Experiment Feedback

ZEN 3.5 (blue edition)

Carl Zeiss Microscopy GmbH
Carl-Zeiss-Promenade 10
07745 Jena
Germany

microscopy@zeiss.com
www.zeiss.com/microscopy

Carl Zeiss Microscopy GmbH
ZEISS Group
Kistlerhofstr. 75 81379 München
Germany

Effective from: 09 / 2021

Contents

1 ZEN 3.5 (blue edition) - Experiment Feedback

This tutorial contains general information about how to use and set up feedback experiments and some test examples for typical use cases. All examples can be tested on a simulated system without any hardware. **Experiment Feedback** is part of the ZEN module **Advanced Processing & Analysis**. In order to use **Experiment Feedback**, also a license for the **Image Analysis** and **Advanced Processing** modules of ZEN 3.5 (blue edition) is required.

1.1 Definition

Experiment feedback (also known as conditional or adaptive experiments) allows you to define specific rules and actions to be performed during the acquisition of an experiment. It is possible to change the course of an experiment depending on the current system status or the nature of the acquired data during acquisition. Moreover, it is possible to integrate certain tasks like data logging or starting of an external application, directly into the ZEN experiment. A typical use-case is to connect the image acquisition with automated image analysis.

1.2 Key Features

- Create smart experiments and modify the acquisition on-the-fly based on online image analysis, hardware changes or external inputs (e.g. TTL signals).
- The adaptive acquisition engine allows modifying running experiments according to the rules defined inside the feedback script.
- The feedback script uses ZEN commands in combination with the Python programming language
- The feedback script gives access to the current system status and results from the online image analysis during runtime of an experiment.
- Data Logging or starting an external application (e.g. Python, Fiji, MATLAB, ...), directly from within the imaging experiment is possible.

1.3 General Workflow

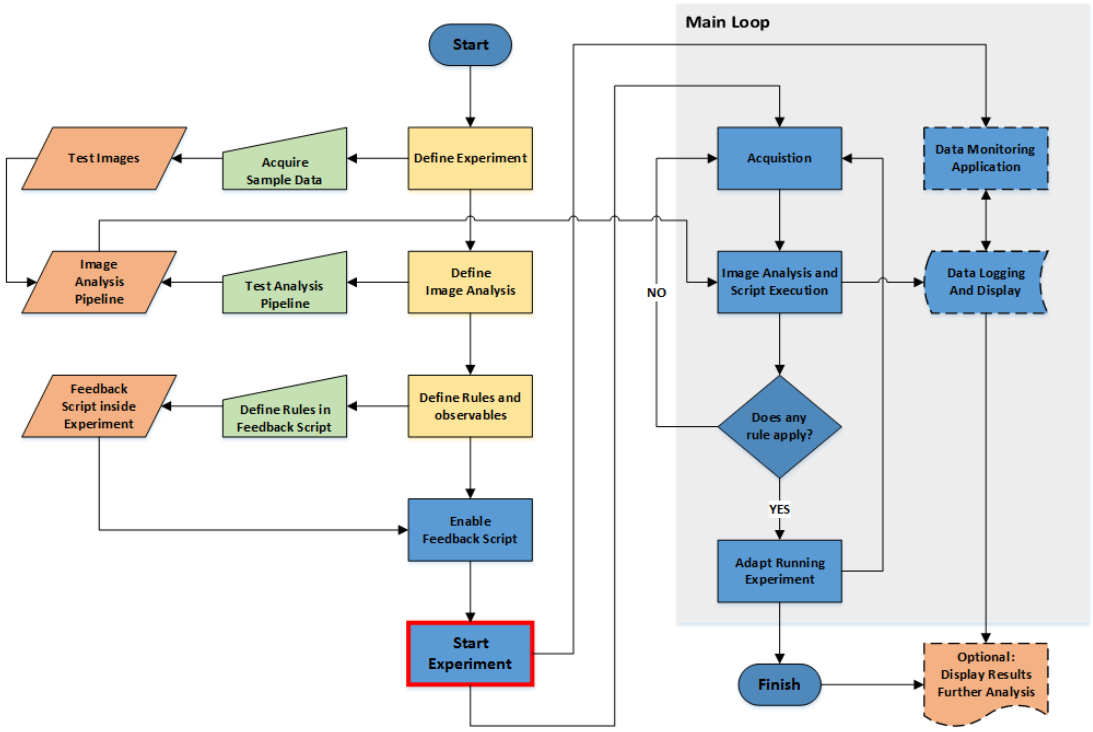

Figure 1: General workflow how to set up and run (main loop) a feedback experiment.

In Fig. 1 you can see the typical workflow of an **Experiment Feedback** experiment and the most important steps involved. The yellow boxes **Define Experiment**, **Define Image Analysis** and **Define Rules and Observables** show the most important steps for setting up a feedback experiment.

- **Define Experiment:** First you need to set up and configure the actual image acquisition experiment to obtain the desired image data (e.g. time lapse, Z-stack, multichannel, tile acquisition, ...). Once the setup of the acquisition is completed, acquire sample data which will be used in the following step to setup and test the image analysis pipeline.

- **Define Image Analysis:** Setup an image analysis pipeline (you can use the image analysis wizard or for advanced topics also an OAD macro that creates a *.czias file) for the use inside the feedback script if required. Only parameters specified in the image analysis pipeline can later be accessed from within the experiment feedback script. Test the image analysis pipeline to ensure the results of the image analysis are meaningful.

- **Define Rules and Observables:** In this step you need to define how the script should actually work. That means you need to define which parameters are observed and how the experiment should react when a certain event occurs.

After the setup is complete you can start the **Experiment Feedback** experiment and watch the output. The general concept behind this workflow can be described as a loop, which is the actual acquisition. For every event, e.g. when a new image has been acquired, the script will be executed. The rules will be checked and if required, certain tasks will be carried out. Additionally it is possible to log data into a text file and/or start an external application at any time point during the experiment.

1.4 Adaptive Acquisition Engine

The script itself is only executed if a parameter has changed. The script run will be only triggered, if an observable has changed or is used inside the script. Typically this can be:

- A new image has been acquired
- The XYZ position has changed
- The status of a trigger port has been altered
- The settings for filter, objective, light source etc. have changed
- The incubation parameters have changed

If the parameters within the loop script do not change, the script will not be executed. (For advanced use cases it is possible to execute the script with the command ***RunLoopScript*** in conjunction with a timer).

2 The Experiment Feedback Script

In order to activate **Experiment Feedback** you need to go to **Tools** ▷ **Modules Manager** and activate the **Advanced Processing** checkbox.

After the successful activation, an additional checkbox **Experiment Feedback** will appear in the **Acquisition** tab of the ZEN 3.5 (blue edition) software. When you activate the **Experiment Feedback** checkbox, the respective tool will appear on the tab (Fig. 4).

Within the **Experiment Feedback** tool you have the following functionality:

- Open the script editor and edit the feedback script (via **Edit Feedback Script...**)
- Choose the script runtime conditions (**Free Run** or **Synchronized**)
- Allow additional loop script runs for hardware and environment variables (e.g. temperature changes etc.)
- Enable/disable script debugging

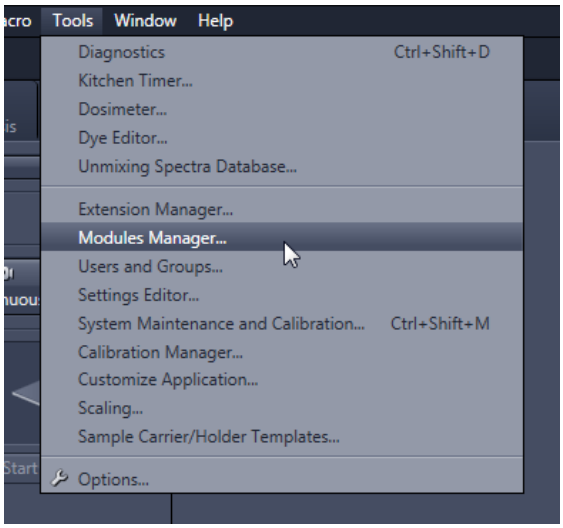

Figure 2: ZEN 3.5 (blue edition) **Tools** drop-down menu

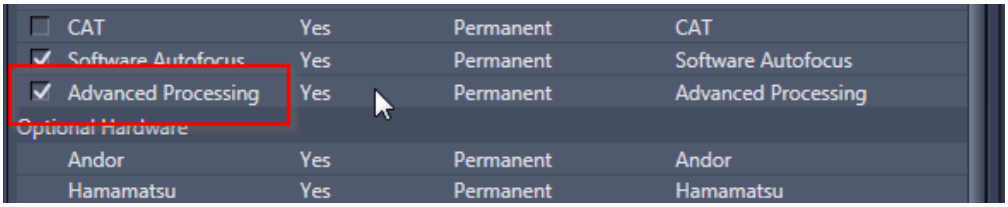

Figure 3: ZEN 3.5 (blue edition) Modules Manager

2.1 Synchronization Options

The script runtime conditions describe how the different steps of the experiment feedback script will be executed. By clicking on the blue buttons you can change the execution order of the script. You can choose between **free-run** and **synchronized** execution (Fig. 5 and 6). For the synchronized run there are three different modalities (Fig. 7):

Option 1 The script run is started after the acquisition together with the online image analysis. Therefore the script execution, image analysis and writing of the image sub-block to the hard drive are not in sync.

Option 2 The online image analysis is started after the acquisition and only when the image analysis is finished the next script run will be triggered. This guarantees that all analysis results exist and can be used in the feedback script. After the analysis is finished the image sub-block will be written to disk.

Option 3 The online image analysis is started after the acquisition, the image data are written to disk and only when all those tasks are finished the next script run will be triggered. Therefore it is guaranteed, that all analysis results exist and the image data is stored on disk before the next script run is triggered. This option is relevant

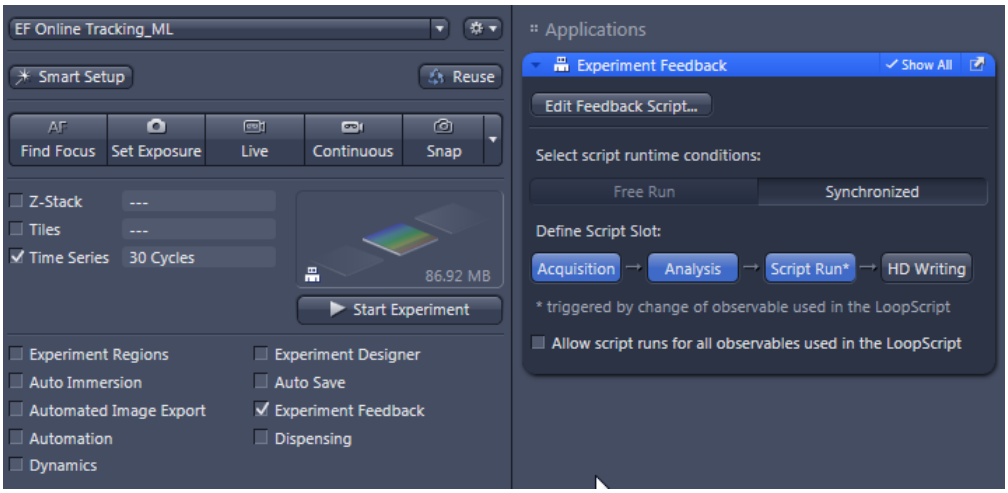

Figure 4: Experiment Feedback tool

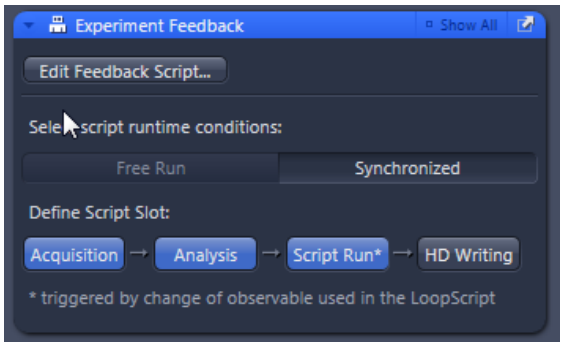

Figure 5: **Synchronized** acquisition

in case the script starts an external application to analyze the data.

2.2 The Feedback Script Editor

The feedback script is defined within the **Experiment Feedback Script Editor**. The editor is started from the **Experiment Feedback** tool via **Edit Feedback Script ...**. Here you can setup and modify the feedback script. The feedback script will be stored as part of the experiment file (***.czexp**). The **Experiment Feedback Script Editor** window consists of three sections: the **PreLoop Script**, the **Loop Script** and the **PostLoop Script**.

- The **PreLoop Script** is used to import modules and define functions or variables. This part is executed only once at the beginning of the feedback experiment.

- The **Loop Script** is used to react on results from the online image analysis (e.g. stop the acquisition when a defined number of cells have been counted) or to take action

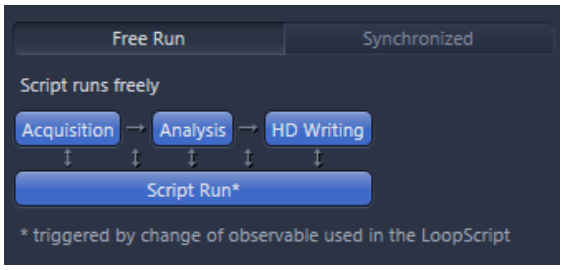

Figure 6: **Free Run** mode

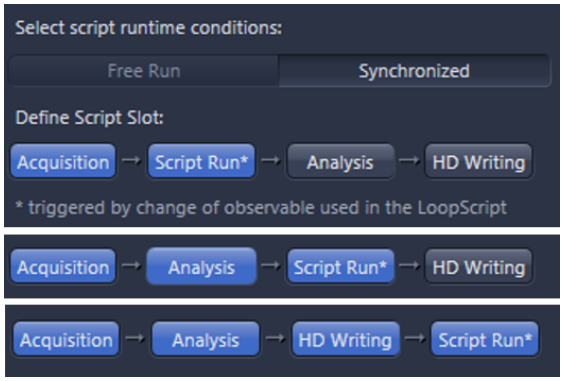

Figure 7: Differences between the three synchronization options

upon external signals. This allows to modify the experiment on-the-fly. This part is executed every time an observable used within the loop script changes.

- The **PostLoop Script** is used to define actions that are executed only once when the acquisition is finished, e.g. to write the data in a logfile.

INFO:

Note that the **Loop Script** will only be executed if it contains an observable that changes during the acquisition or if an observable was modified.

On the right side of the script editor in the **Tools** tab you can select observables and actions that are available for use inside the feedback script (Fig. 8).

If an image analysis is part of your feedback experiment, you can select a previously defined image analysis setting in the **Available Observables** section from the drop-down menu next to **Analysis**. When you click on the arrow next to the drop-down menu you can find a list of the features that were defined in this image analysis setting, e.g. number of cells detected (Fig. 9). If you select an image analysis setting and use one or more features in the feedback script, this image analysis setting will be executed for each acquired image.

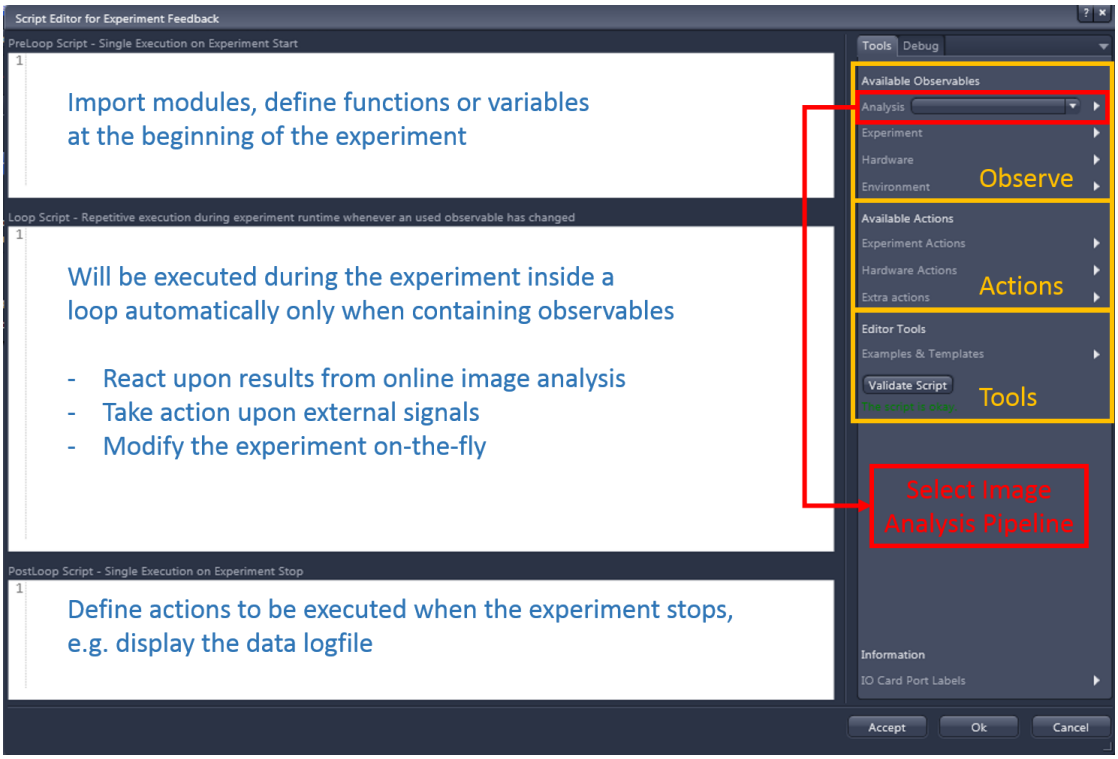

Figure 8: Experiment Feedback Script Editor

INFO:

Only features previously defined in the Image Analysis Wizard are available from within the Feedback Experiment.
If you make any changes within the image analysis wizard to an existing *.czias file you will need to reload the file in the feedback script editor in order to activate the changes in the feedback experiment.

Other observables that can be used inside the feedback script are **Experiment observables** (e.g. current tile index,...), **Hardware observables** (e.g. incubation or trigger parameters) or **Environment observables** (e.g. date, free disk space).

You will also find a list of **Available Actions** that can be be performed during the feedback experiment. The actions are grouped in **Experiment Actions** (e.g. Jump to next Region, pause/continue experiment,...), **Hardware Actions** (e.g. read/set stage position,...) and **Extra Actions** (e.g. execute an external program or append line to logfile,...).

You can add all observables and actions to your experiment feedback script via double-click or drag-and-drop. All observables and actions are also available via IntelliSense auto-completion starting with **ZENService** (Fig. 10).

At **Examples & Templates** you can find some code examples to start with.

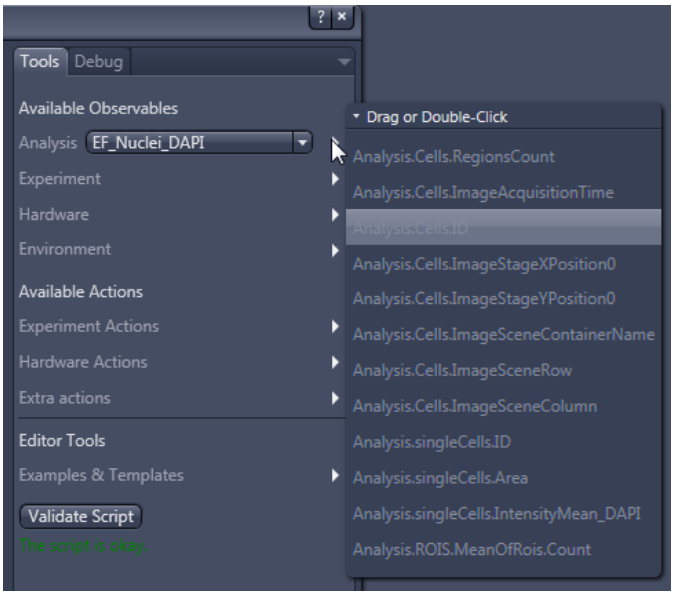

Figure 9: Features defined by the selected image analysis pipeline

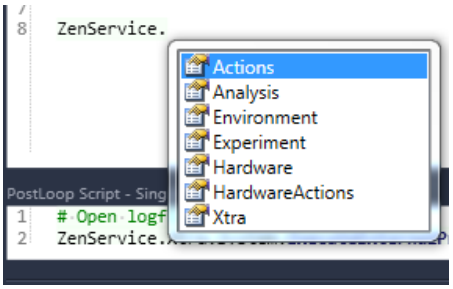

Figure 10: IntelliSense auto-completion for observables and actions

With the **Validate Script** button you can check if there are any syntax errors within the script. Note that when you click on **Accept** or **OK** the existing code will also be checked. That means you can close the script editor only if the code is free from syntax mistakes.

