the *current_step* is modified (Code 3, l. 124). For better visibility, a table of the current values is added to the layout under the plots (Figure 3).

Figure 3: Base canvas for the widget plots with interactive buttons and display labels.

Adding Slow motion shape

A slow motion symbol (Figure 4) is implemented to distinguish two cases: slow frame rate and fast frame rate. Two cases is sufficient since ideally a sample has activity or doesn't. During activity, this is when a higher image frame rate is used. To distinguish these two states otsu thresholding from skimage[4] is imported. The package determines the frame rate threshold delimiting the two cases. The easiest way to display an additional image is to use napari viewer's *add_shape*() (Code 3, l. 240) function. This creates a layer automatically and allows quick drawing of polygons. Since several layers can be inserted for one tif file, a QTimer (Code 3, l. 92) is necessary to allow the complete upload of all the layers. If the widget is activated before loading an image, the QTimer will stop the misplacement of the slow motion layer once data is read.

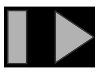

Figure 4: Slow motion icon.

5.2 Time scroller widget

Time scroller widget is the second class widget of this plugin (Figure 5). The widget creates a QScrollbar that discretizes time. The goal of the widget is to animate the time-lapse linearly in time. This is useful because napari's scroll bar animates individual frames with a constant time interval without taking into account the actual time from the metadata. The class is coded similarly to the frame rate widget and is set in a second dock widget.

[4]Thresholding documentation: `https://scikit-image.org/docs/stable/auto_examples/applications/plot_thresholding.html`

Figure 5: Base display for the time scroll bar widget.

Animating linearly in time

For an animation, a relevant time interval needs to be determined. To make this choice, one needs to consider that the animation uses computational power and that the human eye or computer can only detect a limited amount of frames per second. It is therfore unnecessary to overdiscretize. As an arbitrary approximation, 1/4th of the minimum time between image frames is set as the time interval (Code 4, l. 420). A QTimer is linked to a play button and ensures a regular update for the animation. The scroll bar is connected to napari's scroll bar for bidirectional connections. To avoid double calling of events some connections are briefly disconnected (code 5, line: 458-477). For the animation, an index list (Table 1) is created for knowing when to update the viewer's image and not simply just the time incrementation. For speeding up or slowing down the animation, a hybrid method is used. For speeding up the simulation, under a certain critical time interval, $critical\_ms$ = 160ms (equivalent to 60Hz computer fps), instead of changing the QTimer's tick time, the step size is increased by a factor 2 (code 5, line: 349-356). The step size is defined as the amount of time intervals skipped after the timer's timeout. Having a tick time under the critical time interval can cause wasteful computations and potential animations problems.

Table 1: Example of frame number indexing in a list for the animation. Between each column, one time interval elapses. The step size is set to one.

Frame 1	Frame 1	Frame 1	Frame 2	...	Frame n

Step size=1

5.3 Results

The plugin is successfully detected by napari. Both widgets can be opened simultaneously and read data from tif time-lapse images. Example data with variable frame rates were used to verify the execution. The different features of the widgets are demonstrated in the following figures and the AICSImageIO reader is used for it's multiple layer display.
Firstly, figure 6, shows the detection of the plugin in napari. Once the frame rate widget is activated, the plots are displayed on the side, in a dock widget. Since no slow motion icon is visible for the image frame, the threshold frame rate has already been reached. This is the wanted result, since the frame in the viewer is taken during a fast frame rate. The image data used for figure 6 corresponds to a time-lapse of a double channel image sequence. The first channel, in pink, visualises mitochondria fission and the other channel the protein drp1. The camera frame rate used was not implemented with a neural network sensing activity. The frame rate function here is simply for test purposes. On the figure, the two graphs highlight in different ways the rate of image acquisition.

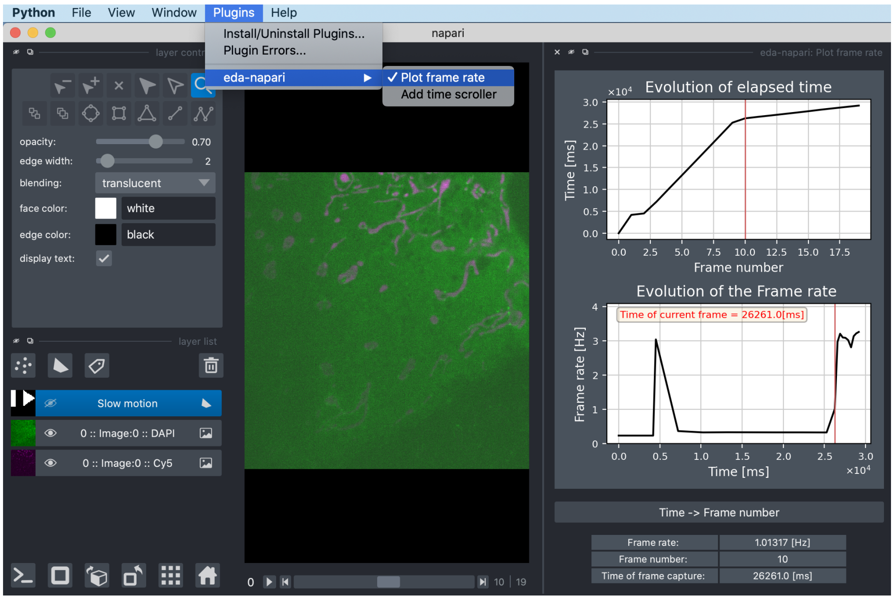

Figure 6: Opening eda-napari Frame rate widget in napari, with mitochondria double channel image sample.