A list of available commands and a short explanation can be found in the ZEN 3.5 (blue edition) online help.

When debugging is enabled it is possible to observe observable changes, called/executed actions, warnings, exceptions as well as user defined output messages for debugging purposes (Fig. 11).

2.3 Advanced Use: Allow additional script runs

The checkbox **Allow additional script runs for all observables used in the LoopScript** on the **Experiment Feedback** tool (Fig. 4) allows also to trigger the main loop if other

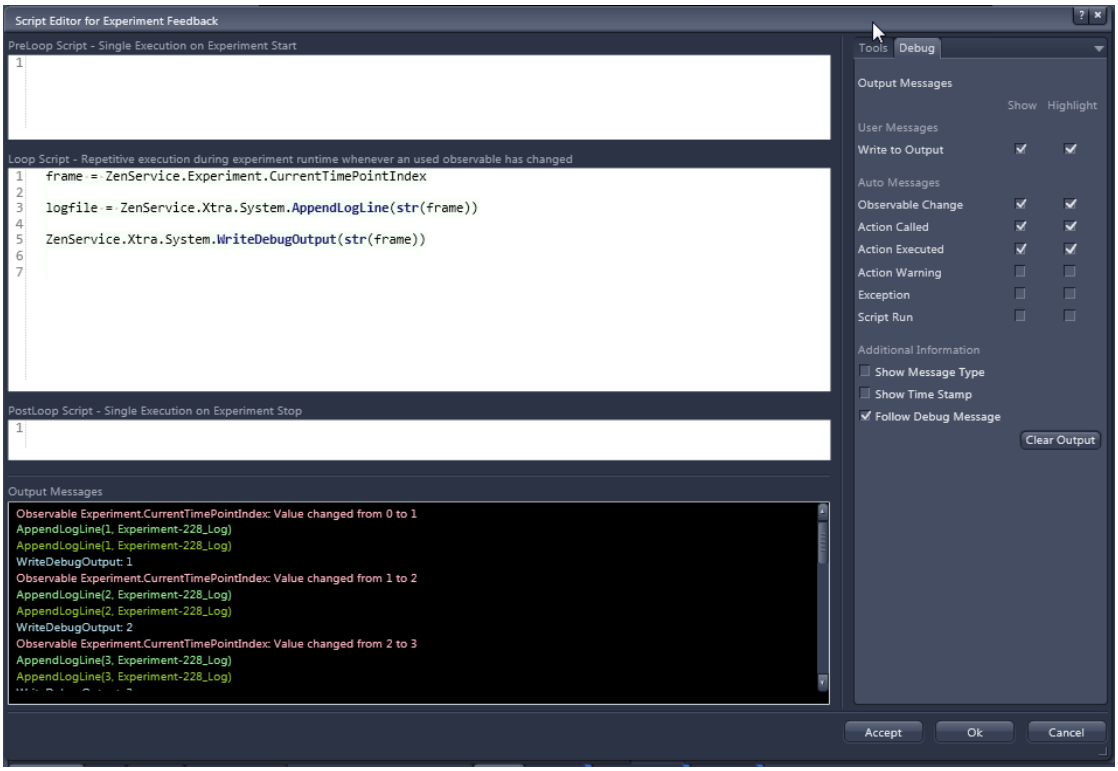

Figure 11: Debugging output of the experiment feedback script editor

observables that are not part of the multidimensional experiment (e.g. frame index, time index, block index, etc.) change. Those observables could be time or temperature of the incubation. If the checkbox is not activated only experiment observables trigger the main loop script. (The checkbox is only visible when **Show All** is activated in the **Experiment Feedback** tool).

Fig. 12 shows the situation when the additional script runs is deactivated. An observable which is not part of the multidimensional experiment, in this case the ElapsedTimeInSeconds does not trigger the script run of the Loop Script. The same experiment with the same Experiment Feedback script is run in Fig. 13. This time however, the checkbox is activated and therefore also the change of the observable ElapsedTimeInSeconds triggers the execution of the Loop script. This can be used if the Loop Script is to be triggered by changes of external triggers or other observables.

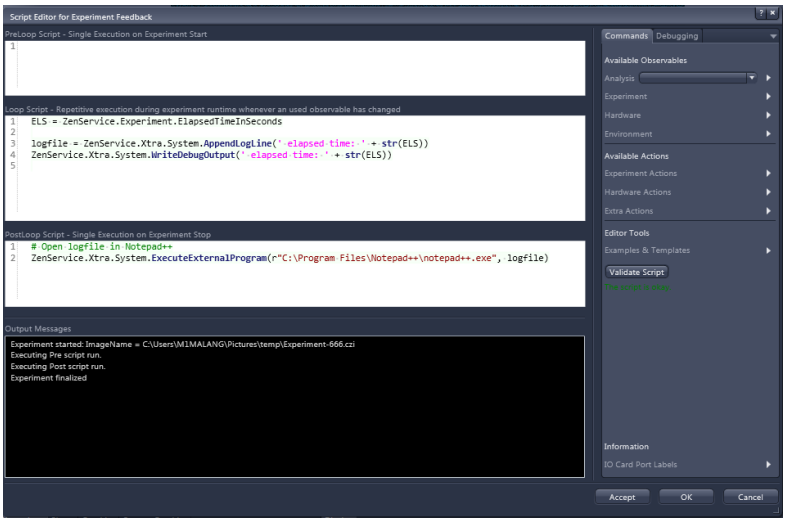

Figure 12: Additional script runs disabled

3 Prerequisites to Run the Application Examples

In the following chapter you will find some examples about how to use the Experiment Feedback in ZEN 3.5 (blue edition). Some of these examples will execute Python scripts to display the acquired data. To be able to run the examples using Python scripts you need to have a Python installation. In oder to get you startet as quickly as possible, the examples provided in this documentation make use of the ZeissPython. That means you don't need a separate Python installation if you just want to execute the sample scripts. However, more advanced Python scripts might require other libraries which are not part of the ZeissPython. In that case we recommend installing your own Python distribution.
In order to simulate experiments without having a real microscope to create the imaging data, the section 3.4 describes how to set up an acquisition that uses the sample data provided on the ZEN 3.5 (blue edition) DVD to simulate a real experiment.

3.1 Python Installation

To plot the results of the image analysis we will use Python scripts. This is a logical choice since the feedback experiment scripts and ZEN 3.5 (blue edition) macros are also written in Python.

Since ZEN blue 2.5, ZEN provides its own Python distribution based on Anaconda Python, which can be normally found here: **"c:/Program Files/Carl Zeiss/ZeissPython/"**. This Python distribution can be used to run the plotting scripts as well by telling your operating system to run *.py files using

"C:/Program Files/Carl Zeiss/ZeissPython/PyYYYYMMDD/env/python.exe".

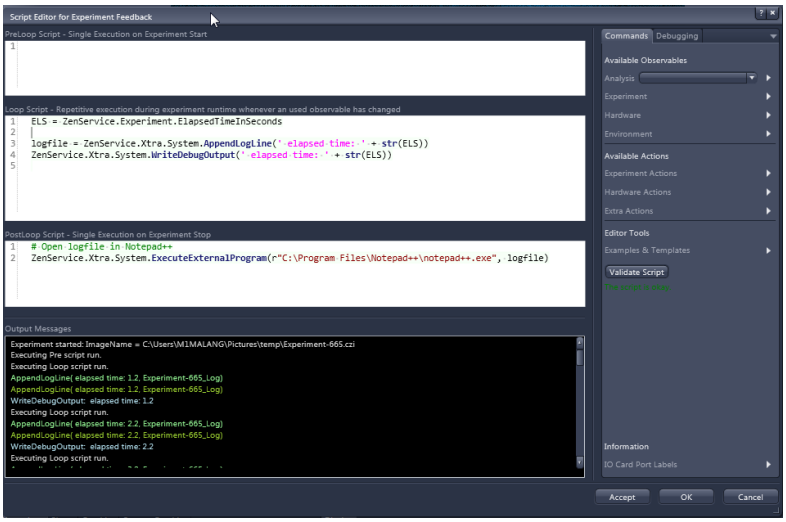

Figure 13: Additional script runs enabled

INFO:

For more convenience you can add the location of the Python.exe to the environmental variable PATH to specify the directory where Python is located on the computer. This allows you to start Python from ZEN without typing the whole path.

There are various other possibilities to install Python and the required modules. Another popular way is to install a complete distribution, for instance the Anaconda distribution, which can be downloaded here:

- **Anaconda Python** - https://www.anaconda.com/distribution/

Just use the full install option and follow the instructions. Of course it is also possible to use any other Python distribution.

In any case, when using Python to plot data and calling the plotting scripts from within ZEN it might be required (depending on your windows system), to check for possible issues described here in chapter Issues

3.2 Useful Python Links

You can find a lot of information about Python on the internet. Here is a selection of useful websites:

- **IPython** - www.ipython.org
- **Python Package Index** - https://pypi.org/

- **The Python Tutorial** - https://docs.python.org/3/tutorial/
- **Official IronPython** - http://ironpython.net/
- **MatPlotLib** - http://matplotlib.org/

3.3 Install Fiji (optional)

The installation of Fiji is only necessary if you want to use Fiji for further image analysis. In this tutorial, we will use it to demonstrate how you can acquire images with ZEN 3.5 (blue edition) and open the resulting data automatically in another application, e.g. Fiji, for further processing (cf. section 4.4). The release version of Fiji can be downloaded from here: http://fiji.sc/Downloads. Before installing Fiji note:

- Use the 64-bit version of Fiji
- Place the Fiji folder here: C:\Users\Public\Documents\Fiji ("Program Files" should not be used)
- Make sure your Fiji is always up-to-date

INFO:

Add the location of the ImageJ-win64.exe to the environmental variable PATH to specify the directory where Fiji is located on the computer. This allows you to start Fiji from ZEN without typing the whole path to Fiji.

3.4 Simulated Sample Camera

The simulated sample camera is an extremely useful feature to test and demonstrate most of the capabilities of the Experiment Feedback module. The idea is to place some "real" sample images in a specific folder and load these images for a simulated experiment including a real online analysis.

This requires a special XML-file (CZIS_Cameras.xml) which is usually located in: C:\ProgramData\Carl Zeiss\MTB2011\X.X.X.XX\CZIS_Cameras.xml (depending on the currently installed MTB version).

If the microscope components are configured correctly within the MTB, you can specify the folder to the demo images within ZEN 3.5 (blue edition) on the **Acquisition** tab in the **Acquisition Mode** tool (make sure to activate **Show All** to see all options). Under **Model Specific** you can define the **Image Path** where the sample data from this tutorial or other sample images are located (Fig. 14). Activate the **Simulate Exposure** checkbox to simulate the effect of the exposure time.

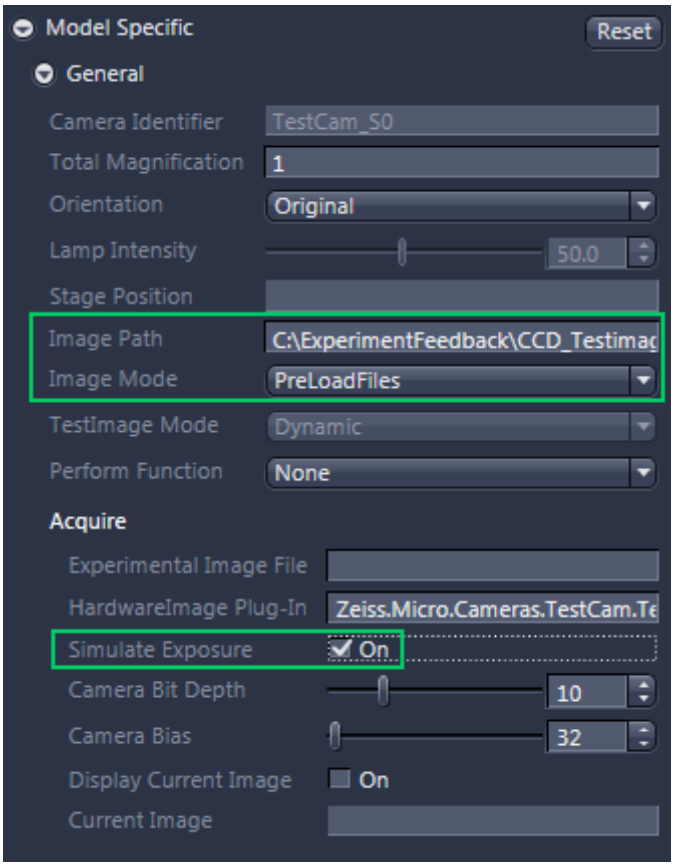

Figure 14: The ZEN 3.5 (blue edition) **Acquisition Mode** tool has the camera settings **Image Path** and **Image Mode** to load sample images

INFO:

You can test if the simulated sample camera was successfully set up by clicking on the **Live** button on the **Acquisition** tab. If everything is set up correctly the images in the defined folder should load in a loop in the main window automatically.

4 Application Examples

In this chapter you will find a collection of examples to illustrate some of the capabilities of the **Experiment Feedback**. You will learn for example how to call external applications from ZEN 3.5 (blue edition), how to define a stop-criterion for the Feedback Experiment and how to modify the acquisition according to the measured data.

If you want to run the provided sample scripts, copy the Experiment files (*.czexp) from the ZEN DVD: ...\Manuals\ZEN (blue edition)\OAD Content\Experiment Feedback\Exp_ with_ Scripts into your ZEN Folder: C:\Users\...\Documents\Carl Zeiss\ZEN\Documents\Experiment Setups and copy the Python scripts from the DVD e.g. under C:\Python_Scripts.

The examples are optimized to be used with the ZeissPython, if you use a different Python distribution, please adapt the path variables in the scripts accordingly.

In the beginning of each Experiment Feedback script which calls an external Python script we first import the necessary libraries, and make sure that the working directory is set properly to the location of the external Python scripts.

```
1 import sys
2 sys.path.append('C:\\Program Files\\Carl Zeiss\\ZEN 2\\ZEN 2 (blue edition)\\IronPython\\Lib')
3 import os
4 os.chdir(r'C:\Python_Scripts')
```

4.1 Count cells, write logfile and start external data display

The main purpose of this workflow is to demonstrate the possibilities of Feedback Experiments from a "broader" point of view. This example demonstrates the possibility to call an external application from within a feedback experiment in order to display the results of the online analysis after the acquisition is finished. The task is to acquire the data, analyze the number of cells per image, write the data into a log file and finally display the results and save the output.

Idea or Task:

- Set up an experiment
- Create an image analysis setting
- Run the feedback experiment and run the image analysis for every frame
- Display the results with an external application

Definition of the Feedback Experiment

For this example you can use the sample image folder ...\Testimages\96_well as described in 3.4. This folder contains 96 single TIFF images with cells where the nucleus is stained with DAPI. In order to set up the experiment you can simply define a DAPI channel and test if the sample camera gives the correct output. Once this works, just follow the general workflow. For this example you can just set up a time-lapse experiment with the following parameters (Fig. 15):

- Cycles = 10
- Time Interval = 1 s
- Channel = DAPI

Or you can simply load the time-lapse experiment called: EF_01_Plot_Cell_Count_Simple.czexp.

Definition of the Image Analysis Pipeline

The next step is to make a snap shot in order to create some sample data which you can use to set up the image analysis. (This tutorial assumes that the user is familiar with the ZEN 3.5 (blue edition) image analysis wizard, therefore only the crucial steps necessary to be able to set up the feedback experiment will be described. For more information about how to set up an image analysis using the image analysis wizard, please refer to the ZEN 3.5 (blue edition) user manual. Please also note that it is not possible to use interactive steps from the wizard for Experiment Feedback scripts).

For this example you should create a measurement program called: **EF_Count_Cells_DAPI.czias** which you should set up as follows:

1. Define the object and region class – in this case **SingleCell** and **Cells** (Fig. 16).

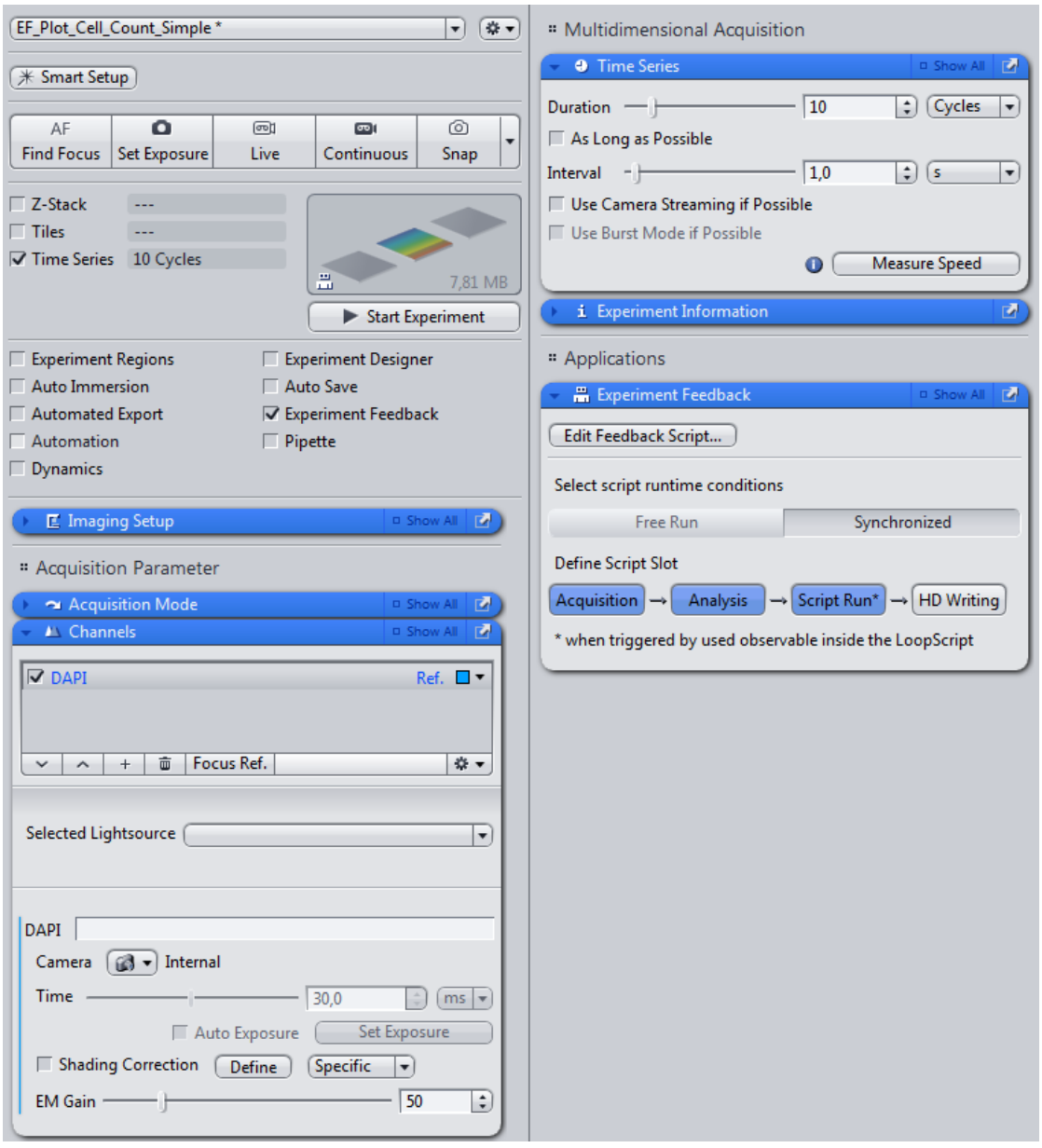

Figure 15: ZEN - Set up a time-lapse experiment

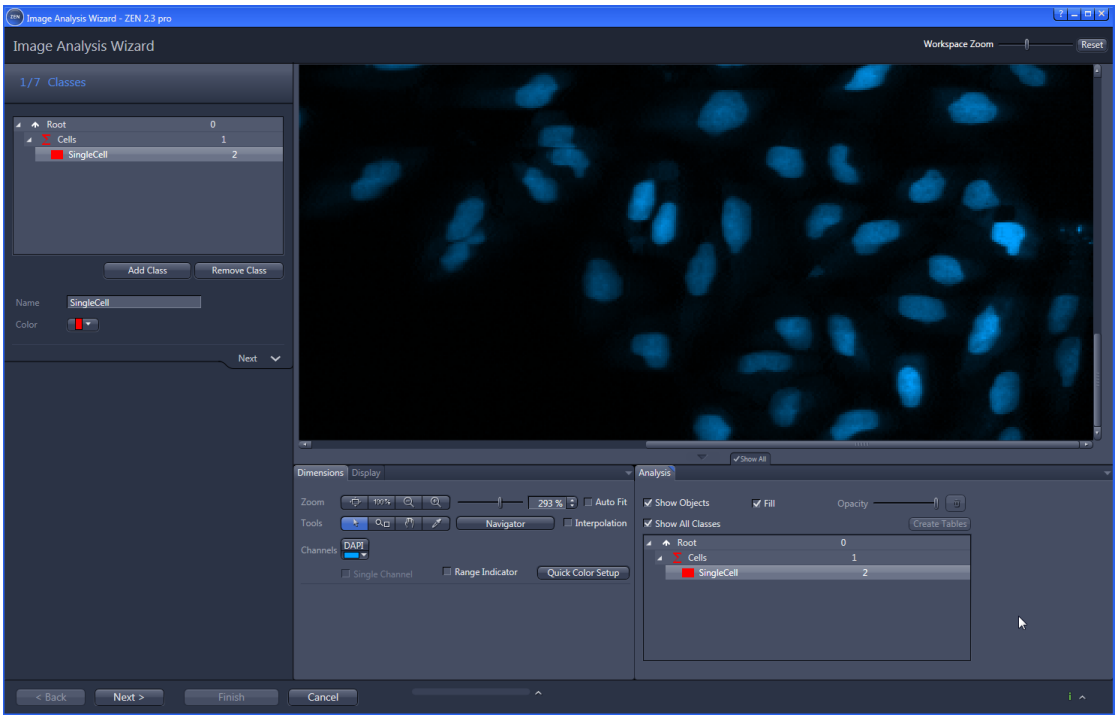

Figure 16: Image Analysis Wizard: Define the classes

2. Adjust the segmentation parameters in order to count single cells (Fig. 17). (For this example it is not crucial to get exactly the same number of cells as in this tutorial.)

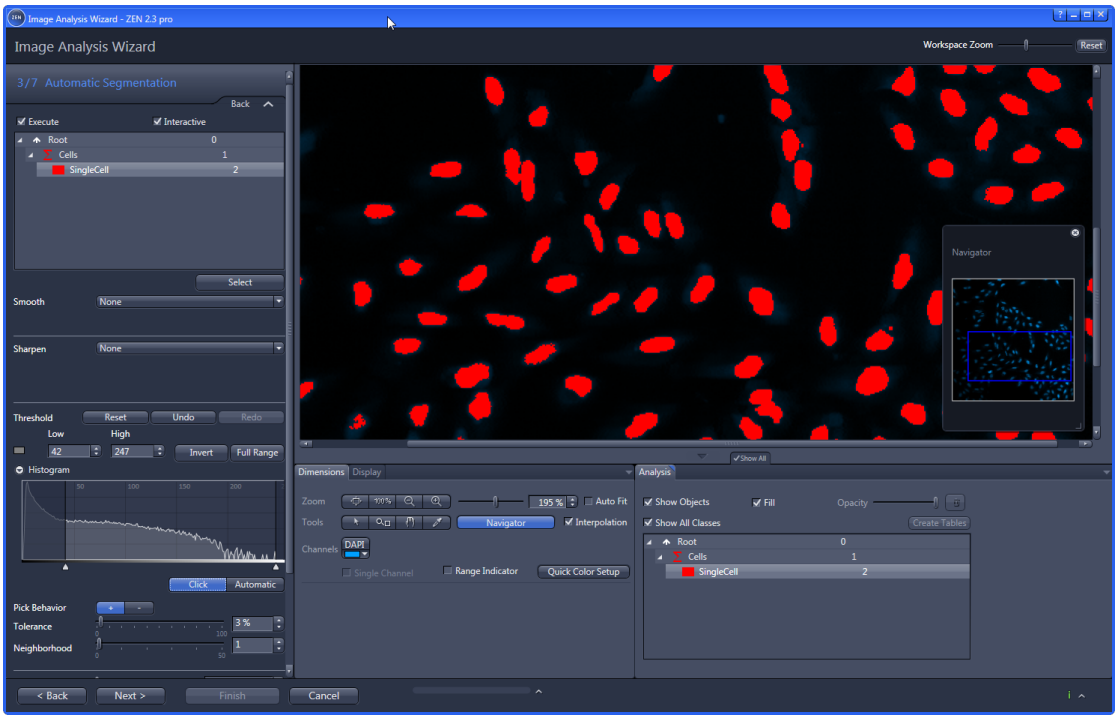

Figure 17: Image Analysis Wizard: Segmentation of the nuclei

3. Define the features for all objects (parameters for 'Cells') and for the single objects (parameters for 'SingleCell'). For this example it is sufficient to define **Count** and **Image Index Time** as parameters for 'Cells', which give you the number of single cells for each frame and the image index time, respectively (Fig. 18).

INFO:

Note that only the features you define within the image analysis wizard will later be available inside the Experiment Feedback.

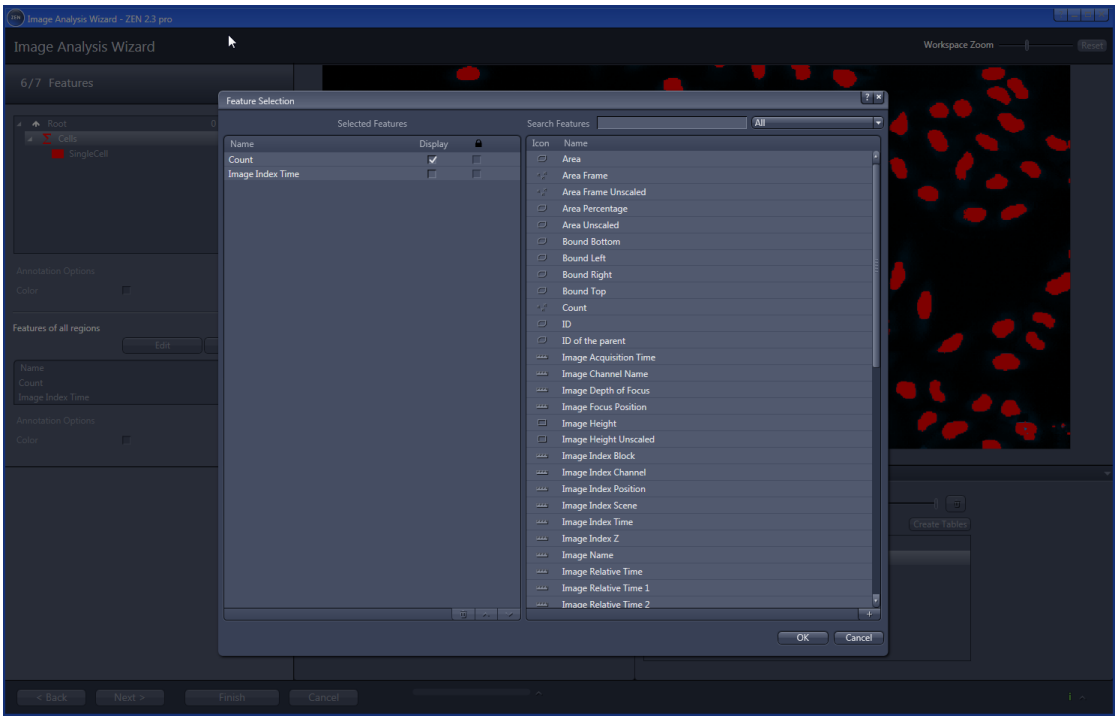

Figure 18: Image Analysis Wizard: Select the measurement features

4. Finalize the image analysis pipeline and check the results (Fig. 19).

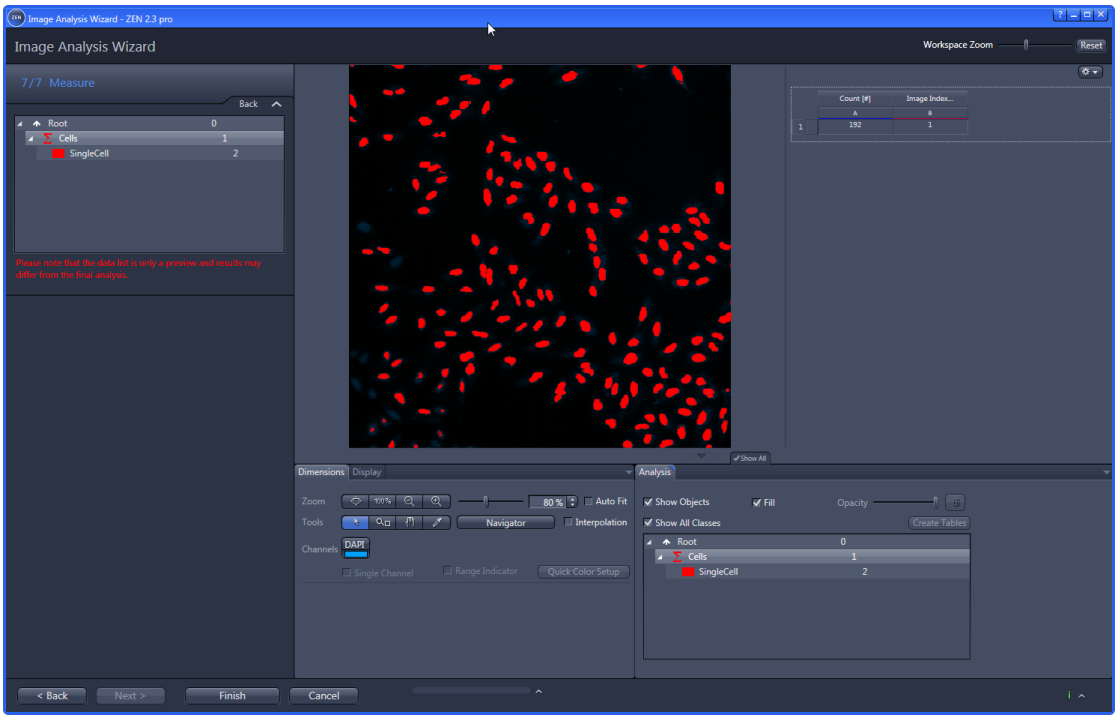

Figure 19: Image Analysis Wizard: Finalize and check the results

5. Run the analysis pipeline using the **Analyze** button and check the results to ensure that the image analysis gives the results for further use within the Experiment Feedback (Fig. 20).

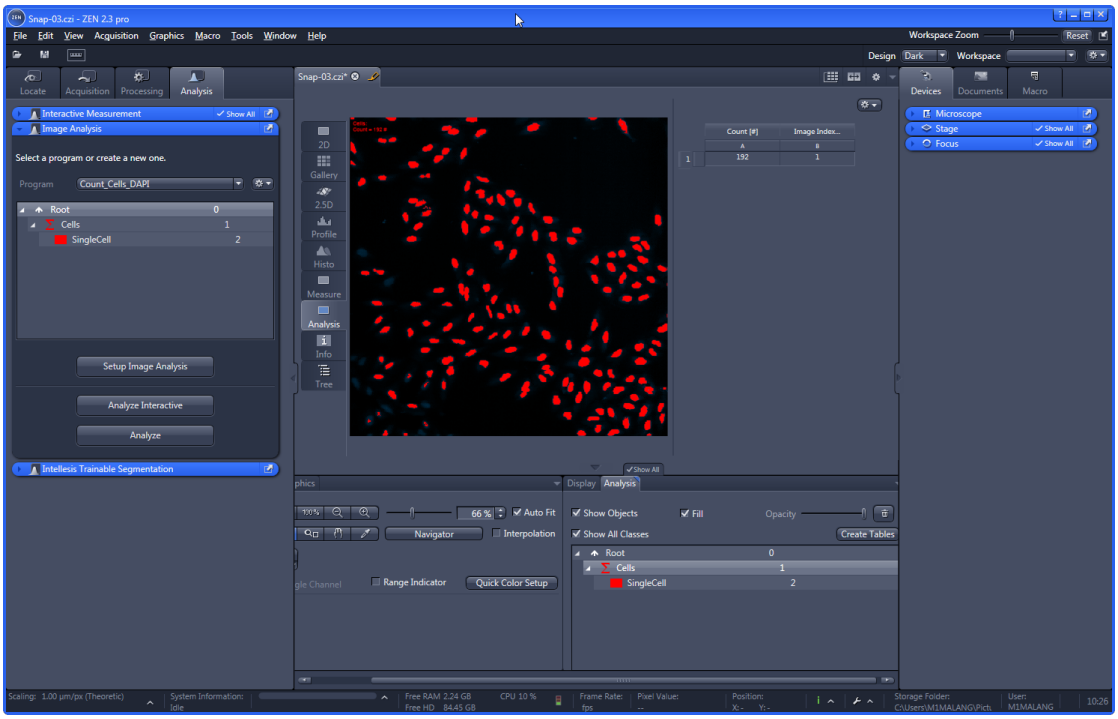

Figure 20: Image Analysis: Run the image analysis on sample data

Creation of the Feedback Script

The next step is to create the actual feedback script. Go to the **Experiment Feedback** box on the **Acquisition** tab. For this example we choose the synchronization settings shown in figure **15**. The order of those sequential steps (one after the other) will be:

1. **Run 1st image acquisition, e.g. the 1st time frame.**
2. **Analyze the 1st image just acquired.**
3. **Run script again when the image analysis of the 1st image is finished.**
4. **Repeat the process until all images are acquired.**

Next press the **Edit feedback Script...** button. If you have previously opened the **EF_01_Plot_Cell_Count_Simple.czexp** file, the script editor will already contain the necessary code to run the experiment, otherwise the empty script editor opens up. In that case you can enter the script as shown below.

The main idea of this feedback experiment is to count the number of cells for every frame, log the data and display the result at the end. It is important to keep in mind that there is more than one possible solution how to write a feedback script that fulfills a certain task. Whatever works is fine, and the solution shown here is just one out of a few possible solutions to point new users into the "right direction" and show some useful hints and tricks.

The first step is to select the proper image analysis pipeline in order to be able to use the parameters defined in the image analysis inside the feedback script (Fig.21). Select the image analysis pipeline you just created, i.e. **EF_Count_Cells_DAPI**.

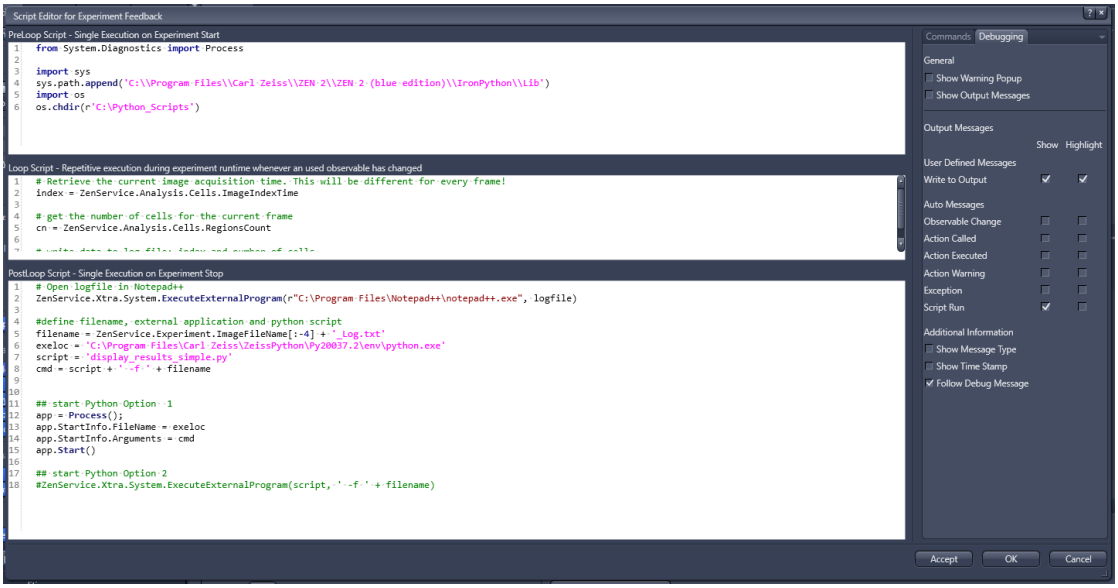

Figure 21: Select the image analysis pipeline and create the script

- **PreLoop Script**: Single execution on experiment start
  The first part of the script editor is frequently used to initialize variables, so they are available in the experiment feedback script.

- **Loop Script**: Repetitive execution for each loop
  The main loop of the script is executed during experiment runtime every time an observable has changed or is used inside the script. In this example the current frame number and the number of detected objects for each frame will be passed to the script and will be written into a logfile. The first column of the logfile contains the frame number, the 2nd contains the number of counted cells. The two columns are separated by tabs. This loop is repeated for every frame of the image acquisition.

```
### ------------------- PreScript ----------------------------------------- ###


from System.Diagnostics import Process
import sys
sys.path.append('C:\\Program Files\\Carl Zeiss\\ZEN 2\\ZEN 2 (blue edition)\\IronPython\\Lib')
import os
os.chdir('C:\\Python_Scripts')


### ------------------- LoopScript ---------------------------------------- ###


# Retrieve the current image acquisition time. This will be different for every frame!
#index = ZenService.Analysis.Cells.ImageIndexTime
index = ZenService.Experiment.CurrentTimePointIndex

# get the number of cells for the current frame
cn = ZenService.Analysis.Cells.RegionsCount

# write data to log file: index and number of cells
```

```
logfile = ZenService.Xtra.System.AppendLogLine(str(index) + "\t" + str(cn))



### -------------------- PostScript ------------------------------------------ ###


# Open logfile in Notepad++
ZenService.Xtra.System.ExecuteExternalProgram(r"C:\Program Files (x86)\Notepad++\notepad++.exe", logfile)

#define filename, external application and python script
filename = ZenService.Experiment.ImageFileName[:-4] + '_Log.txt'
exeloc = 'C:\\Program Files\\Carl Zeiss\\ZeissPython\\Py20283.2\\env\\python.exe'
#script = 'C:\\Python_Scripts\\display_results_simple.py'
script = "display_results_simple.py"
cmd = script + ' -f ' + filename

ZenService.Xtra.System.WriteDebugOutput(filename)
ZenService.Xtra.System.WriteDebugOutput(cmd)

## start Python Option  1
app = Process();
app.StartInfo.FileName = exeloc
app.StartInfo.Arguments = cmd
app.Start()

## start Python Option 2
#ZenService.Xtra.System.ExecuteExternalProgram(script, ' -f ' + filename)
```

- **PostLoop Script**: Single execution on experiment stop
  This part of the feedback script is executed only once after completion of the acquisition. In this example the logfile will be opened with an external program (i.e. a text editor) and additionally a python script **(**display_results_simple.py) is executed. This python script automatically displays the data after the experiment is finished and saves the graph. (Note that the paths to the external programs to be executed from the feedback experiment need to be adapted to match their location on your computer).