The second widget, the time scroll bar, can also be activated and docked under the main window (Figure 7). The same mitochondria are again visualised but at a different time frame, hence this time a slow motion icon is visible. The x-axes of the frame plot plot has been switched to frame number by clicking on a button.

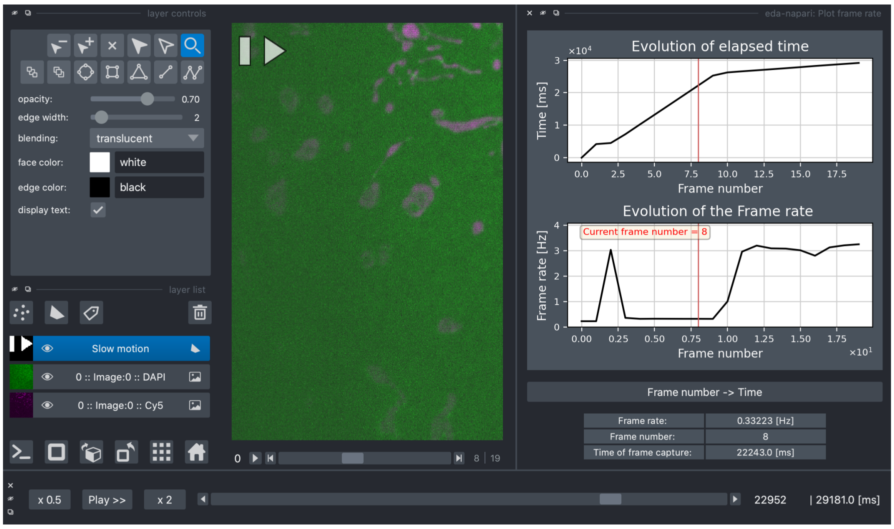

Figure 7: Frame rate widget and time scroller widget activated with mitochondria time-lapse image and during slow image acquisition.

The final viewer, figure 8, uses test data captured with a periodic evolving frame rate. The regular intervals are clearly seen with the evolution of the side plots. The frame is again in the low frame rate domain, explaining why a slow motion icon appears.
The plugin, as a whole, visualises frame rates from tif files but can also animate linearly in time with the time scroll bar *play >>* button.

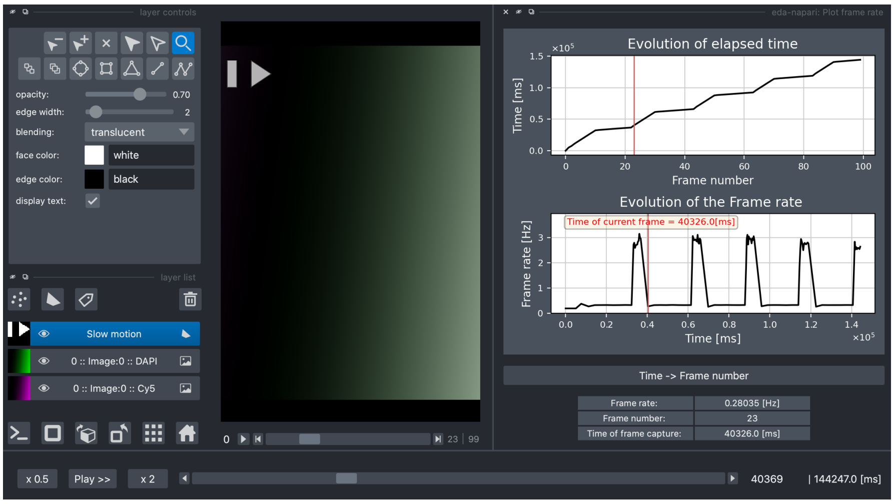

Figure 8: Napari viewer with loaded example data sample and periodic varying frame rate.