It strongly depends on the application, what is a good way to display the data, so this example is just a suggestion. It is also important to note that with the execution of external programs and code, the user leaves the ZEN 3.5 (blue edition) world. As already mentioned it makes sense to use Python to write the script for displaying the data in order to use the same scripting language as in the Feedback Experiment, but this is of course a matter of taste.

Python Script for Data Display

Once the format of the data log file is defined, you can start thinking about how to display the data. This task requires a certain degree of Python knowledge, but very good and easy to use examples can be found on the internet and especially here:
http://matplotlib.org/gallery.html
Just check the example code that is given on the website on how to create the desired types of plots and use this as a starting point.

To create a Python script you can use every text editor (e.g. Notepad++. . .) or Spyder or PyCharm to edit and test the script. This tutorial will not explain the Python script in detail, and it is important to keep in mint that the provided Python script is just an example and the correct way to import the data of course depends on the output of the analysis.

The main tasks of this example script are:

- Enable command line parsing, that means you can "call" the script and provide the filename of the input data file as an argument.
- Read in the data correctly.
- Display the data in an appropriate way.

Once the data are imported into Python, it can also be used for all kinds of subsequent analysis and further data processing.

```
# -*- coding: utf-8 -*-
"""
Created on Wed Sep 26 11:26:27 2012

@author:Sebastian Rhode
"""
import sys
import os
env = os.environ
env.update({'QT_API':'pyside'})
# To ensure that only libraries that we have control of are used
env['PATH'] = os.path.join(os.path.dirname(sys.executable), 'Library', 'bin')
import numpy as np
import optparse

import matplotlib.pyplot as plt

# configure parsing option for command line usage
parser = optparse.OptionParser()

parser.add_option('-f', '--file',
    action="store", dest="filename",
    help="query string", default="No filename passed")

parser.add_option('-b', '--block',
    action="store", dest="block",
    help="set to False for interactive behaviour", default="True")

# read command line arguments
options, args = parser.parse_args()
savename = options.filename[:-4] + '.png'
print('Filename: ', options.filename)
print('Savename: ', savename)

block = True
if options.block == 'False':
        block = False

# load data
data = np.loadtxt(options.filename, delimiter='\t')
# create figure
figure = plt.figure(figsize=(10,5), dpi=100)
ax1 = figure.add_subplot(121)
ax2 = figure.add_subplot(122)

# create subplot 1
ax1.plot(data[:,0],data[:,1],'bo-', lw=2, label='Cell Count')
ax1.grid(True)
ax1.set_xlim([data[0,0]-1,data[-1,0]+1])
ax1.set_xlabel('Frame Number')
ax1.set_ylabel('Cells detected')

# create subplot 2
ax2.bar(data[:,0],data[:,1],width=0.7, bottom=0)
ax2.grid(True)
ax2.set_xlim([data[0,0]-1,data[-1,0]+1])
ax2.set_xlabel('Frame Number')

# adjust subplots
figure.subplots_adjust(left=0.10, bottom=0.12, right=0.95, top=0.95,wspace=0.20, hspace=0.20)
# save figure
plt.savefig(savename)
# show graph
plt.show(block=block)
```

Running the Feedback Experiment

Activate **Experiment Feedback** in the **Acquisition** tab and click on **Start Experiment**. For this example a simple time series is recorded. Every frame is analyzed according to the defined image analysis pipeline and the data is written into the specified logfile. If everything worked fine, the editor you selected for displaying the logged data will open once the acquisition is completed. The external application, in this case Python, will display the number of detected cells per frame using the Experiment-XYZ_Log.txt file as a data source (Fig. 22). Here the first column represents the frame number and the second column the number of detected cells of each frame. Both values are separated with a tab. Of course the format, how the data is written into the logfile has to be taken into account when setting up the Python script used for displaying the acquired data.

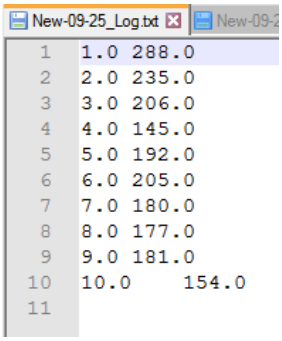

Figure 22: Logfile created by the experiment feedback script

Fig. 23 shows the plot created by the Python script and a PNG-file is the saved to the image folder.

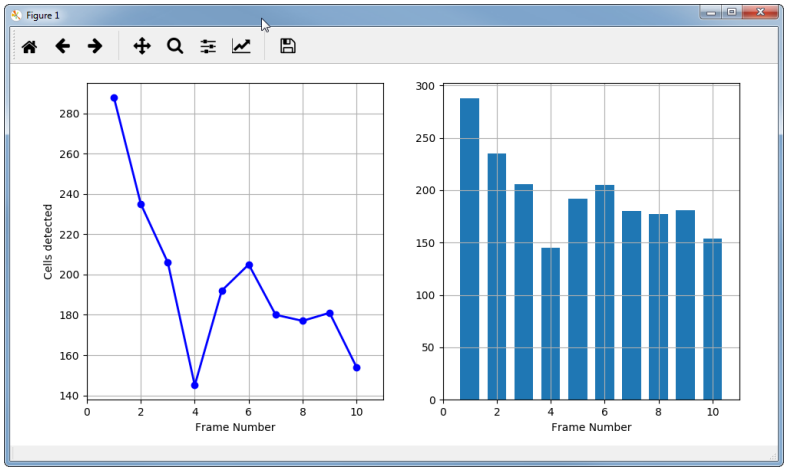

Figure 23: Plot created by the Python script

This example demonstrates the possibility to call an external application, in this case Python,

to display the analysis data collected by the Experiment Feedback script. A next step could be to try the data display online ("while the data arrives") as described for example in section 4.3 or to include further image analysis steps.

4.2 Acquire Tile Images until a Total Number of Objects is Reached

In this application example you will learn how to set up a Feedback Experiment that stops the acquisition after a previously defined number of cells has been reached. This can be used to save time and storage if only a certain number of cells is required.
To run this experiment it is nice, but not necessary, to have a real microscope setup with motorized XY-stage and a real sample with objects to be counted, e.g. stained nuclei. Alternatively, you can use the data in the folder ...\Testimages\96_well to setup this experiment.

Idea or Task:

- Define an experiment
- Count the number of objects
- Run the experiment and stop the acquisition when a certain amount of objects has been counted
- Display the results with an external application

Definition of the Feedback Experiment

Set up a tile experiment with a sufficiently high number of tiles (so there will enough cells to count) and save it as **EF_02_Count_Cells_Tiles_DAPI.czexp**. You can use the following experiment settings:

- 2.5x objective
- Number of tiles $= 6 \times 6 = 36$ images (this depends on the selected magnification)
- Channel = DAPI (or whatever the nuclei are stained with...)

After setting up the experiment you should make a snapshot acquisition in order to have some data for setting up the image analysis.

Definition of the Image Analysis Pipeline

Set up an image analysis pipeline that counts the number of cells in each frame. You can use a similar analysis pipeline as described in example 4.1. Be sure to define the following features for **regions**:

- **Count**
- **Image Index Time**
- **Image Stage Position** X
- **Image Stage Position** Y

for later use in the Feedback Experiment.

The Feedback Script

The Feedback Experiment should perform the following tasks:

1. For each tile image count the number of objects
2. Calculate the total number of objects acquired so far
3. If the desired number of objects was reached, stop the image acquisition
4. Create a logfile for the data, display the data and save the plot

```
### -------------------- PreScript ---------------------------------------------- ###


from System.Diagnostics import Process
cells_total = 0
logfile = ZenService.Xtra.System.AppendLogLine('Tile \t Cells \t Total \t PosX \t PosY')


import sys
sys.path.append('C:\\Program Files\\Carl Zeiss\\ZEN 2\\ZEN 2 (blue edition)\\IronPython\\Lib')
import os
os.chdir(r'C:\Python_Scripts')


### -------------------- LoopScript --------------------------------------------- ###


# get cell number for current tile
cn = ZenService.Analysis.Cells.RegionsCount
# get current tile number
tn = ZenService.Experiment.CurrentTileIndex
# calculate sum of cells measured up to now
cells_total = cells_total + cn

# stop condition if a cell number is reached
if (cells_total > 3000):
    ZenService.Actions.StopExperiment()

#read xy position of current image
posx = ZenService.Analysis.Cells.ImageStageXPosition0
posy = ZenService.Analysis.Cells.ImageStageYPosition0

# write into log file
logfile = ZenService.Xtra.System.AppendLogLine(str(tn) + '\t' + str(cn) + '\t' + str(cells_total) + '\t' + str(posx) + '\t' +
↪  str(posy))


### -------------------- PostScript --------------------------------------------- ###


# open editor to display the logfile
ZenService.Xtra.System.ExecuteExternalProgram(r"C:\Program Files (x86)\Notepad++\notepad++.exe", logfile)

#define filename, external application and python script
filename = ZenService.Experiment.ImageFileName[:-4] + '_Log.txt'
exeloc = 'C:\Program Files\Carl Zeiss\ZeissPython\Py20037.2\env\python.exe'
script = 'display_results_tiles.py'
cmd = script + ' -f ' + filename

## start Python Option  1
app = Process();
app.StartInfo.FileName = exeloc
app.StartInfo.Arguments = cmd
app.Start()

## start Python Option 2
#ZenService.Xtra.System.ExecuteExternalProgram(script, ' -f ' + filename)
```

- **PreLoop Script**: Single execution on experiment start

In line 5 a variable called **cells_total** is initialized. This variable serves as a container for the number of total objects. This value is not available from the image analysis directly, and therefore needs to be calculated by adding up the number of cells for every frame. In line 6 a header line for the logfile is created to make the logfile more readable.

- **Loop Script**: Repetitive execution for each loop

  Whenever a new tile is acquired, the total cell number detected so far needs to be calculated. Therefore, in each loop the number of cells of the current tile which is obtained via (**ZenService.Analysis.Cells.RegionsCount**) is added to the sum of cells detected so far, which is stored in **cells_total** (line 17). After that, an if-statement checks if the total number of cells exceeds a certain value, e.g. 3000 (feel free to adapt the value). If this statement is true, the experiment is stopped (line 21).
  For the display with a Python script, the X and Y positions of the stage for each tile are recorded and written in the logfile, together with tile number, number of cells per frame and total number of cells (line 28). The 1st column of the logfile contains the tile number, the 2nd the number of cells, the 3rd the total amount of cells so far followed by X and Y position of the stage.

- **PostLoop Script**: Single execution on experiment stop

  The last part of the script is just to start the Python script to display the data and open the logfile in an editor.

Figure 24 shows the recorded logfile for the experiment.

Experiment-78_Log.txt

	Tile	Cells	Total	PosX	PosY
2	1	309.0	309.0	-2637.674	-2153.544
3	2	243.0	552.0	-1997.674	-2153.544
4	3	225.0	777.0	-1357.674	-2153.544
5	4	184.0	961.0	-717.674	-2153.544
6	5	199.0	1160.0	-77.674	-2153.544
7	6	220.0	1380.0	562.326	-2153.544
8	7	192.0	1572.0	562.326	-1513.544
9	8	195.0	1767.0	-77.674	-1513.544
10	9	193.0	1960.0	-717.674	-1513.544
11	10	163.0	2123.0	-1357.674	-1513.544
12	11	165.0	2288.0	-1997.674	-1513.544
13	12	188.0	2476.0	-2637.674	-1513.544
14	13	155.0	2631.0	-2637.674	-873.544
15	14	192.0	2823.0	-1997.674	-873.544
16	15	238.0	3061.0	-1357.674	-873.544
17					

Figure 24: Logfile created by the Experiment Feedback Script

Creating a Python Script for the Data Display

The Feedback Experiment will call the Python script **(**display_results_tiles.py) to display the data graphically after the image acquisition is completed. Of course there many ways to display the data (for possible implementations cf. http://matplotlib.org/). The script shown below is just one possibility. It is more or less a variation of the script already used in section 4.1. It will display the data from the logfile in three different ways to demonstrate different possibilities.

These are the key steps of the python script:

- Configure a parser to read the command line
- Obtain the filename and create a savename (*.png) for the result
- Load the data from the logfile in **data**
- Create a figure with three subplots
- Set values to be displayed for the three plots as well as size and style of the plot and define labels
- Save the plot
- Display the plot

```
# -*- coding: utf-8 -*-
"""
Created on Wed Sep 26 11:26:27 2012

@author: m1srh
"""
import sys
import os
env = os.environ
env.update({'QT_API':'pyside'})
# To ensure that only libraries that we have control of are used
env['PATH'] = os.path.join(os.path.dirname(sys.executable), 'Library', 'bin')
import numpy as np
import optparse

import matplotlib.pyplot as plt

# configure parsing option for command line usage
parser = optparse.OptionParser()

parser.add_option('-f', '--file',
    action="store", dest="filename",
    help="query string", default="spam")

parser.add_option('-b', '--block',
    action="store", dest="block",
    help="set to False for non interactive behaviour", default="True")

# read command line arguments
options, args = parser.parse_args()

savename = options.filename[:-4] + '.png'
print ('Filename: ', options.filename)
print ('Savename: ', savename)

block = True
if options.block == 'False':
        block = False

# load data
data = np.loadtxt(options.filename, delimiter='\t', skiprows=1)

# create figure
figure = plt.figure(figsize=(12,4), dpi=100)
ax1 = figure.add_subplot(131)
ax2 = figure.add_subplot(132)
ax3 = figure.add_subplot(133)
```

```
48
49  # create subplot 1
50  ax1.bar(data[:,0],data[:,1],width=0.7, bottom=0, color='blue')
51  ax1.grid(True)
52  ax1.set_xlabel('Tile Number')
53  ax1.set_ylabel('Cells detected')
54
55  # create subplot 2
56  ax2.bar(data[:,0],data[:,2],width=0.7, bottom=0, color='green')
57  ax2.grid(True)
58  ax2.set_xlabel('Tile Number')
59  ax2.set_ylabel('Cells in total')
60
61  # create subplot 3
62  ax3.plot(data[:,3],data[:,4],'ro', markersize = 5)
63  ax3.grid(True)
64  ax3.set_xlim([data[:,3].min()-100,data[:,3].max()+100])
65  ax3.set_ylim([data[:,4].min()-100,data[:,4].max()+100])
66  ax3.set_xlabel('X Stage Position')
67  ax3.set_ylabel('Y Stage Position')
68
69  # add frame number directly to every data point
70  for X, Y, Z in zip(data[:,3], data[:,4], data[:,0]):
71      # Annotate the points
72      #ax3.annotate('{}'.format(int(Z)), xy=(X,Y), xytext=(5, 10), ha='right', textcoords='offset points')
73      ax3.annotate('{}'.format(int(Z)), xy=(X,Y), xytext=(0, 15), ha='right',
74                   textcoords='offset points',arrowprops=dict(arrowstyle='->', shrinkA=0))
75
76  # configure plot
77  figure.subplots_adjust(left=0.07, bottom=0.12, right=0.95, top=0.95,wspace=0.30, hspace=0.20)
78  # save figure
79  plt.savefig(savename)
80  # show graph
81  plt.show(block=block)
```

Running the Feedback Experiment

Activate **Experiment Feedback** in the **Acquisition** tab and click on **Start Experiment**. In the ZEN 3.5 (blue edition) main window you can observe how the tiles are acquired one after another. For this example the tile regions were initially set to 6×6, but depending on the actual selected size, the image analysis and the sample, the acquisition will stop earlier. The data log file in Fig. 24 shows that the tile acquisition was stopped after 15 tiles out of 36 that were initially defined. After 15 tiles the total number of cells was greater than 3000, therefore the experiment was stopped, skipping the remaining 21 tiles. This way the Feedback Experiment allows you to save a time and space.

After the acquisition is stopped, the Python script is started to plot the data (Fig. 26). The 1st subplot contains the number of cells per tile, the 2nd the total cell number acquired after the n^{nt}-frame and the last plot visualizes the XY-stage positions of each acquired frame. Additionally, the script saves the plot.

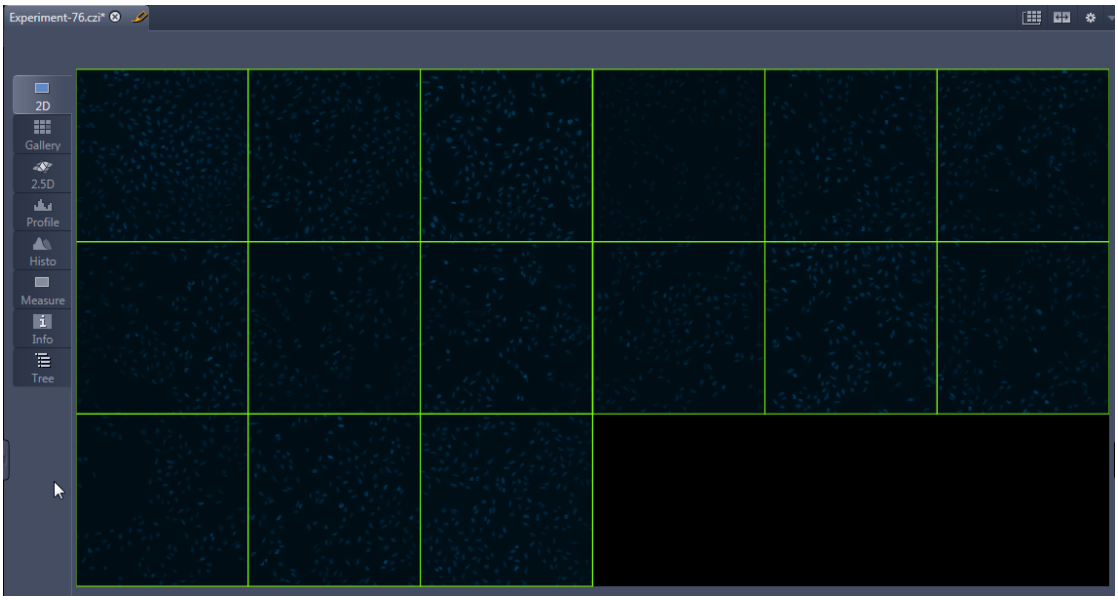

Figure 25: The acquisition was terminated after the the "stop" criterion was met

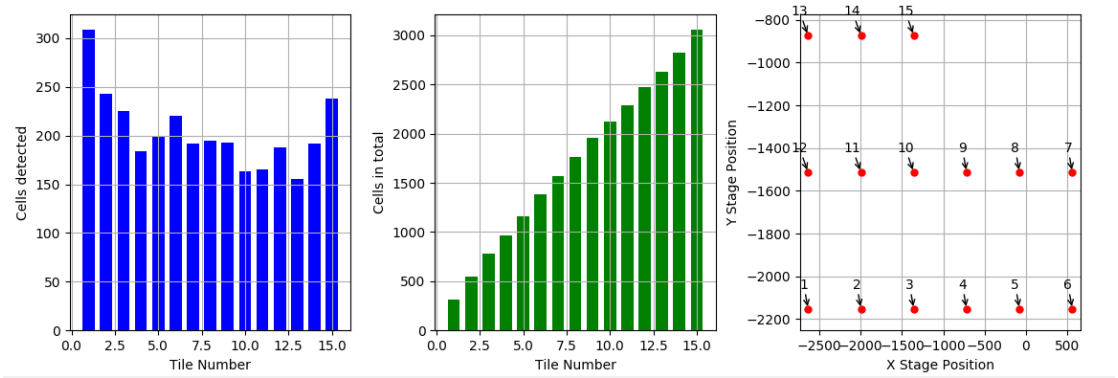

Figure 26: The Python script is used to graphically display the content of the log file

This example demonstrates the basic possibility to change an ongoing experiment based on the results obtained by the online image analysis. You can use Python scripting to calculate all kinds of custom parameters and, if required by the application, you can also extend the Feedback Experiment to a highly sophisticated decision tree.