5.4 Limitations and Improvements

The main limitations are due to lengthy computations needed to update the plots. With frame widget activated, napari's scroll bar reactivity is slowed down. It can take up to 1 second for the stop button to stop the animation. The time scroll bar is however not effected. The plugin is overall faster, while using only one single widget.
The animation with the time scroll bar suffers in linear time stepping, when the frame rate widget is used. Between each time tick, many operations need to be executed. During fast animations, the updating of the plots can cause visible halting and disturb the animation.
The plugin is also limited in the format of readable files. More specifically, it only accepts tif files with metadata in s or ms. Additionally, the metadata needs to be structured with specific keywords.
By default, the plugin takes the source of the channel 0 image data and thus assumes that images from other channels have the same acquisition time. This could also be a potential limitation.
General improvements in the plugins reactivity could smoothen the user experience. Depending on the types of microscope data read, additional reading capabilities could also be desired. Currently the plugin has not been published on napari-hub, therefore it can't be downloaded directly on napari but needs to be installed following section 5.7. For publishing in napari, more intensive tests and documentation would be necessary.

5.5 Automatic testing

The plugin has been mostly tested with non automatic tests, but ideally intensive automated tests should be implemented. Several automated loading tests have, nevertheless, been implemented with pytest. Pytest operates with fixtures that create an object instance. Functions can then be created to test the instances and simulate user interaction or check for predicted behaviour.
For pytest to evaluate the plugin, it needs access to certain packages. This is why in the *setup.cfg* file of the project there are extra option requirements that are specified for testing. To run the tests, one must simply navigate into the directory eda-napari and run in the terminal *pytest*. Pytest detects any test file starting by the name *test_* in the eda-napari folder. For a reader wanting to add tests to the plugin, it can be useful to remember that a QTimer postpones the

initialisation of frame rate widget, while data is uploaded. This could induce test failures, if the test does not wait for the widget to load the data.

5.6 Issues

A program, especially one with widgets, must be adapted to all sort of different event loops, due to different usages. This is why, for example, the initialisation is programmed with a *try* and *except* statement. This avoids the programme crashing due to unreadable image formats. On the **github** for the project, all current issues are listed and commented. The current detected issues consist of one unwanted interaction between the two sliders and a small loading inconsistency.

5.7 Installing the eda napari plugin

This section will be helpful for downloading eda-napari. Eda-napari is not available on napari-hub and must therefore be downloaded manually. The following steps explain the process.

1. First, it is recommended to set up a virtual environment.

2. From **github** download or clone the whole eda-napari folder. All the files must be placed in a folder named "eda-napari" such that the directory structure (Annex 7.1) is respected.

3. Activate the virtual environment.

4. Pip install a backend for Qt. PyQt5 or PySide2 is recommended. The plugin does not specify the backend, but lets the user choose.

   ```
   pip install PyQt5 #or PySide2
   ```

5. Still within the virtual environment, navigate into the eda-napari directory and install.

   ```
   pip install -e . # "-e" for editting mode
   ```

6. Napari can now be opened from the terminal and will detect eda-napari.

   ```
   napari
   ```

6 Conclusion

This project illustrates how to create an elaborate npe2 napari plugin, starting from napari's plugin tutorial. The basics of the Qt frame work and napari's python classes can be better understood by following the programming of the plugin's widgets. The plugin is functional and can display, as planned, the frame rate of different time-lapse images. The program can also animate the time-lapse with a separate scroll bar. Theses widgets facilitate the visualisation of event driven acquisition with several means and user interactions. The code could also be used as a template for creating additional npe2 features that handle image metadata.
However, the plugin does still need intensive testing to ensure no bugs can occur and minor computational improvements could be beneficial for the user experience.

References

[1] Perkel, J. M. (2021). Python power-up: New image tool visualizes complex data. Nature, 600(7888), 347–348. https://doi.org/10.1038/d41586-021-03628-7

[2] Mahecic, D., Stepp, W. L., Zhang, C., Griffié, J., Weigert, M., & Manley, S. (2021). Event-driven acquisition for content-enriched microscopy [Preprint]. Cell Biology. https://doi.org/10.1101/2021.10.04.463102

[3] napari contributors (2022). napari: a multi-dimensional image viewer for python. doi:10.5281/zenodo.3555620

[4] napari contributors (2022). napari tutorial: Your First Plugin. https://napari.org/plugins/first_plugin.html

[5] Allen Institute for Cell Science. (2021). AICSImageIO. https://github.com/AllenCellModeling/aicsimageio

[6] Christopher Nauroth-Kress. (2021), A napari Plugin for visualisaton of pixel values over time (t + 3D, t+ 2D) as graphs. https://github.com/ch-n/napari-time_series_plotter

[7] Haase, Robert & Fazeli, Elnaz & Legland, David & Doube, Michael & Culley, Siân & Belevich, Ilya & Jokitalo, Eija & Schorb, Martin & Klemm, Anna & Tischer, Christian. (2022). A Hitchhiker‘s Guide through the Bio-image Analysis Software Universe.