4.3 Online Data Display

In some experiments it can be extremely useful to check some experiment or image analysis parameters already while the experiment is running, e.g. to check if everything is as expected. The following example describes a typical use case for online data display. The task of this application is to set up a multi-well experiment and acquire one image per well. During the acquisition the online analysis determines the number of cells per well and a Python script is used to display the amount of cells per well color-coded with a heatmap.

Idea or Task:

- Setup image acquisition: acquire one image per well
- On experiment start-up start an online data display
- Count the number of objects per well/image
- Continuously monitor the data log file and refresh the display

Definition of the Feedback Experiment

For this example you can use the sample image folder ...\Testimages\96_well. It contains 96 single images with cells where the nucleus is stained with DAPI. This data set allows you to simulate a 96-well plate (or 96 tiles). In ZEN 3.5 (blue edition) set up a tiles-experiment with the following parameters:

- Positions = 96
- One image per well (position)
- Channel = DAPI (adapt to actual stain if you use another sample)

Alternatively you can directly load the provided experiment called EF_03_Count_Cells96Well_Heatmap.czexp which contains all experiment settings as well as the feedback script. Note that the stage motion follows a meander scheme. When have successfully set up the acquisition, make a snap shot to have some data for setting up the image analysis.

Definition of the Image Analysis Pipeline

Set up an image analysis pipeline that counts the number of cells in each frame. You can use a similar analysis pipeline as described in example 4.1. Be sure to define the following features for **Regions** as they are necessary for the feedback script:

- Regions Count (= number of cells)
- Image Scene Container Name (= well number)

- Image Scene Column (= column number)
- Image Scene Row (= row number)

The well number allows you to assign the acquired image to the correct well/position. (Alternatively you can use the image analysis pipeline EF_Nuclei_DAPI.czias).

Creation of the Feedback Script

In the pre-loop part of the feedback script you need to obtain the filename from ZEN and define the format of the multi-well plate so the Python script will put the information on the correct position. The Python script needs to be started in the pre-loop script for the online data display. The main loop gets the features necessary for the data display from the online analysis and writes them into the logfile. The external Python script will continuously monitor the logfile and update the data display accordingly. In the post-loop script you can optionally open the logfile with a text editor.

1. Get the filename of the logfile
2. Define the used plate format so that the external Python script correctly displays the data
3. Start the Python script with the defined parameters
4. Get the number of objects per well and write them into the logfile
5. Open the data logfile (optional)

```
### -------------------- PreScript ------------------------------------------- ###


from System.Diagnostics import Process


import sys
sys.path.append('C:\\Program Files\\Carl Zeiss\\ZEN 2\\ZEN 2 (blue edition)\\IronPython\\Lib')
import os
os.chdir(r'C:\Python_Scripts')


filename = ' -f ' + ZenService.Experiment.ImageFileName[:-4] + '_Log.txt'
# !!! watch the space before the -c in params !!!
params = ' -c 12 -r 8' # specify well plate format, e.g. 12 x 8 = 96 Wells

## specify the script used to display the data and call the script with arguments
exeloc = 'C:\Program Files\Carl Zeiss\ZeissPython\Py20037.2\env\python.exe'
script = "dynamic_plot_96well_animate.py"
cmd = script + filename + params


## start Python Option 1
app = Process();
app.StartInfo.FileName = exeloc
app.StartInfo.Arguments = cmd
app.Start()

## start Python Option 2
#ZenService.Xtra.System.ExecuteExternalProgram(script, filename + ' -c 12 -r 8')




### -------------------- LoopScript ------------------------------------------ ###


#get number of cells from current image
cn = ZenService.Analysis.Cells.RegionsCount

```

```
# get the current well name, column idex, row index and position index
well = ZenService.Analysis.Cells.ImageSceneContainerName
col = ZenService.Analysis.Cells.ImageSceneColumn
row = ZenService.Analysis.Cells.ImageSceneRow

# create logfile
logfile = ZenService.Xtra.System.AppendLogLine(str(well)+'\t'+str(cn)+'\t'+str(col)+'\t'+str(row))



### -------------------- PostScript ------------------------------------------ ###


ZenService.Xtra.System.ExecuteExternalProgram(r"C:\Program Files (x86)\Notepad++\notepad++.exe", logfile)
```

Python Script for the Data Display

In the main part of the script the data are imported and converted into a 2D array. The important point here is to use the column and row index to assign the number of cells to the correct well.

```

LabelX = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24]
LabelY = ['A','B','C','D','E','F','G','H','I','J','K','L','M','N','O','P']

labelx = LabelX[0:Nc]
labely = LabelY[0:Nr]

#print ("Label X: ",labelx)
#print ("Label Y: ",labely)


##############################
# Main part
##############################

def animate(i):

    try:

        datain = np.genfromtxt(options.filename, delimiter='\t', usecols=(1,2,3))
        well = np.zeros([Nr, Nc]) # create a new array of Size Nr x Nc filled with zeros

        for j in range(0, datain.shape[0]):

            # read entries
            cn = datain[j,0]
            col = datain[j,1]-1 # numpy is zero-based ...
            row = datain[j,2]-1
            # update well array at the correct position
            well[int(row), int(col)] = cn
            cn_max = np.max(datain[:,0])
            cn_min = np.min(datain[:,0])

        fig.clear()
        plt.title('Cell Count per Well') #plottitle
        # do the plot
        cax = plt.imshow(well, cmap='plasma', interpolation='nearest')
        # define and plot colorbar
        plt.clim(cn_min, cn_max)
```

Run the Feedback Experiment and watch the results

Start the experiment and watch the heatmap fill up with the detected number of objects per well. The number of detected cells is color coded. Yellow corresponds to a high number of objects, while blue represents a low object count. The final result is automatically saved as PNG-file within the same folder.

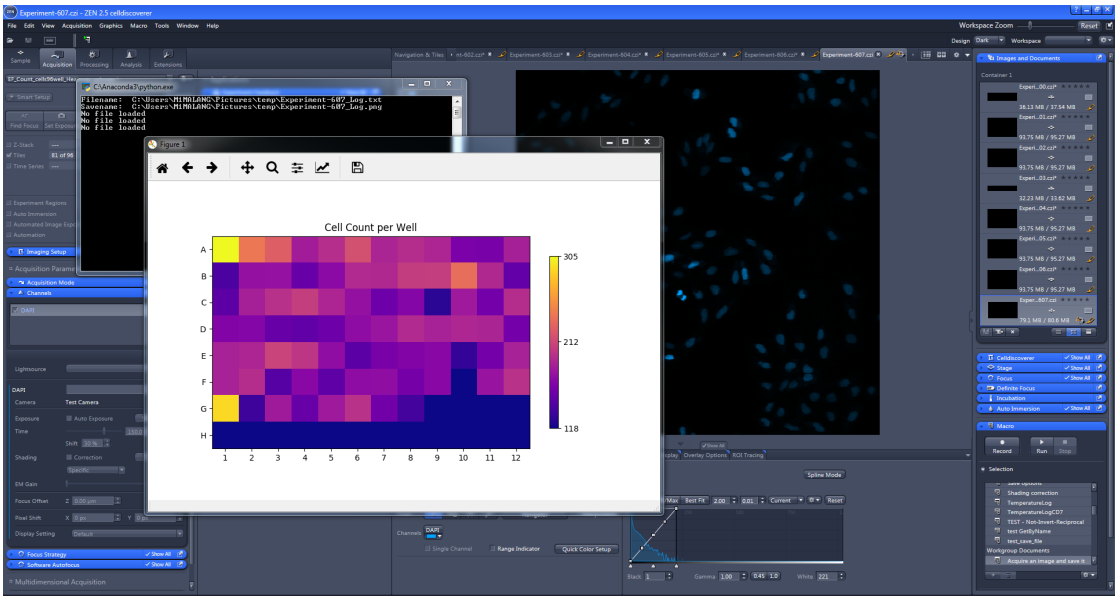

Figure 27: The heatmap is updated constantly during the course of the experiment.

Finally the data logfile is displayed with the defined text editor. (Adapt the paths within the feedback script, when using a program other than Notepad++ or the location is different on your system).

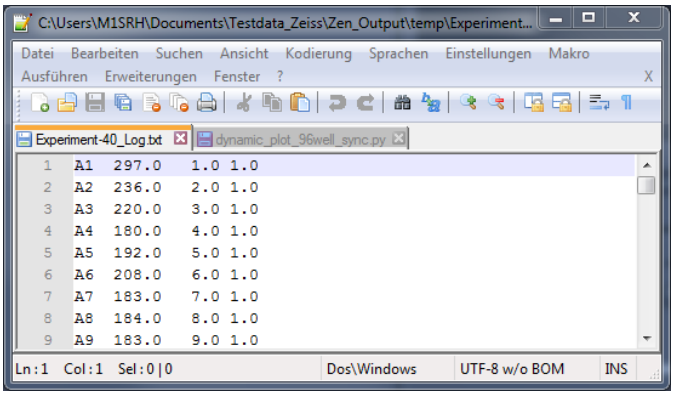

Figure 28: Logfile data

4.4 Acquire Image Data, Open in Fiji and Apply a Macro

The main purpose of this workflow is to demonstrate the possibilities to automate and customize an experiment. In order to be able to run this example you need to have an installation of Fiji (cf. section 3.3).

Idea or Task:

- Define an experiment in ZEN 3.5 (blue edition)
- Set up a simple image processing macro in Fiji for demonstration
- Set up the feedback experiment
- Run the feedback experiment and open the resulting CZI-file automatically with Fiji
- Further image processing with Fiji

The main idea of this example is simply to automate a workflow. That means it is not necessary to define a feedback rule based on observables or to set up an image analysis pipeline. This example just uses the **PostLoop Script** of the Experiment Feedback to call an external application (in this case Fiji) and pass the filename as well as a simple Fiji macro. Those three steps will be included directly into the ZEN experiment itself:

- Acquire the image data
- Transfer the image data directly to Fiji
- Apply the Fiji macro (*.jim) directly after the data transfer

For this application example you can set up any kind of experiment. For demonstration purposes we will use a simple Z-stack experiment.

Definition of the Feedback Experiment

For this example you can use the sample image folder **...\Testimages\HeLa_ZStack**. The folder contains 15 single TIFF images with cells where the mitochondria are stained with MitoTracker Red.
In ZEN 3.5 (blue edition) you first need to define a MitoRed channel and test if the sample camera gives the correct output. The next step is to set up a Z-stack experiment with 15 planes. Save the experiment as EF_04_Open_Fiji_after_End.czexp.

- Number of Z-planes = 15
- Channel = MitoRed

Finally, run the Z-stack experiment and save the *.czi file to have some data for setting up the Fiji macro.

Creating a Fiji macro

In the following we will set up a simple Fiji macro to demonstrate how to start an external application from ZEN 3.5 (blue edition) and perform an action on the acquired data. Fiji (as well as ImageJ) has powerful scripting capabilities. For further information and examples, please refer to:

- http://fiji.sc/wiki/index.php/Introduction_into_Macro_Programming
- http://rsb.info.nih.gov/ij/developer/macro/functions.html

If you already have a suitable Fiji macro, you can skip this part and continue at section **Creation of the Feedback Script** of this chapter. You can also find an example macro provided with the sample data which you may use: Open_CZI_and_MaxInt.ijm.

In order to create a macro for Fiji from scratch, you can use every text editor (e.g. Notepad++,. . .) but you can also use Fiji's built-in script editor (this is where the following screen-shots are taken from). You can access the script editor in Fiji via **Plugins** ▷ **New** ▷ **Macro** or use the macro-recorder to record a macro **Plugins** ▷ **Macros** ▷ **Record. . .** .
For starters we will define a very simple macro, which has the following function:

- Read the filename of the ZEN CZI-file via command line input
- Import the CZI-file using the Bio-Formats library (this format preserves most of the metadata, like image dimensions and stage coordinates)
- Perform image processing (maximum-projection of the Z-stack)

You can use the macro-recorder of Fiji to record the macro: open and read the CZI-file you just acquired using the Bio-Formats library and subsequently perform an image processing function. In our case we will calculate a maximum-projection of the acquired Z-stack. Finally we will apply a different lookup table (LUT) to display the data.

- **Plugins** ▷ **Macros** ▷ **Record. . .** : opens the macro recorder
- **Plugins** ▷ **BioFormats** ▷ **Bioformats Importer**: select the *.czi you previously acquired
- **Image** ▷ **Stacks** ▷ **Z-project. . .** : calculates a Z-projection of the image stack
- **Image** ▷ **Lookup Tables** ▷ **Fire**: changes the LUT to "Fire"

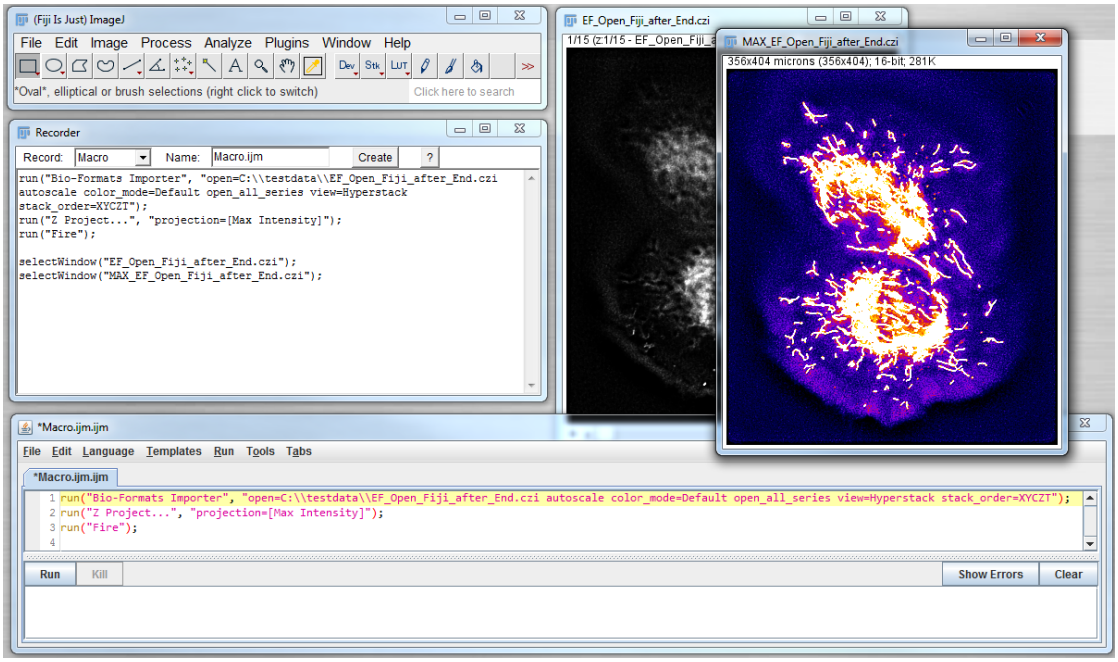

Figure 29: Record a macro in Fiji

In the recorder window you see how the code is added for each operation. In the recorder window click on **Create**. The macro window opens where you can modify and run the macro.

```
1 run("Bio-Formats Importer", "open=C:\\testdata\\EF_Open_Fiji_after_End.czi autoscale color_mode=Default open_all_series
  ↪ view=Hyperstack stack_order=XYCZT");
2 run("Z Project...", "projection=[Max Intensity]");
3 run("Fire");
```

Now you just need to modify the recorded macro so you can pass a filename from the ZEN 3.5 (blue edition) acquisition to the Fiji macro. Therefore we need to remove the fixed filepath and filename with a variable "open=[" + name + "], and make sure that Fiji gets the correct filename (via getArgument). The modified macro looks as follows:

```
1  // read the filename that is passed from ZEN
2  name= getArgument;
3
4  // error message if there was no filename
5  if (name=="") exit("No file selected!");
6
7  // Import the CZI file with Bio-formats
8  run("Bio-Formats Importer", "open=[" + name + "] autoscale color_mode=Default open_all_series view=Hyperstack stack_order=XYCZT"
   );
9
10 // apply a maximum projection
11 run("Z Project...", "projection=[Max Intensity]");
12
13 // change the lookup table
14 run("Fire");
```

- Line 2: Read the filename.

- Line 5: Print an error message in case there was no filename.
- Line 8: Open the CZI-file using Bio-Formats and configure the display options.
- Line 11: Do a Z-projection.
- Line 14: Apply a special LUT.

The Feedback Script

For this example no special image analysis is required, so there is no need to set up an image analysis pipeline. There is also no need for a **PreLoop Script** or **Loop Script**. The **PostLoop Script** contains the following steps:

- Start Fiji when the acquisition is finished.
- Hand over the correct file name.
- Define the Fiji-macro to be executed on the acquired data.

The key to this application is the possibility to start Fiji via the command line with options as shown in Fig. 30. That means that you can also start Fiji via the Experiment Feedback from ZEN 3.5 (blue edition) and transfer the necessary options (in this case the name of the Fiji macro to be executed and the name of the data to be analyzed). You can run the macro you just created on the previously acquired Z-stack from the command line by using the following command structure:

▷ **fiji.exe -macro macroname.ijm filename.czi**

Running this command should start Fiji and apply the macro to the specified dataset.

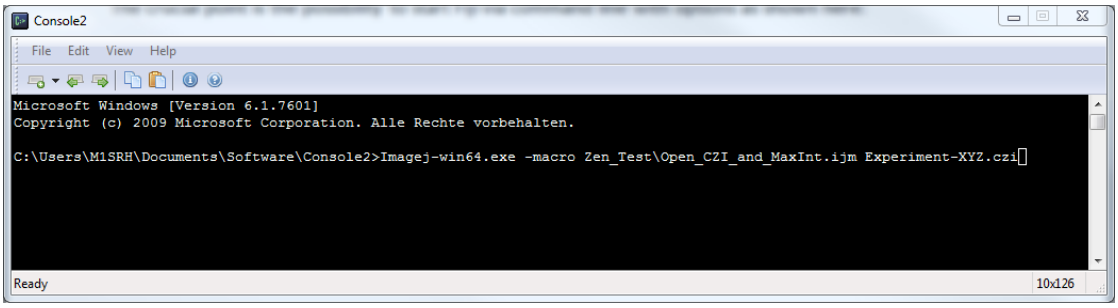

Figure 30: Test the Fiji macro from the command line

When you want to directly start Fiji from ZEN 3.5 (blue edition) you need to pass the macro name as well as the filename from within the Experiment Feedback script. You need to get the filename of the acquired CZI-file from ZEN (via **ZenService.Experiment.ImageFileName)** and then call Fiji.exe with the options **-macro macroname.ijm filename**. Below you see the corresponding script for the feedback experiment.

```
### ------------------- PreScript ----------------------------------------- ###



### ------------------- LoopScript ---------------------------------------- ###



### ------------------- PostScript ---------------------------------------- ###


# get the name of the current image data set
filename = ZenService.Experiment.ImageFileName
# use the absolute path --> this works always
exeloc = "C:\Fiji.app\ImageJ-win64.exe"
 # specify the Fiji macro one wants to use
macro = r'-macro C:\ExperimentFeedback\Fiji\Open_CZI_and_MaxInt.ijm'
# 'glue' together the options
option =  macro + ' ' + filename
# start Fiji, open the data set and execute the macro
ZenService.Xtra.System.ExecuteExternalProgram(exeloc, option)
```

Run the Feedback Experiment and watch the results

Activate the **Experiment Feedback** in the **Acquisition** tab of ZEN 3.5 (blue edition) and start the experiment. After the Z-stack is acquired, Fiji will be started and import the *.CZI-file. It will apply a maximum intensity projection to the Z-stack and display the result with a different LUT. If everything worked fine one should get the result as shown in Fig. 31.