[8] napari workshops. (2022). Napari Plugin Accelerator Grant workshop series. https://chanzuckerberg.github.io/napari-plugin-accel-workshops/

7 Annexes

7.1 Project structure

```
eda-napari
├── eda_napari
│   ├── __init__.py
│   ├── _widget.py
│   ├── plot_stylesheet.mplstyle
│   ├── q_label_stylesheet.qss
│   └── napari.yaml
├── pyproject.toml
├── setup.cfg
├── tests
│   └── test_widget.py
├── debug.py
├── images
│   └── example_image.tif
└── LICENSE
```

Figure 9: Project structure

7.2 The main widget python code

The following code is from *_widget.py* and is accessible also on the **github**.

1. Imports

```python
import os
import traceback
import matplotlib.style as style
import matplotlib.pyplot as plt
from matplotlib.backends.backend_qtagg import FigureCanvas  #Qt binding not specified (PyQt5, PyQt4...)
from qtpy.QtWidgets import QWidget, QVBoxLayout, QHBoxLayout, QPushButton, QLabel, QGridLayout, QScrollBar
from typing import Union
import qtpy
from qtpy.QtCore import Qt, QTimer
import magicgui
from magicgui import magic_factory
import numpy as np
import math

import napari
import tifffile
import xmltodict

from skimage.filters import threshold_otsu

#Stylesheets
############
style.use(str(os.path.dirname(__file__))+'/plot_stylesheet.mplstyle') #get path of parent directory of script since plot_stylesheet is in the same directory
stylesheet = open(str(os.path.dirname(__file__))+'/q_label_stylesheet.qss',"r")
label_style = stylesheet.read()
```