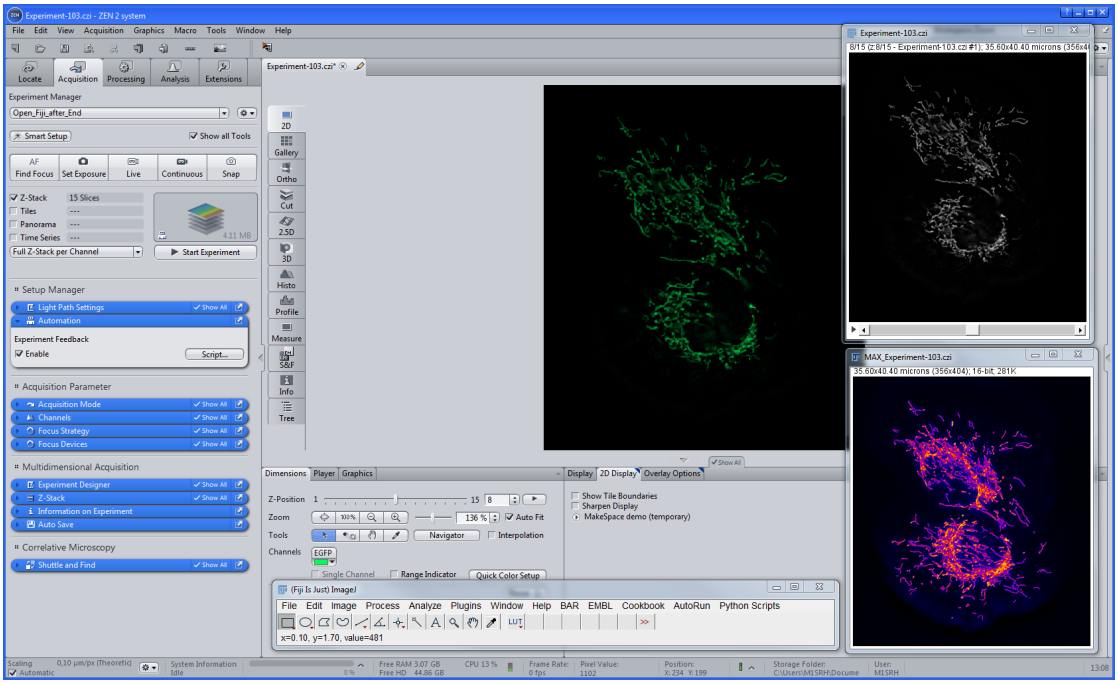

Figure 31: Fiji opens the CZI-file automatically and runs the macro

This example demonstrates how to call an external application (e.g. Fiji), from inside a Feedback Experiment to 'do something'. As an example we imported the *.CZI-file and performed some image processing. However, the macro could also perform other functions such as:

- Image analysis
- Filtering
- 3D stitching
- Tracking
- ...

Of course all those tasks could also be done manually after image acquisition. You can import the *.CZI-file manually in Fiji and work from there on inside the Fiji software. But often it can be a good idea to automate what is possible, especially if the same operations need to be performed over and over a again.

4.5 Jump to next Well

This experiment is basically a variant of example 4.2. Here you will now learn how to stop acquiring images within a specific well when a certain condition is met and then jump to the next well. Such a condition could be the total numbers of objects, e.g. cells, that were already detected. Another application example would be to stop the acquisition when the object number after n tiles is still below a certain threshold, i.e. "the well seems to be empty".

Idea or Task:

- Select multi-well plate (e.g. 96 well plate) as sample carrier
- Define a wellplate experiment with several tiles per well
- Sum up the number of objects per well
- Jump to next well when a certain number of cells per well has been reached

Definition of the Feedback Experiment

The experiment setup is basically the same as in example 4.3 but with several tiles per well. For this example you can use the sample image folder . . . \Testimages\96_well again. It contains 96 single images with cells where the nucleus is stained with DAPI. Set up a Tiles-experiment with the following parameters:

- Positions = 4 wells
- Tiles per well: 25
- Channel = DAPI (or whatever the nuclei are stained with ...)

The experiment used is called EF_05_JumpToNextWell.czexp. Important is to choose the **Synchronized** execution mode to make sure that the image is analyzed and the data is available before acquiring the next frame.

Definition of the Image Analysis Pipeline

Set up an image analysis pipeline that counts the number of cells in each frame. You can use a similar analysis pipeline as described in example 4.1. But here you additionally need to read out the wellID in order to assign the acquired data to the correct well. Be sure to define the following features for **Regions** as they are necessary for the feedback script:

- Regions Count (= number of cells)
- Image Scene Container Name (= well number)
- Image Acquisition Time

Alternatively you can also use the image analysis pipeline EF_Nuclei_Dapi.czias for this example.

The Feedback Script

In the feedback script you need to obtain the number of cells of every frame and sum up the number of cells for all frames in one well acquired so far. When the number exceeds a defined limit (e.g. 2000), jump to the next well (by using ZenService.Actions.JumpToNextContainer()) and reset the counter **cells_per_well** to zero in order to start the count again from zero for the next well. The most important steps are:

1. Get the number of cells per tile (=cpt), the well name (= well) and the tile index (= tile).
2. Sum up the number for every new tile (cells_per_well)
3. Jump to the next well when the number of cells in the well exceeds the limit.
4. Reset the cell counter

```
### -------------------- PreScript ----------------------------------------------- ###



cells_per_well = 0 # total number of cells
logfile = ZenService.Xtra.System.AppendLogLine('Well\tTile\tCells/Tile\tCell/Well')
last_tile = 0
current_well = 1
number_of_wells = 4


### -------------------- LoopScript ---------------------------------------------- ###


# this value is different for every acquired picture !!!
index = ZenService.Analysis.Cells.ImageAcquisitionTime

# get cell number for the current tile
cpt = ZenService.Analysis.Cells.RegionsCount

# get current well name
well = ZenService.Analysis.Cells.ImageSceneContainerName

# get current tile number
tile = ZenService.Experiment.CurrentTileIndex

# reset the counter if a new well is started, (i.e. tile < lasttile)
if tile != last_tile:
    if tile < last_tile:
        cells_per_well = 0

    # add cells from current tile to cell_per_well
    cells_per_well = cells_per_well + cpt

# write data into log file
logfile = ZenService.Xtra.System.AppendLogLine(well+'\t'+str(tile)+'\t'+str(cpt)+'\t'+str(cells_per_well))

# update lasttile
last_tile = tile

# jump to next well if the desired cell number was reached
if (cells_per_well > 2000):
    # if current well is not the last well jump to next well
    if current_well < number_of_wells:
        ZenService.Actions.JumpToNextContainer()

        if last_tile < 25:
            logfile = ZenService.Xtra.System.AppendLogLine('Jumped to next well after ' + str(last_tile) + ' tiles')

        if last_tile == 25:
            logfile = ZenService.Xtra.System.AppendLogLine('Well completed')
    current_well += 1

```

```
54
55      # if current well is last well stop the experiment
56      if current_well == number_of_wells:
57          ZenService.Actions.StopExperiment()
58
59          if last_tile < 25:
60              logfile = ZenService.Xtra.System.AppendLogLine('Jumped to next well after ' + str(last_tile) + ' tiles')
61
62          if last_tile == 25:
63              logfile = ZenService.Xtra.System.AppendLogLine('Well completed')
64
65
66
67
68  ### -------------------- PostScript ------------------------------------------- ###
69
70
71  ZenService.Xtra.System.ExecuteExternalProgram(r"C:\Program Files (x86)\Notepad++\notepad++.exe", logfile)
```

Run the Feedback Experiment and watch the results

Activate **Experiment Feedback** in the **Acquisition** tab and click on **Start Experiment**. In the ZEN (blue Edition) main window you can observe how the tiles are acquired one after another. The original experiment is configured to acquire 25 tiles per well. As you can see from the logfile (Fig. 32) the **JumpToNextContainer** command was called after 10 tiles acquired within in well A1. The accumulated number of detected cells was 2110 and therefore the jump command was called. Inside well B4 the required limit of 2000 cells was reached after 12 tiles and the experiment was finished there.

Experiment-08_Log.txt

```
Well    Tile    Cells/Tile  Cell/Well
A1  1   305.0   305.0
A1  2   242.0   547.0
A1  3   227.0   774.0
A1  4   184.0   958.0
A1  5   196.0   1154.0
A1  6   218.0   1372.0
A1  7   189.0   1561.0
A1  8   195.0   1756.0
A1  9   190.0   1946.0
A1  10  164.0   2110.0
Jumped to next well after 10 tiles
A2  1   163.0   163.0
A2  2   186.0   349.0
A2  3   153.0   502.0
A2  4   192.0   694.0
A2  5   237.0   931.0
A2  6   206.0   1137.0
A2  7   205.0   1342.0
A2  8   192.0   1534.0
A2  9   193.0   1727.0
A2  10  172.0   1899.0
A2  11  154.0   2053.0
Jumped to next well after 11 tiles
A3  1   177.0   177.0
A3  2   176.0   353.0
A3  3   142.0   495.0
A3  4   149.0   644.0
A3  5   186.0   830.0
A3  6   197.0   1027.0
A3  7   206.0   1233.0
A3  8   190.0   1423.0
A3  9   176.0   1599.0
A3  10  157.0   1756.0
A3  11  169.0   1925.0
A3  12  128.0   2053.0
Jumped to next well after 12 tiles
A4  1   182.0   182.0
A4  2   160.0   342.0
A4  3   195.0   537.0
A4  4   160.0   697.0
A4  5   189.0   886.0
A4  6   191.0   1077.0
A4  7   187.0   1264.0
A4  8   193.0   1457.0
A4  9   179.0   1636.0
A4  10  173.0   1809.0
A4  11  157.0   1966.0
A4  12  151.0   2117.0
Jumped to next well after 12 tiles
```

Figure 32: The data logfile shows the "jump" after the object limit was reached.

4.6 Adapt Exposure Time during Acquisition

The main idea of this experiment is to change hardware parameters, in this case the exposure time, during the image acquisition. This could be in response to a certain parameter, but for demonstration we will just increase the exposure time after every frame.

Idea or Task:

- Modify the exposure time while the experiment is running.

Definition of the Feedback Experiment

This example is rather simple. It just illustrates the possibility to adjust the exposure time while the experiment is running. This opens up the possibility to increase or decrease the exposure time or other hardware parameters based on a certain event. For this example you can set up a simple time lapse experiment with 10 cycles.

- Exposure Time = 2 ms
- Time Series = 10 Cycles

The other settings depend on your sample and hardware settings. For proof of concept you can use the sample data in the folder ...\Testimages\10_frames_constant. It contains ten identical images with DAPI-stained nuclei.
The experiment example itself is called EF_06_Brighter_and_Brighter.czexp.

Definition of Image Analysis Pipeline

Set up an image analysis pipeline that detects the cells in each frame. You can use a similar analysis pipeline as described in example 4.2. For this example it is important to define the mean intensity of all cells in the DAPI channel. Be sure to define the following feature for **Regions** as they are necessary for the feedback script:

- **RegionsIntensityMean_DAPI**

The Feedback Script

In this application example the feedback script needs to fulfill the following tasks:

1. Initialize a variable for the exposure time with a certain value

2. Get mean DAPI value of all cells of one frame
3. Write frame number and mean intensity DAPI in the logfile (optional)
4. Set the new exposure time depending on the frame number
5. Use a Python script to display the data (optional)

```
### -------------------- PreScript ----------------------------------------------- ###


from System.Diagnostics import Process

exposure = 2 # exposure time set in the acquisition parameters



import sys
sys.path.append('C:\\Program Files\\Carl Zeiss\\ZEN 2\\ZEN 2 (blue edition)\\IronPython\\Lib')
import os
os.chdir(r'C:\Python_Scripts')


### -------------------- LoopScript ---------------------------------------------- ###


# get current time index and mean intensity DAPI
index = ZenService.Experiment.CurrentTimePointIndex
mean_dapi = ZenService.Analysis.Cells.RegionsIntensityMean_DAPI

#write logfile
logfile = ZenService.Xtra.System.AppendLogLine(str(index) + '\t' + str(mean_dapi))

# calculate amplification factor
index = 1 + index *0.5
# set new exposure time
ZenService.Actions.SetExposureTime(1, index*exposure)


### -------------------- PostScript ---------------------------------------------- ###


# Open logfile in Notepad++
ZenService.Xtra.System.ExecuteExternalProgram(r"C:\Program Files (x86)\Notepad++\notepad++.exe", logfile)

#define filename, external application and python script
filename = ZenService.Experiment.ImageFileName[:-4] + '_Log.txt'
exeloc = 'C:\Program Files\Carl Zeiss\ZeissPython\Py20037.2\env\python.exe'
script = 'display_results_simple_intensity.py'
cmd = script + ' -f ' + filename

## start Python Option  1
app = Process();
app.StartInfo.FileName = exeloc
app.StartInfo.Arguments = cmd
app.Start()

## start Python Option 2
#ZenService.Xtra.System.ExecuteExternalProgram(script, ' -f ' + filename)
```

Python Script for Data Display

You can modify the same Python script used for example 4.1 to create a simple data display. As the structure of the logfile is identical it is just necessary to adapt the labels for the two plots.

Run the Feedback Experiment and watch the results

Start the experiment and use the **Gallery View** to compare the different frames. You will see that the brightness was increased for every frame.

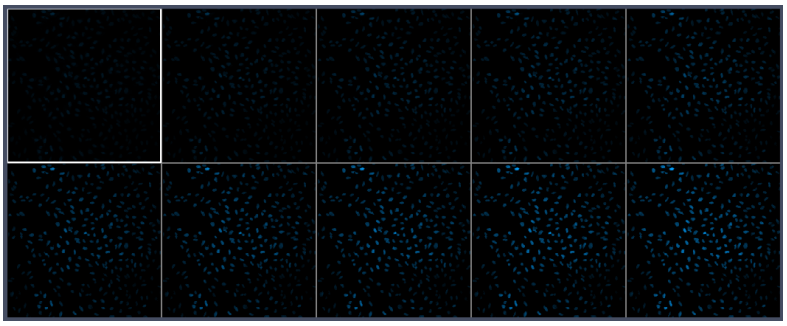

Figure 33: The gallery view shows that the ten acquired frames have increasing intensity.

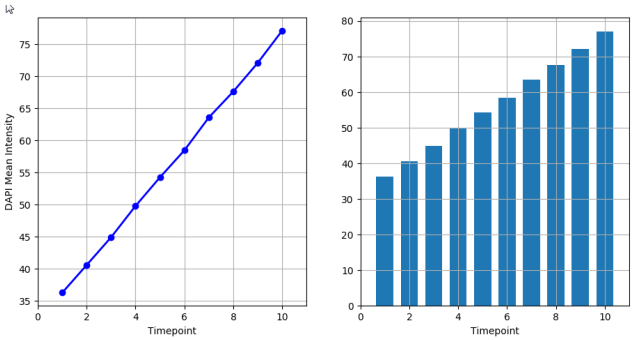

Figure 34: Display of the results with a Python script: frame by frame the Mean Intensity DAPI increases.

4.7 Modify the Blocks of an Experiment

This example demonstrates the use of a feedback script in connection with heterogeneous experiments, where you can define several experiment blocks using the **Experiment Designer**. For more information about how to use the **Experiment Designer** please refer to the ZEN 3.5 (blue edition) user manual.
We will define a multi-block experiment with three blocks. In this example in every block a time series with 10 cycles is acquired. But of course, it could also be completely different experiments in each block. The idea of this example is to show you how to set up a feedback script that allows you to perform online image analysis only for the previously selected blocks (and skip the other blocks).

Idea or Task:

- Define a multi-block experiment
- Do something different for every block inside the experiment

INFO:

The **Experiment Designer** is only available if you have licensed the module **Experiment Designer** and it is activated in the Modules Manager.

Definition of the Feedback Experiment

For this example you can use the sample image folder . . . \Testimages\96_well. It contains 96 single images with cells where the nucleus is stained with DAPI.
As experiment you can define a simple time lapse experiment, which is identical for all three blocks.

- Time cycles = 10
- Interval = 1 sek

You need to make sure that the checkbox **Experiment Designer** is activated on the **Acquisition** tab in ZEN 3.5 (blue edition). This opens the **Experiment Designer** toolbox. Here you can duplicate the experiment two times to create in total three identical acquisition blocks. (The experiment is called **EF_07_Cell_Count_Blocks.czexp**).

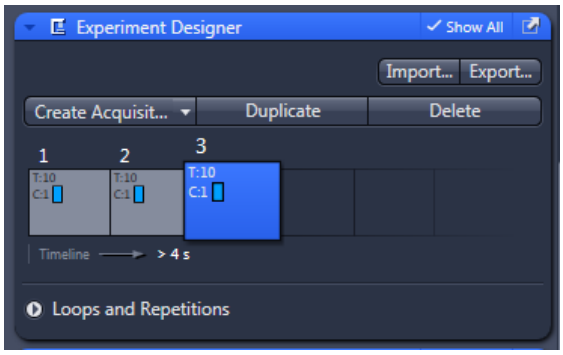

Figure 35: Setup of the multi-block experiment.

Definition of the Image Analysis Pipeline

You can use a similar analysis pipeline as for the example 4.1 in order to count the number of cells. You can define the the number of cells as the parameter to be analyzed:

- Regions Count (= number of cells)

(Alternatively you can use the image analysis pipeline EF_Nuclei_DAPI.czias).

Creation of the Feedback Script

In the **PreLoop Script** you need to define for which blocks a certain task or action (in our case an online image analysis) should be performed. You can also write an header for the logfile.

In the **LoopScript** you need to check the current block index (via the command Zen.Service.Experiment.CurrentBl
With an ***if***-condition you can define what should happen with a certain block and execute one part of the feedback script or the other. For example, for block 1 and 3 you want to perform the online image analysis and get the number of cells and add this information to the logfile. For block 2 you could just do nothing, play a sound or write an entry into the logfile.

In the **PostLoop Script** you can call an application to open and display the logfile.

```
1  ### ------------------- PreScript ----------------------------------------- ###
2
3
4  # define blocks to be analyzed
5  blocks2do = [1,3]
6
7  # blocks where nothing happens
8  blocksnot2do = [2]
```

```
9
10  # header for logfile
11  logfile = ZenService.Xtra.System.AppendLogLine('Block\tFrame\tCells')
12
13
14  ### -------------------- LoopScript ------------------------------------------- ###
15
16
17  block = ZenService.Experiment.CurrentBlockIndex
18  frame = ZenService.Experiment.CurrentTimePointIndex
19
20  if block in blocks2do:
21      # get current frame number, number of cells
22      cn = ZenService.Analysis.Cells.RegionsCount
23      # write into logfile
24      logfile = ZenService.Xtra.System.AppendLogLine(str(block) + '\t' + str(frame) + '\t' + str(cn) )
25
26
27  elif block in blocksnot2do:
28      # write into logfile
29      logfile = ZenService.Xtra.System.AppendLogLine(str(block) + '\t' + str(frame) + '\t' +  'skipped analysis')
30      # play sound
31      ZenService.Xtra.System.PlaySound()
32
33
34
35  ### -------------------- PostScript ------------------------------------------- ###
36
37
38  # Open logfile in Notepad++
39  ZenService.Xtra.System.ExecuteExternalProgram(r"C:\Program Files (x86)\Notepad++\notepad++.exe", logfile)
```

Run the Feedback Experiment and watch the results

Activate experiment feedback in the acquisition tab and click on **Start Experiment**. In the ZEN 3.5 (blue edition) main window you can observe how the images of the three blocks of the multi-block experiment are acquired. For block 2 instead of running the image analysis a sound is played. After the acquisition is completed the logfile opens. The resulting TXT file only contains cell numbers for the blocks 1 and 3. For each frame in block 2 the logfile contains the information that the analysis was skipped.

New-19_Log.txt

	imageBlock	Frame	Cells
1			
2	1	1	306.0
3	1	2	244.0
4	1	3	225.0
5	1	4	184.0
6	1	5	198.0
7	1	6	220.0
8	1	7	192.0
9	1	8	195.0
10	1	9	190.0
11	1	10	163.0
12	2	1	skipped analysis
13	2	2	skipped analysis
14	2	3	skipped analysis
15	2	4	skipped analysis
16	2	5	skipped analysis
17	2	6	skipped analysis
18	2	7	skipped analysis
19	2	8	skipped analysis
20	2	9	skipped analysis
21	2	10	skipped analysis
22	3	1	307.0
23	3	2	242.0
24	3	3	224.0
25	3	4	182.0
26	3	5	196.0
27	3	6	217.0
28	3	7	193.0
29	3	8	194.0
30	3	9	193.0
31	3	10	163.0
32			

Figure 36: The resulting data logfile only contains cell numbers for block 1 and 3

4.8 Time lapse per Z-Plane

This example illustrates the possibility to use the Experiment Feedback together with the **Experiment Designer** to enable "special" experiments, which could otherwise not be realized. The aim of this example is to acquire a time-series at every position of a Z-stack. The frontend of ZEN 3.5 (blue edition) does not allow to you acquire a complete time lapse for every Z-plane inside a Z-stack directly. So one way to solve this problem is to move the focus position while repeating the time lapse itself. Applying this concept will allow you to run also other nested experiments within ZEN 3.5 (blue edition). For more information about how to use the **Experiment Designer** please refer to the ZEN 3.5 (blue edition) user manual.

Idea or Task:

- Define a time-lapse acquisition.
- Use the **Experiment Designer** to define the number of repetitions for the loop
- Use the **Experiment Feedback** to change the Z-position for every repetition

This way a time-lapse experiment is performed on every position of the Z-stack.

INFO:

The **Experiment Designer** is only available if you have licensed the module **Experiment Designer** and it is activated in the **Modules Manager**.

Definition of the Feedback Experiment

For setting up the experiment you can use the images in the folder ...\testimages\HeLa_Z-Stack. It contains a Z-stack with 15 images of HeLa cells where the mitochondria were labeled with Mitotracker Red. This script does not require an image analysis pipeline. Important however, is the correct use of the **Experiment Designer**. You need to make sure that the checkbox **Experiment Designer** is activated on the **Acquisition** tab in ZEN 3.5 (blue edition). This opens the **Experiment Designer** toolbox.

First set up a simple time-lapse experiment:

- Time cycles = 5

In the **Experiment Designer** you can then set the number of repetitions of the Z-stack (if necessary check the **Show All** checkbox of the **Experiment Designer** toolbox). The number of repetitions represents the number of planes of the Z-stack. In the example the number of repetitions is three, therefore the Z-stack later will have three layers.

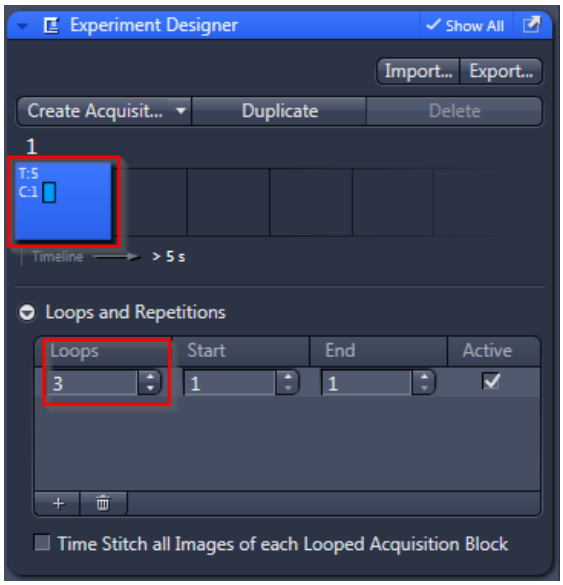

Figure 37: In the **Experiment Designer** toolbox set the number of repetitions.

For this kind of nested experiment the built-in Z-stack tool cannot be used, however you can use it to extract the necessary parameters to define the Z-stack in the feedback script manually. (You can also use the corresponding experiment which is called **EF_08_Time_Lapse_zStack.czexp**).

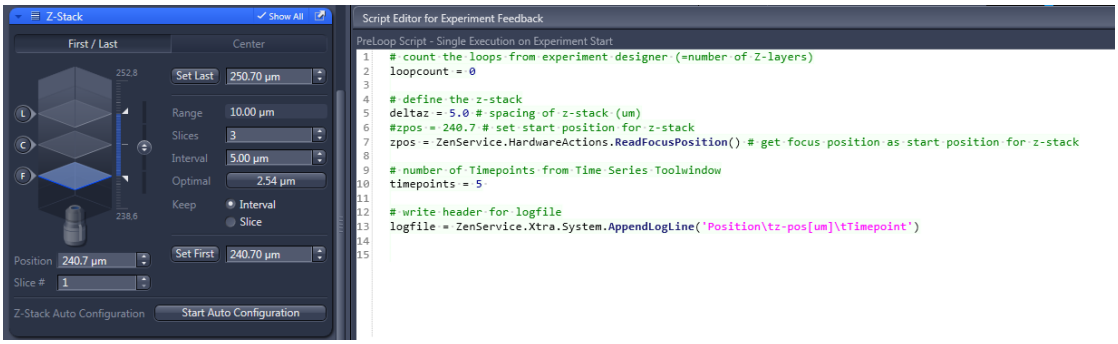

Figure 38: Read out the parameters necessary for the definition of the feedback script in the Z-**Stack** toolbox.

The Feedback Script

The basic steps of the feedback script are the following:

- Define the parameters for the Z-stack (number of planes, interval and start position)
- After a repetition is finished (i.e. the time-lapse is acquired) move to the next Z-position

- Write a log file for testing (optional).

```
### -------------------- PreScript ----------------------------------------------- ###


# count the loops from experiment designer (=number of Z-layers)
loopcount = 0

# define the z-stack
deltaz = 5.0 # spacing of z-stack (um)
#zpos = 240.7 # set start position for z-stack
zpos = ZenService.HardwareActions.ReadFocusPosition() # get focus position as start position for z-stack

# number of Timepoints from Time Series Toolwindow
timepoints = 5

# write header for logfile
logfile = ZenService.Xtra.System.AppendLogLine('Position\tz-pos[um]\tTimepoint')




### -------------------- LoopScript ---------------------------------------------- ###


# current time point
timepoint = ZenService.Experiment.CurrentTimePointIndex

# create logfile
logfile = ZenService.Xtra.System.AppendLogLine(str(loopcount+1) + '\t' + str(zpos) + '\t' + str(timepoint))

if (timepoint == timepoints):
    # increase loop counter
    loopcount = loopcount + 1
    #calculate new z-position for next timelapse
    zpos = zpos + deltaz
    # set new focus position
    ZenService.HardwareActions.SetFocusPosition(zpos)



### -------------------- PostScript ---------------------------------------------- ###


# Open logfile in Notepad++
ZenService.Xtra.System.ExecuteExternalProgram(r"C:\Program Files (x86)\Notepad++\notepad++.exe", logfile)
```

In the **PreLoop Script** you have to define the properties of the Z-stack (Z-position and interval) manually inside the feedback script (see Fig. 38). Remember, that the number of slices in the Z-stack is already defined by the number of repetitions of the loop. Instead of setting the start position in the feedback script to the value in the Z-stack toolwindow, you can also read out the current focus position (ZenService.HardwareActions.ReadFocusPosition()) to use the current focus position as start position for the Z-stack. You also need to initialize the loop counter.

In the **Loop Script** you need to make sure to move to the next Z-plane only after the desired number of time points was acquired. For this you can use the time point index of the time-lapse (ZenService.Experiment.CurrentTimePointIndex). If this equals 5 the complete time-lapse at the current Z-position was acquired and the objective needs to move to the next Z-position (ZenService.HardwareActions.SetFocusPosition()).

Run the Feedback Experiment and watch the results

Before starting the experiment be sure to have the **Experiment Feedback** and the **Experiment Designer** enabled. The acquired images are collected in a CZMBI-file, that consists

of the three CZI-files for the different acquisition blocks (Fig. 39). Each CZI-file consists of the time-lapse image data from one Z-position. Therefore the block slider corresponds to the Z-dimension for this special experiment.

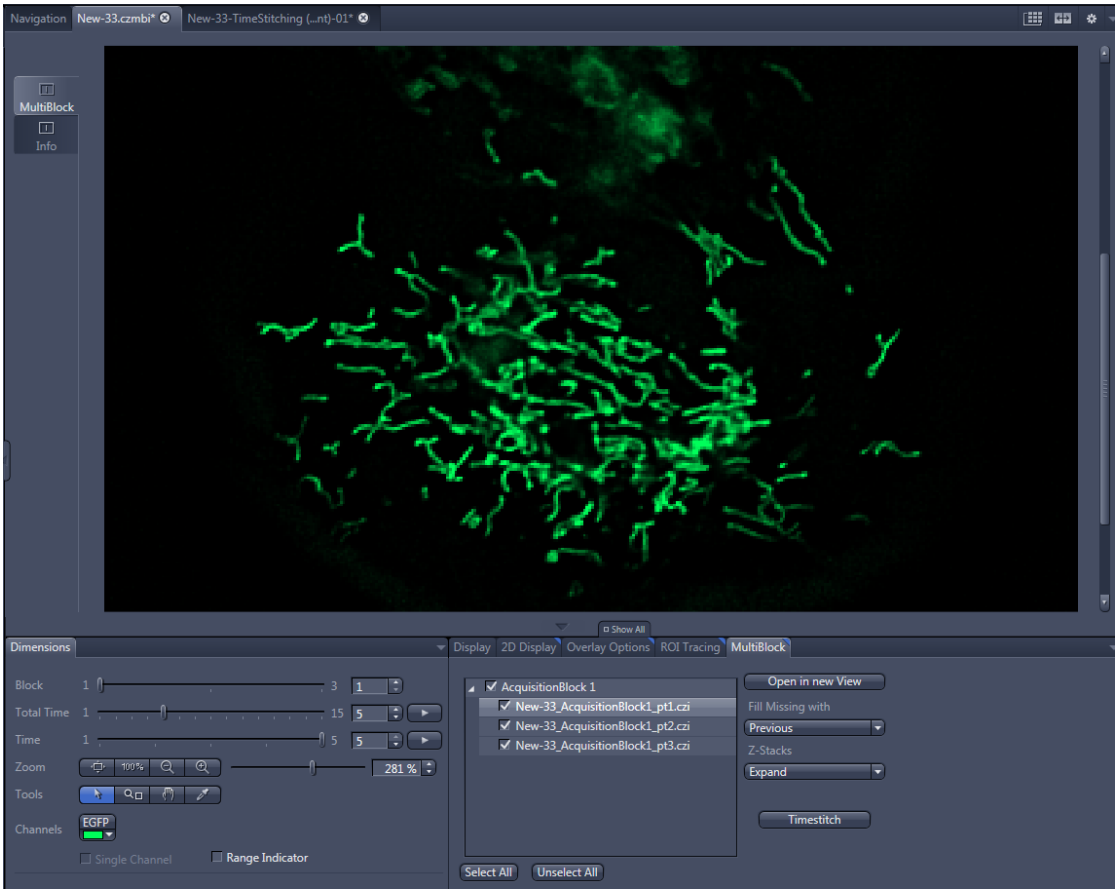

Figure 39: The resulting CZMBI consists of three acquisition blocks.

1. 1st block $\widehat{=}$ 1st Z-plane at $z = 240\ \mu m$, includes 5 time points.
2. 2nd block $\widehat{=}$ 2nd Z-plane at $z = 245\ \mu m$, includes 5 time points.
3. 3rd block $\widehat{=}$ 3rd Z-plane at $z = 250;\mu m$, includes 5 time points.

After time stitching of the three blocks the gallery displays the final result: five images at different times for each Z-position (Fig. 40). The data was produced using the TestCam feature, simulating a Z-stack. Therefore the images acquired for every block are the same. If you have also written a logfile with the data (Fig. 41), you can use it to cross-check if the experiment produced the results you expected.

Figure 40: After **TimeStitching** the **Gallery View** shows the correct result.

New-24_Log.txt

	Position	z-pos[um]	Timepoint
1			
2	1	240.7	1
3	1	240.7	2
4	1	240.7	3
5	1	240.7	4
6	1	240.7	5
7	2	245.7	1
8	2	245.7	2
9	2	245.7	3
10	2	245.7	4
11	2	245.7	5
12	3	250.7	1
13	3	250.7	2
14	3	250.7	3
15	3	250.7	4
16	3	250.7	5
17			

Figure 41: The logfile containing position (= loopcount+1), Z-position in μm and timepoint of the time-lapse.

4.9 Automatic Event Detection

This example demonstrates how to set up a feedback experiment that reacts on a change of intensity / number of detected events of automatically detected cells. Of course the intensity parameter can easily be exchanged for something else. For demonstration you can use the sample data set with Fluo-4 stained cells. In this time-lapse experiment different cells are active at different time points, so the number of active cells (i.e. cells that are brighter than a certain intensity threshold) changes over time. The goal in this example is to set up a feedback experiment that performs a certain action after a certain event occurs, in this case, if the number of detected cells per frame increases/decreases.

Idea or Task:

- Define a time-lapse acquisition.
- Setup an image analysis pipeline to detect the events (e.g. number of active cells).
- Within the feedback script define the trigger-event (e.g. change in number of active cells).
- Define which action should be taken when the trigger-event occurs.

This kind of experiment feedback allows you to set up an acquisition that reacts dynamically on certain events and to adapt the acquisition accordingly. This is especially useful in cases where you cannot predict when an event will take place. For example, you could set up a time-lapse experiment that changes the acquisition interval when a certain event occurs, so to acquire images in large intervals as long as nothing of interest is happening and then increase the number of frames per second after an activation event.

The Feedback Experiment

For simulation of the experiment you can use the sample image folder **...\Testimages\ Jurkat_Activation_Detection.** The folder contains 70 single TIFF images with Jurkat cells stained with Fluo-4. Just define a single channel and test if the sample camera gives the correct output. Define an time-lapse experiment with the following parameters:

- Cycles = 70
- Time Interval = 700 ms
- Channel 1 = Fluo-4

The corresponding experiment is called **EF_09_Jurkat_Activation.czexp**.

Definition of the Image Analysis Pipeline

Set up an image analysis pipeline that uses a simple threshold to detect active (bright) cells. For the feedback experiment the number of cells of a frame is required. Be sure to define the following features for **Regions**:

- Count
- Image Index Time

for later use in the feedback experiment. (Alternatively you can also use the analysis setting **EF_Jurkat_Cells.czias**).

The Feedback Script

The main tasks for this experiment feedback script can be described as follows:

- Initialize the required variables
- Check the number of detected objects every frame
- If the number increased or decreased: do "something"
- Write a log file for testing (optional).

Initialize all required variables in the **PreLoop Script**.

In the **Loop Script** you can get the number of objects detected by the EF_Jurkat_cells image analysis pipeline for every frame. Check if the number has changed and perform an action, depending on the outcome. In this example two different soundfiles will be played, depending on if the number has increased or decreased. This is just a placeholder for a more meaningful operation.

In the **PostLoop Script** you can start a text editor to display the logfile and run a Python script to display the results graphically.

```
### ------------------- PreScript ----------------------------------------- ###


from System.Diagnostics import Process
lastindex = 0
cn_last = 0
soundfile1 = r'C:\ExperimentFeedback\SoundFiles\PsychoScream.wav'
soundfile2 = r'C:\ExperimentFeedback\SoundFiles\YEAH.WAV'

import sys
sys.path.append('C:\\Program Files\\Carl Zeiss\\ZEN 2\\ZEN 2 (blue edition)\\IronPython\\Lib')
import os
os.chdir(r'C:\Python_Scripts')


### ------------------- LoopScript ---------------------------------------- ###


# get parameters
frame = ZenService.Analysis.Cells.ImageIndexTime
cn = ZenService.Analysis.Cells.RegionsCount
```

```

# calculate the change of cell numbers
delta = cn - cn_last

# write to log file (optional)
logfile = ZenService.Xtra.System.AppendLogLine(str(frame)+'\t'+str(cn)+'\t'+str(delta))
cn_last = cn

# check if the number of active cells has changed

# if active cell number has increased, play soundfile 1
if (delta > 0):
    ZenService.Xtra.System.PlaySound(soundfile2)

# if actice cell number has decreased, play soundfile 2
elif (delta < 0):
    ZenService.Xtra.System.PlaySound(soundfile1)



### -------------------- PostScript ------------------------------------------- ###


ZenService.Xtra.System.ExecuteExternalProgram(r"C:\Program Files (x86)\Notepad++\notepad++.exe", logfile)

#define filename, external application and python script
filename = ZenService.Experiment.ImageFileName[:-4] + '_Log.txt'
exeloc = 'C:\Program Files\Carl Zeiss\ZeissPython\Py20037.2\env\python.exe'
script = "display_jurkat.py"
cmd = script + ' -f ' + filename

## start Python Option  1
app = Process();
app.StartInfo.FileName = exeloc
app.StartInfo.Arguments = cmd
app.Start()

## start Python Option 2
#ZenService.Xtra.System.ExecuteExternalProgram(script, ' -f ' + filename)
```

Python Script for the Data Display

Run a Python script that displays the data (optional). The Python script shown below displays the data of the logfile and automatically saves a *.PNG file in the same folder as the logfile.

```
import sys
import os
env = os.environ
env.update({'QT_API':'pyside'})
# To ensure that only libraries that we have control of are used
env['PATH'] = os.path.join(os.path.dirname(sys.executable), 'Library', 'bin')
import numpy as np
import optparse
import matplotlib.pyplot as plt

# configure parsing option for command line usage
parser = optparse.OptionParser()

parser.add_option('-f', '--file',
    action="store", dest="filename",
    help="query string", default="spam")

parser.add_option('-b', '--block',
    action="store", dest="block",
    help="set to False for non interactive behaviour", default="True")

# read command line arguments
options, args = parser.parse_args()

print ('Filename:', options.filename)
savename = options.filename[:-4] + '.png'
print ('Savename: ', savename)

block = True
if options.block == 'False':
        block = False
```

```
32
33  # load data
34  data = np.loadtxt(options.filename, delimiter='\t')
35  # create figure
36  figure = plt.figure(figsize=(12,6), dpi=100)
37  ax1 = figure.add_subplot(211)
38  ax2 = figure.add_subplot(212)
39
40  # create subplot 1
41  ax1.bar(data[:,0],data[:,1],width=0.8, bottom=0)
42  ax1.grid(True)
43  ax1.set_xlim([0,data[:,0].max() + 1])
44  ax1.set_ylim([0,data[:,1].max() + 0.5])
45  ax1.set_ylabel('Cells Active')
46
47  # create subplot 2
48  ax2.bar(data[:,0],data[:,2], width=0.8, bottom=0, color = 'red')
49  ax2.set_xlim([0,data[:,0].max() + 1])
50  ax2.set_ylim([data[:,2].min()-0.5,data[:,2].max()+0.5])
51  ax2.set_xlabel('Frame Number')
52  ax2.set_ylabel('Delta')
53  ax2.grid(True)
54  figure.subplots_adjust(left=0.10, bottom=0.12, right=0.95, top=0.95,wspace=0.10, hspace=0.20)
55
56  # save figure
57  plt.savefig(savename)
58
59  # show graph
60  plt.show(block=block)
```

Running the Feedback Experiment

Activate experiment feedback in the **Acquisition** tab and click on **Start Experiment**. As soon as a new cell "lights up" (i.e. the number of detected cells in the current frame increases compared to the previous frame) you will hear soundfile 1 ("Yeah"). When the intensity inside a cell drops below the threshold and therefore the number of detected cells in the current frame decreases compared to the previous frame, you will hear a different sound file ("Scream").