2. Frame rate widget

Code 3: Frame rate widget

```python
#Widgets
############

#Union is a type: it forms the math union.
#It means the widget could be a magicgui widget or a qtpy widget.
Widget = Union["magicgui.widgets.Widget", "qtpy.QtWidgets.QWidget"]

class Frame_rate_Widget(QWidget):
    """The Frame rate plotter widget.

      This widget is a object inheriting from QWidget. Defining Frame_rate_Widget in the
    manifest.
      "napari.yaml" allows Napari to reconise it's existance and display it in the plugin.
    Upon it's creation the __init__ function ensures
      the execution of specific functions and plots the frame rates of an OME.tiff file
    inserted in the napari viewer.
    """

    def __init__(self, napari_viewer):
        """Constructor of the Frame_rate_Widget.

        This constructor initialises two blank canvases,the class's viewer to the napari viewer
      and connects events to functions to allow dynamic plots.
        A newly inserted file or modification of slider position in napari causes the plot to
      update. The update after an insertion waits a certain Twait time before executing to
      ensure
        napari has time to fully import all the layers. The wait is implemented with a QTimer.
        """
        super().__init__()
        self._viewer = napari_viewer
        self.image_path=None
        self.time_data=None
        self.frame_rate_data=None
        self.channel=0
        self.frame_x_axis_time=True
        self.setStyleSheet(label_style)


        #main plot widget
        self.layout=QVBoxLayout(self)
        self.grid_layout = QGridLayout()

        self.button_txt=('Time -> Frame number')
        self.button_axis_change=QPushButton(self.button_txt)
        self.button_axis_change.clicked.connect(self.change_axis)

        self.grid_layout.setAlignment(Qt.AlignHCenter)
        self.grid_layout.setSpacing(2)
        self.grid_layout.setColumnMinimumWidth(0, 86)
        self.grid_layout.setColumnStretch(1, 1)

        self.frame_number_label = QLabel("Frame number: ")
        self.grid_layout.addWidget(self.frame_number_label, 1, 0)
        self.frame_number_value=QLabel("-")
        self.grid_layout.addWidget(self.frame_number_value, 1, 1)

        self.frame_time_label = QLabel("Time of frame capture: ")
        self.grid_layout.addWidget(self.frame_time_label, 2, 0)
        self.frame_time_value=QLabel("-")
        self.grid_layout.addWidget(self.frame_time_value, 2, 1)

        self.frame_rate_label = QLabel("Frame rate: ")
        self.grid_layout.addWidget(self.frame_rate_label, 0, 0)
        self.frame_rate_value=QLabel("-")
        self.grid_layout.addWidget(self.frame_rate_value, 0, 1)

        self._init_mpl_widgets()
        self.layout.addWidget(self.button_axis_change)
        self.layout.addLayout(self.grid_layout)
        #QTimer, this Q timer will be used to load data after a certain wait time
        self.Twait=2500
```

```
        self.timer=QTimer()
        self.timer.setInterval(self.Twait)
        self.timer.setSingleShot(True)#timer only runs once
        self.timer.timeout.connect(self.init_data)

        #events
        self._viewer.layers.events.inserted.connect(self.init_after_timer)
        self._viewer.layers.events.removed.connect(self.update_widget)

    def _init_mpl_widgets(self):
        """Method to initialise two matplotlib figure canvases with a basic layout and title.

        This method generates a matplotlib FigureCanvas and populates it with a
        matplotlib.pyplot.figure. The canvas is added to the QWidget Layout afterwards.
        """
        self.fig = plt.figure()
        self.canvas = FigureCanvas(self.fig)
        self.ax = self.fig.add_subplot(211)
        self.ax2 = self.fig.add_subplot(212)
        self.layout.addWidget(self.canvas)
        self.init_data() #try to init data if it exists

    def init_data(self):
        try:
            if self.image_path != self._viewer.layers[0].source.path : #update data if new source is added
                self.image_path = self._viewer.layers[0].source.path
                self.time_data=get_times(self)#init times of initial image
                self.frame_rate_data=self.get_frame_rate()#init frame rate of initial image
                self.plot_frame_data()
                self.create_SlowMo_icon()
                self.slow_mo()

                self._viewer.dims.events.current_step.connect(self.plot_slider_position)
                self._viewer.dims.events.current_step.connect(self.update_slowMo_icon)

        except(IndexError,AttributeError): # if no image is placed yet then Errors would occur when the source is retrieved
            print('Meta data not readable')
        except KeyError:
            print('Dictionary access in get_times fails. Tif file does not have the adapted keys.')
            traceback.print_exc()#prints the info of error

    def init_after_timer(self):
        self.timer.start(self.Twait) #restarts the timer with a timeout of Twait ms

    def plot_times(self):
        self.ax.clear() #clear plot before plotting
        self.ax.set_ylabel('Time [ms]')
        self.ax.set_xlabel('Frame number')
        self.ax.set_title('Evolution of elapsed time')
        self.ax.plot(self.time_data)
        self.ax.ticklabel_format(axis='y', style='scientific', scilimits=(0,0), useMathText='True')
        self.line_1=self.ax.axvline(self._viewer.dims.current_step[0],0,1,linewidth=1, color='indianred')#initilaise a vertical line
        self.fig.canvas.draw()

    def get_frame_rate(self,unit_frame_r='Hz'):
        """ Method that returns frame rate.

        Input: unit of frame rate: [kHz] or [Hz]
        Output: Vector of frame rates in [kHz] or [Hz]

        The frame rate is calculate for each frame. Since the system is discrete, it must be approximated.
        For the first frame, the frame rate is approximated with the second frame time.For the other frames, the rate is calculated with the previous frame time."""

        if unit_frame_r=='Hz':
            conversion_factor =1000     #convert to kHz to Hz
        elif unit_frame_r=='kHz':
            conversion_factor=1  #inital data is in kHz since time is diplayed in ms
        else:
```

```
            print('unit of frame rate not reconised: please use Hz or kHz')

        N_frames= len(self.time_data)
        frame_rate=[conversion_factor/abs(self.time_data[1]-self.time_data[0])]#the first frame rate
        for i in range(1,N_frames):
            frame_rate.append(conversion_factor/abs(self.time_data[i]-self.time_data[i-1]))
        return frame_rate

    def plot_frame_rate(self,unit_frame_r='Hz'):

        self.ax2.clear() #clear plot before plotting
        self.ax2.set_ylabel('Frame rate ['+unit_frame_r+']')
        self.ax2.ticklabel_format(axis='x', style='scientific', scilimits=(0,0), useMathText='True')
        self.ax2.set_title('Evolution of the Frame rate ')
        if self.frame_x_axis_time:
            self.ax2.set_xlabel('Time [ms]')
            self.ax2.plot(self.time_data,self.frame_rate_data)
            vline_pos=self.time_data[self._viewer.dims.current_step[0]]
            txt_text_box='Time of current frame = '+str(self.time_data[self._viewer.dims.current_step[0]])+'[ms]'
        else:
            self.ax2.set_xlabel('Frame number')
            self.ax2.plot(self.frame_rate_data)
            vline_pos=self._viewer.dims.current_step[0]
            txt_text_box='Current frame number = '+str(self._viewer.dims.current_step[0])

        self.ax2.set_ylim(0,self.ax2.get_ylim()[1]*1.2) #increase plot for text space
        self.line_2=self.ax2.axvline(vline_pos,0,1,linewidth=1, color='indianred')#initialise a vertical line
        props = dict(boxstyle='round', facecolor='wheat', alpha=0.3)
        self.text_box=self.ax2.text(0.05, 0.95, txt_text_box, transform=self.ax2.transAxes, fontsize=8, verticalalignment='top', bbox=props,color='red')
        self.fig.canvas.draw()

    def plot_frame_data(self):
        self.plot_times()
        self.plot_frame_rate()
        current_frame=self._viewer.dims.current_step[0]
        if self.frame_x_axis_time:
            self.line_2.set_xdata(self.time_data[current_frame])#update according to x axis (time or frame)
        else:
            self.line_2.set_xdata(current_frame)

        self.frame_rate_value.setText(str(np.around(self.frame_rate_data[current_frame],5)) + ' [Hz]')#init Qlabels
        self.frame_number_value.setText(str(current_frame))
        self.frame_time_value.setText(str(self.time_data[current_frame])+' [ms]')

    def plot_slider_position(self,event): #event information stored in "event"
        """ Method plots and updates slider position on the canvas.

        Input: event of current_step from slider
        Output: -
        After moving the slider on napari viewer, this function is called to update the vertical lines. The vertical
        lines show the frame rate and capture time of the current image disaplayed in the napari viewer."""
        current_frame=event.source.current_step[0]
        self.line_1.set_xdata(current_frame) #update line
        if self.frame_x_axis_time:
            self.line_2.set_xdata(self.time_data[current_frame])#update according to x axis (time or frame)
            self.text_box.set_text('Time of current frame = '+str(self.time_data[current_frame])+'[ms]')
        else:
            self.line_2.set_xdata(current_frame)
            self.text_box.set_text('Current frame number = '+str(self._viewer.dims.current_step[0]))

        self.frame_rate_value.setText(str(np.around(self.frame_rate_data[current_frame],5)) + ' [Hz]')#update Qlabels
        self.frame_number_value.setText(str(current_frame))
```

```
        self.frame_time_value.setText(str(self.time_data[current_frame])+' [ms]')
        self.fig.canvas.draw()

    def change_axis(self):
        self.frame_x_axis_time=not self.frame_x_axis_time
        if self.frame_x_axis_time:
            button_txt='Time -> Frame number'
            self.current_frame_txt='Current frame number = '

        else:
            button_txt='Frame number -> Time'
        self.button_axis_change.setText(button_txt)
        self.plot_frame_rate()

    def create_SlowMo_icon(self):
        triangle=np.array([[20, 60], [60, 60], [40, 90]])
        rectangle=np.array([[20, 40],[60, 40],[60,25],[20,25]])
        polygon=[triangle,rectangle]
        self._viewer.add_shapes(polygon, shape_type='polygon', face_color='white',edge_width=2,
                                edge_color='black', name='Slow motion')
        self.slow_mo_channel=self._viewer.layers.index('Slow motion')
        self._viewer.layers[self.slow_mo_channel].visible=False #init to invisible

    def update_slowMo_icon(self,event):
        if(self.slow_mo_array[event.source.current_step[0]]):
            self._viewer.layers[self.slow_mo_channel].visible=True
        else:
            self._viewer.layers[self.slow_mo_channel].visible=False

    def slow_mo(self):
        size=len(self.time_data)
        self.slow_mo_array=np.empty(size)
        thresh=threshold_otsu(np.array(self.frame_rate_data))#threshold to determine weather slow motion or fast speed
        for i in range(0,size):
            if self.frame_rate_data[i]>=thresh:
                self.slow_mo_array[i]=False
            else:
                self.slow_mo_array[i]=True

    def update_widget(self,event):
        """ Method updates the widget in case of layer deletion.

        Input: event created my deleted layer.
        Output: -
        If slow motion is removed then the napari viewer is disconnected to the slow motion shape.
        If all layers are removed the dock plugin is removed.
        """
        if not ('Slow motion ' in event.source):
            self._viewer.dims.events.current_step.disconnect(self.update_slowMo_icon)
        if len(event.source)==0:
            try:
                self._viewer.window.remove_dock_widget(self)
                self.image_path=None
            except:
                print('Dock already deleted')
```

3. Time scroller widget

Code 4: Time scroller widget

```
class Time_scroller_widget(QWidget):
    """Time_scroller_widget class is a widget that creates a time scroll bar. The scroll bar allows to animate stacks of
    images linearly with time. Similary to the napari scroll bar, it has a play, stop, next and previous button."""

    def __init__(self, napari_viewer):
        """Constructor of the Time_scroller_widget.

        This constructor initialises the button widgets and scroll bar widget in a QHBoxlayout. It also initialise the time of the frames
        from the image data available. Some signals are also defined to allow interaction and automatic updates between the current layer and
```

```python
        the Time_scroller_widget widget. A Qtimer is defined to controll the animation.
        """
        super().__init__()
        self._viewer = napari_viewer
        self.image_path=None #folder path of the image data
        self.time_data=None
        self.channel=0
        self.number_frames=None
        self.show_time=50#animation default display time ms
        self.time_interval=None # time  of the discretised time interval [ms]
        self.interval_frames_index=[]#discretised time interval filled with an index the the different frames.
        self.step=1
        self.critical_ms=160
        self.timer=QTimer(self)
        self.timer.timeout.connect(self.play_step)

        self.layout=QHBoxLayout(self)
        self.setMinimumWidth(500)
        self.setMaximumHeight(100)

        self.create_bottom_dock_button()
        self.create_slow_down()
        self.create_play_button()
        self.create_speed_up()
        self.create_time_scroller()
        self.create_axis_label()

        self.layout.addWidget(self.bottom_dock_button)
        self.layout.addWidget(self.slow_down_button)
        self.layout.addWidget(self.play_button)
        self.layout.addWidget(self.speed_up_button)
        self.layout.addWidget(self.time_scroller)
        self.layout.addWidget(self.axis_label1)
        self.layout.addWidget(self.axis_label2)

        self.init_data()
        self._viewer.layers.events.inserted.connect(self.init_data) #init data when layer is inserted

    def create_bottom_dock_button(self):
        self.bottom_dock_button=QPushButton('        ')
        self.bottom_dock_button.setMaximumWidth(35)

    def move_dock_to_bottom(self):
        self.parentWidget().parentWidget().addDockWidget(Qt.BottomDockWidgetArea, self.parentWidget()) # init position in QDockWidget (parent) to bottom in QWindow
        self.bottom_dock_button.deleteLater()

    def create_slow_down(self):
        self.slow_down_button=QPushButton('x 0.5')
        self.slow_down_button.setMaximumWidth(50)

    def slow_animation(self):
        if self.step == 1:
            self.show_time = self.show_time*2
            if self.play_button_txt=='Stop': #if program is playing
                self.timer.stop()
                self.timer.start(self.show_time) #set new show_time
        else:
            self.step=self.step/2
    def create_speed_up(self):
        self.speed_up_button=QPushButton('x 2')
        self.speed_up_button.setMaximumWidth(50)

    def speed_animation(self):
        if self.show_time >=self.critical_ms:
            self.show_time = self.show_time*0.5
            if self.play_button_txt=='Stop':
                self.timer.stop()
                self.timer.start(self.show_time)
        else:
            self.step=self.step*2

    def create_time_scroller(self):
```

```python
        self.time_scroller= QScrollBar(Qt.Horizontal)
        self.time_scroller.setMinimum(0)
        self.time_scroller.setSingleStep(1)
        self.time_scroller.setMinimumWidth(150)

    def create_play_button(self):
        self.play_button_txt='Play >>'
        self.play_button=QPushButton(self.play_button_txt)
        self.play_button.setMaximumWidth(60)

    def create_axis_label(self):
        self.axis_label1=QLabel('End time')
        self.axis_label2=QLabel('Current time')

    def init_data(self):
        """This method initialises all the additonal data after an image stack is available and readable.
        """
        try:
            if self.image_path!=self._viewer.layers[0].source.path: #Only inits data if the layer is new
                    self.image_path = self._viewer.layers[0].source.path
                    self.time_data=get_times(self)#init times of image stack
                    self.number_frames=len(self.time_data)
                    self.init_time_interval()
                    self.set_frames_index()
                    #time scroller
                    self.time_scroller.setMaximum(len(self.interval_frames_index)-1)
                    idx=self.interval_frames_index.index(self._viewer.dims.current_step[0])
                    self.time_scroller.setValue(idx) #set init position of scroller
                    #init label2
                    self.axis_label2.setText('| '+str(self.time_data[-1])+' [ms]')
                    width2 = self.axis_label2.fontMetrics().boundingRect(self.axis_label2.text()).width() #max width of text
                    self.axis_label2.setFixedWidth(int(1.1*width2))
                    #init Label1
                    self.axis_label1.setText(str(self.time_scroller.value()*self.time_interval))
                    self.axis_label1.setFixedWidth(int(0.7*width2))

                    #events
                    self.bottom_dock_button.clicked.connect(self.move_dock_to_bottom)
                    self.slow_down_button.clicked.connect(self.slow_animation)
                    self.play_button.clicked.connect(self.play)
                    self.speed_up_button.clicked.connect(self.speed_animation)
                    self._viewer.dims.events.current_step.connect(self.update_scroller_from_dims)#link window srolle to time srolle
                    self.time_scroller.valueChanged.connect(self.update_scroller_from_scroller) #link
                    self._viewer.layers.events.removed.connect(self.update_widget)
                    self.data_is_avable=True

        except (IndexError,AttributeError): # if no image is found then an index Error would occur
              print('Meta data not readable')
        except KeyError:
            print('Dictionary access in get_times fails. Tif file does not have the adapted keys.')
            traceback.print_exc()#prints the info of error

    def init_time_interval(self):
        """This method sets the time discretisation interval of the system, for the animation and scroll bar.
        self.time_interval is initiliased as the 1/4 of the minimum time between to images frames. This creates a revelant discretisation of time with
        respect to the frame rates.
        """
        diff=[]
        for i in range(1,self.number_frames):
            diff.append(self.time_data[i]-self.time_data[i-1])
        min_diff=min(diff)
        self.time_interval = math.floor(min_diff/4)#this makes sure a relevant discretisation of time is made for the animation

    def set_frames_index(self):
        """This method sets self.interval_frames_index with image frames numbers. Each index
```

```
corresponds the appropriate image frame
        that should be displayed in the discretized time.
        """
        frame_index=0
        t=0
        self.interval_frames_index=[0]
        while frame_index < self.number_frames-1:
            t+=self.time_interval
            if self.time_data[frame_index+1] < t:
                frame_index+=1
            self.interval_frames_index.append(frame_index)

    def play_step(self):
        """This method advances the time of 1 time interval and takes care of updating the current displayed frame if necessary.
        """
        if self._viewer.dims.current_step[0]== self.number_frames-1:
            self._viewer.dims.set_current_step(0, 0)#restart at frame 0
            self.time_scroller.setValue(0)
        else:
            self.time_scroller.setValue(self.time_scroller.value()+self.step)
            if self.interval_frames_index[self.time_scroller.value()] != self._viewer.dims.current_step[0]: #update viewer
                self._viewer.dims.set_current_step(0, self.interval_frames_index[self.time_scroller.value()])

    def play(self): # play and stop method
        if self.play_button_txt =='Play >>':
            self.timer.start(self.show_time)
            self.play_button_txt = 'Stop'
            self.play_button.setText(self.play_button_txt)
            self.play_button.setMaximumWidth(80)

        else: #stop
            self.timer.stop()
            self.play_button_txt = 'Play >>'
            self.play_button.setText(self.play_button_txt)

    def update_scroller_from_dims(self):
        idx=self.interval_frames_index.index(self._viewer.dims.current_step[0])
        self.time_scroller.valueChanged.disconnect(self.update_scroller_from_scroller)#avoid double calling
        self.time_scroller.setValue(idx)
        if self._viewer.dims.current_step[0]==self.number_frames-1:
            self.axis_label1.setText(str(self.time_data[-1]))
        else:
            self.axis_label1.setText(str(self.time_scroller.value()*self.time_interval))
        self.time_scroller.valueChanged.connect(self.update_scroller_from_scroller) #reconnect

    def update_scroller_from_scroller(self):
        if self.interval_frames_index[self.time_scroller.value()] != self._viewer.dims.current_step[0]: #update viewer #one
            self._viewer.dims.events.current_step.disconnect(self.update_scroller_from_dims) #avoid double calling
            self._viewer.dims.set_current_step(0, self.interval_frames_index[self.time_scroller.value()])
            self._viewer.dims.events.current_step.connect(self.update_scroller_from_dims) #reconnect

        if self._viewer.dims.current_step[0]==self.number_frames-1:
            self.axis_label1.setText(str(self.time_data[-1]))
        else:
            self.axis_label1.setText(str(self.time_scroller.value()*self.time_interval))

    def update_widget(self,event):
        if len(event.source)==0:
            try:
                self._viewer.window.remove_dock_widget(self)
                self.image_path=None
            except:
                print('Dock already deleted')
```

4. Read tiff file metadata

To read the meta data from tiff files, the following function has been implemented. The meta data is read with TiffFile and the xmltodict package. With the correct dictionary keys the time data of the file is then accessed. This function is not a class method and is accessible from both widgets.

Code 5: Get_times metadata access function

```python
def get_times(widget):
    """Method that gets the capture time from the metadata.

    Input:-
    Output: Vector of time metadata [ms] of images found at given image path and channel.

    The times of each image stack from a ome.tif file is read in [ms] or [s] and then returned in [ms].
    The times are taken from a given channel. The data can only be read from an ome.tif file. The Offset
    from time t=0 subrtracted to the times before it is returned.
    The following code is inspired from the solution for reading tiff files metadata from Willi Stepp.
    """
    times=[]
    with tifffile.TiffFile(widget.image_path) as tif:
        XML_metadata= tif.ome_metadata #returns a reference to a function that accesses the metadata as a OME XML file
        dict_metadata=xmltodict.parse(XML_metadata) #converts the xml to a dictionary to be readable
        num_pages=len(tif.pages) #the number of images stacked
        for frame in range(0,num_pages):
            #time should be in either s or ms
            if float(dict_metadata['OME']['Image']['Pixels']['Plane'][frame]['@TheC'])==widget.channel: #checks if correct channel
                frame_time_unit=dict_metadata['OME']['Image']['Pixels']['Plane'][frame]['@DeltaTUnit']
                if frame_time_unit== 's' :
                    convert_unit_to_ms=1000
                    times.append(convert_unit_to_ms*float(dict_metadata['OME']['Image']['Pixels']['Plane'][frame]['@DeltaT']))
                elif frame_time_unit == 'ms':
                    convert_unit_to_ms=1
                    times.append(convert_unit_to_ms*float(dict_metadata['OME']['Image']['Pixels']['Plane'][frame]['@DeltaT']))
                else:
                    print('Time units not in ms or s but in '+ frame_time_unit+'. A conversion to ms or s must be done.')

    times = [x - times[0] for x in times] #remove any offset from time=0
    return times
```

7.3 Debugging

The plugin was coded with the python editor Virtual Studio Code. VS code allows connecting with git and debugging. Setting up a *debug.py* file was useful for gaining time during tests. The debug file opens up napari with the debugger and loads automatically an image. With the debugger the plugin runs much slower than directly being opened in the terminal. All the variables and plugin objects can be accessed any time during the code execution, which helps for locating errors.