The logfile (Fig. 42 contains the frame number, the number of detected cells in each frame and the change of detected cells compared to the previous frame. Note that the number of detected cells depends on how you defined the threshold in the image analysis pipeline.

Experiment-111_Log.txt

```
1.0 8.0 8.0
2.0 7.0 -1.0
3.0 10.0    3.0
4.0 10.0    0.0
5.0 10.0    0.0
6.0 11.0    1.0
7.0 10.0    -1.0
8.0 10.0    0.0
9.0 10.0    0.0
10.0    10.0    0.0
11.0    10.0    0.0
12.0    10.0    0.0
13.0    10.0    0.0
14.0    10.0    0.0
15.0    10.0    0.0
16.0    11.0    1.0
17.0    12.0    1.0
18.0    12.0    0.0
19.0    12.0    0.0
```

Figure 42: The logfile contains Frame number / Number of detected cells / Change of the number of active cells with respect to previous frame

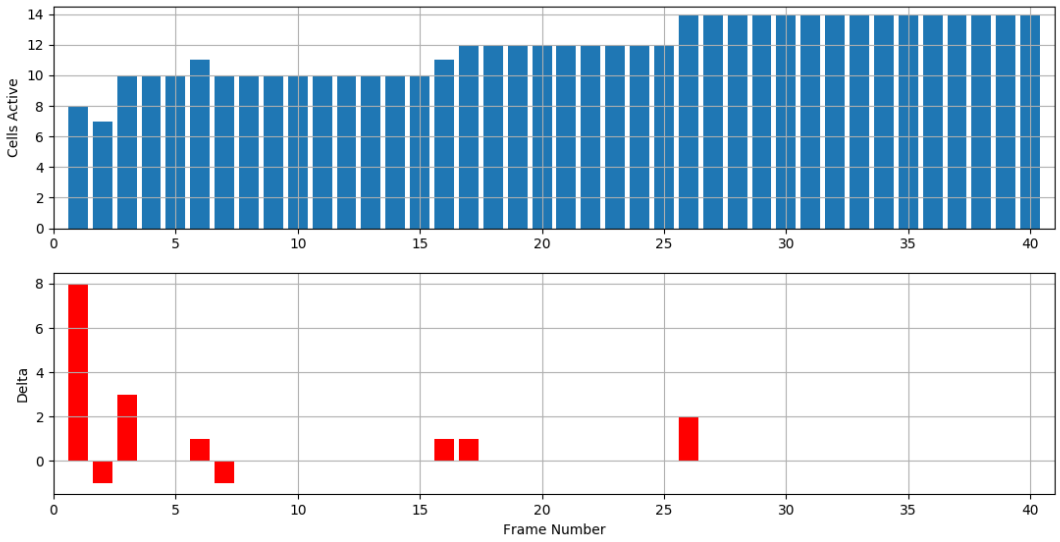

Figure 43: The data from the logfile displayed as a graph.

4.10 Online Dynamics

This advanced example illustrates the possibility to use the online image analysis to display parameters from the image analysis online combined with automated object detection.

Idea or Task:

- Define an time lapse acquisition.
- Use the ZEN 3.5 (blue edition) image analysis to auto-detect the cells
- Get the parameters from the analysis, do some math and plot the data online.

Definition of the Feedback Experiment

For this example you can use the sample image folder **...\Testimages\Calcium_340-380_100frames**. This folder contains 100 single TIFF images with Fura2-stained cells. Fura-2 is a dye used for Calcium imaging. It changes its emission spectrum upon Calcium-binding. Therefore the data set contains two channels - one for unbound (F2) and one for bound (F2C) Fura-2. During this time-lapse experiment some event induces the release of Calcium, changing the relative intensity detected of the two channels over time (Fig. 44).

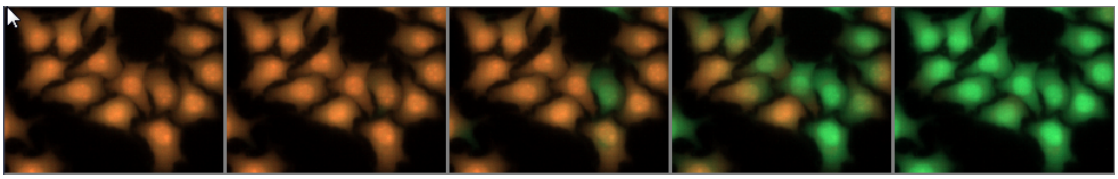

Figure 44: Change of the ratio between F2 and F2C.

The idea behind this example is to measure the intensities of the two channels for every detected cell during acquisition and calculate the ratio of the two mean intensities. The data will be plotted online (using a Python script), so it is possible to observe the calcium-wave during image acquisition. To setup the acquisition, define a two-channel time series experiment with the following parameters:

- Cycles = 100
- Time Interval = 500 ms
- Channel 1 = Fura-2 with Calcium (F2C)
- Channel 2 = Fura-2 wihout Calcium (F2)

Definition of the Image Analysis Pipeline

Set up an image analysis pipeline that uses a simple threshold to detect the cells (use the F2C channel for object detection). For the feedback experiment the number of cells of a frame is required. Be sure to define the following features for **Regions**:

- **Image Index Time**

and the following parameters for **Region** (i.e. on single-cell level)

- **Intensity Mean Value of channel 'F2'**
- **Intensity Mean Value of channel 'F2C'**

for later use in the feedback experiment. (Alternatively you can also use the analysis pipeline **EF_Fura.czias**.)

The Feedback Script

The main tasks for this script can be described as follows:

- Initialize the required variables.
- Create a function that divides the two intensities: as the values are part of an array, it is necessary to define function that divides the entries element-wise.
- Start the external python script for the online display.
- Get the intensities of all cells as arrays and do the math.
- Write data to the log file.

For further information also read the comments in the script. The corresponding experiment is called **EF_10_Ratio_Feedback_Online.czexp**.

```
### -------------------- PreScript ----------------------------------------- ###


from System import Array
from System.Diagnostics import Process


import sys
sys.path.append('C:\\Program Files\\Carl Zeiss\\ZEN 2\\ZEN 2 (blue edition)\\IronPython\\Lib')
import os
os.chdir(r'C:\Python_Scripts')


# calculate ratio
def ArrayDiv(a, b):
    out = Array.CreateInstance(float, len(a))
    outstr = ''
    for i in range(0, len(a)):
        out[i] = a[i] / b[i]
        outstr = outstr + str(round(out[i], 2)) + '\t'
    return out, outstr

filename = ZenService.Experiment.ImageFileName[:-4] + '_Log.txt'
exeloc = 'C:\Program Files\Carl Zeiss\ZeissPython\Py20037.2\env\python.exe'
script = 'dynamic_MeanROI_Cells.py'
cmd = script + ' -f ' + filename
```

```
27
28  ## start Python Option 1
29  app = Process();
30  app.StartInfo.FileName = exeloc
31  app.StartInfo.Arguments = cmd
32  app.Start()
33
34  ## start Python Option 2
35  #ZenService.Xtra.System.ExecuteExternalProgram(script, ' -f ' + filename)
36
37
38
39
40
41
42  ### -------------------- LoopScript ------------------------------------------- ###
43
44
45  frame = ZenService.Analysis.Cells.ImageIndexTime
46
47  # get intensities for all cells
48  int340 = ZenService.Analysis.Cell.IntensityMean_F2
49  int380 = ZenService.Analysis.Cell.IntensityMean_F2C
50
51  # calculate intensity ratio
52  ratio, ratiostr = ArrayDiv(int380, int340)
53
54  logfile = ZenService.Xtra.System.AppendLogLine(str(frame) + '\t' + ratiostr)
55
56
57  ### -------------------- PostScript ------------------------------------------- ###
58
59
60  ZenService.Xtra.System.ExecuteExternalProgram(r"C:\Program Files (x86)\Notepad++\notepad++.exe", logfile)
```

Python Script for Data Display

The python script uses the matplotlib.animation function for live-updating the graph. It will be called every 500 ms (line 49: interval = 500). It reads the data and plots a new graph. The plot is saved as an PNG-file.

```
1   # -*- coding: utf-8 -*-
2   """
3   Created on Thu Feb 15 15:23:07 2018
4
5   @author: M1MALANG
6   """
7   import sys
8   import os
9   env = os.environ
10  env.update({'QT_API':'pyside'})
11  # To ensure that only libraries that we have control of are used
12  env['PATH'] = os.path.join(os.path.dirname(sys.executable), 'Library', 'bin')
13
14  import matplotlib.pyplot as plt
15  import matplotlib.animation as animation
16  from matplotlib import style
17  import optparse
18  import numpy as np
19
20  # configure parsing option for command line usage
21  parser = optparse.OptionParser()
22
23  parser.add_option('-f', '--file',
24      action="store", dest="filename",
25      help="query string", default="No filename passed")
26
27  parser.add_option('-b', '--block',
28      action="store", dest="block",
29      help="set to False for non interactive behaviour", default="True")
30
31  # read command line arguments
32  options, args = parser.parse_args()
33  savename = options.filename[:-4] + '.png'
34  print('Filename: ', options.filename)
35  print('Savename: ', savename)
36
```

```
block = True
if options.block == 'False':
        block = False

# define plot layout
style.use('fivethirtyeight')
fig = plt.figure(figsize=(10,8))
ax1 = fig.add_subplot(1,1,1)
plt.title('Ratio CH1/CH2')
plt.xlabel('Frame Nr.')
plt.ylabel('Ratio')


def animate(i):

    try:
        graph_data = np.genfromtxt(options.filename, dtype=float, invalid_raise=False, delimiter='\t')
        ax1.plot(graph_data[:,0], graph_data[:,1:], '-', lw=2)
        #save plot
        plt.savefig(savename)
    except:
        print('No file loaded')


ani = animation.FuncAnimation(fig, animate, interval=500, repeat=False)
plt.show(block=block)
plt.pause(0.1)
```

Run the Feedback Experiment and watch the results

Activate experiment feedback in the **Acquisition** tab and click on **Start Experiment**. During the experiment the online display will build up. The colored curves are the ratios for different cells. The final image contains the plots for all 100 frames.

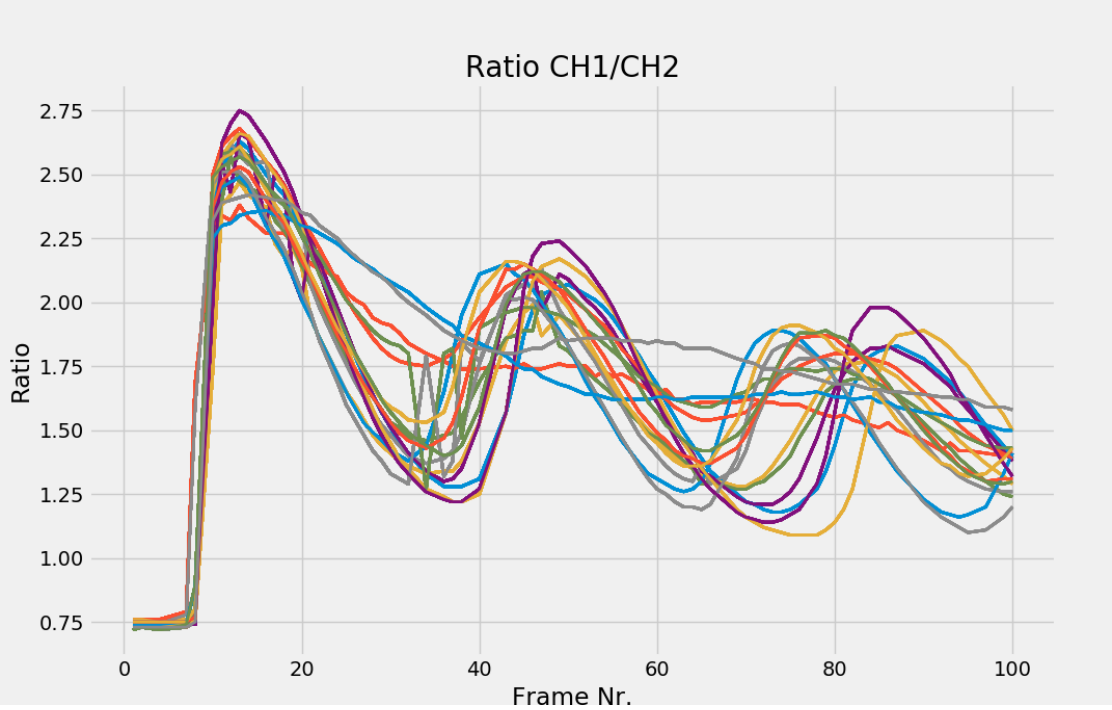

Figure 45: The resulting online data display showing the intensity ratio.

4.11 Online Scratch Assay

This example illustrates how to use the **Feedback Experiment** to run an online analysis of a scratch assay. A scratch assay is a tool to measure cell migration (e.g. for investigating wound healing). For the assay, a scratch is made through a layer of cells and it is observed how fast the cells migrate back into the scratch area. Here you need to setup an online analysis that monitors the area of the scratch of each acquired image frame. As the cells migrate, the scratch area is reduced over time (Fig. 46). The general idea of this example is to set up a time-lapse image acquisition and an image analysis pipeline that detects the scratch area. An online Python plot allows you to observe how the scratch area decreases over time during data acquisition.

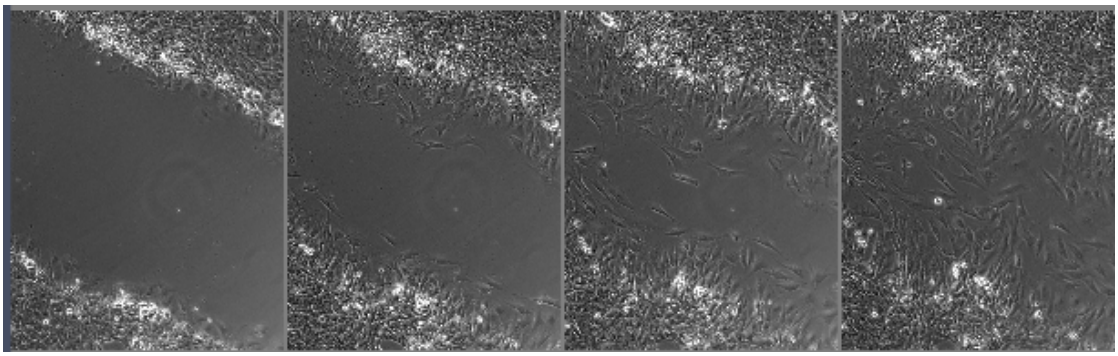

Figure 46: Four different time-points taken from the time-lapse experiment. The cells migrate into the scratch area over time.

Idea or Task:

- Set up a time-lapse image acquisition.
- Use ZEN 3.5 (blue edition) image analysis to detect the scratch and measure the scratch area.
- Plot the data (i.e. the change of scratch area over time) online.

Definition of the Feedback Experiment

For this example you can use the image data in the sample image folder **...\Testimages\Scratch_Assay**. The folder contains 32 phase-contrast images of a scratch assay that allow you to simulate a real time-lapse experiment. Define a single-channel time series experiment to simulate the measurement with the following parameters:

- Cycles = 32
- Time Interval = 1000 ms
- Channel 1 = Brightfield (etc.)

Definition of the Image Analysis Pipeline

Set up an image analysis pipeline that automatically detects the scratch area. For this case the segmentation algorithm SegmenterClassVariance gives good results. Choose this segmentation algorithm in the 3rd step of the image analysis wizard by clicking "Select" and choosing "From Image Channel" as segmentation source and "SegmenterClassVariance" as segmenter. For the feedback experiment the area of the scratch and the scratch area in percent are required. Be sure to define the following features for **Regions**:

- **Area**
- **Area Percentage**
- **Image Index Time**

for later use in the feedback experiment. (Alternatively you can also use the analysis pipeline **EF_Scratch_Assay.czias**).

The Feedback Script

The main tasks for this script can be described as follows:

- Define the required parameters for the online plotting.
- Specify the path to the Python script used for online plotting.
- Start the Python script.
- Get the timepoint, the area and the area percentage values.
- Write data to the log file.

In the **PreLoop Script** you need to define and start the Python script which will be used for online data display. In the **Loop Script** you need to obtain the features for plotting from the online analysis: i.e. the timepoint, the scratch area and the percentage of the scratch area and write these features in the logfile. The logfile will be continuously monitored for changes by the Python script. In the **PostLoop Script** you can start an editor to display the logfile (optional). The corresponding experiment is called **EF_11_Scratch_Assay_dynamic.czexp**.

```
### ------------------- PreScript ----------------------------------------- ###


from System.Diagnostics import Process

import sys
sys.path.append('C:\\Program Files\\Carl Zeiss\\ZEN 2\\ZEN 2 (blue edition)\\IronPython\\Lib')
import os
os.chdir(r'C:\Python_Scripts')

#define filename, external application and python script
filename = ZenService.Experiment.ImageFileName[:-4] + '_Log.txt'
exeloc = 'C:\Program Files\Carl Zeiss\ZeissPython\Py20037.2\env\python.exe'
script = 'EF_ScratchAssay.py'
cmd = script + ' -f ' + filename
ZenService.Xtra.System.WriteDebugOutput(str(filename))

```

```
## start Python Option 1
app = Process();
app.StartInfo.FileName = exeloc
app.StartInfo.Arguments = cmd
app.Start()

## start Python Option 2
#ZenService.Xtra.System.ExecuteExternalProgram(script, ' -f ' + filename)




### -------------------- LoopScript ------------------------------------------- ###


# get the current well name, column idex, row index and position index
frame = ZenService.Experiment.CurrentTimePointIndex

# get area parameters for the scratchnumber of cells from current image
area_t = ZenService.Analysis.Scratch.RegionsArea
area_p = ZenService.Analysis.Scratch.RegionsAreaPercentage

# create logfile
logfile = ZenService.Xtra.System.AppendLogLine(str(frame)+'\t'+str(area_t) + '\t'+ str(area_p))


### -------------------- PostScript ------------------------------------------- ###


# open logfile
ZenService.Xtra.System.ExecuteExternalProgram(r"C:\Program Files (x86)\Notepad++\notepad++.exe", logfile)
```

Python Script for Data Display

The python script uses the matplotlib.animation function for live-updating the graph. It will be called every second (line 62: interval = 1000) and reads the data from the logfile and plots a new graph. The plot is saved as a PNG file.

```
# -*- coding: utf-8 -*-
"""
Created on Wed Feb  7 13:51:15 2018

@author: M1MALANG
"""
import sys
import os
env = os.environ
env.update({'QT_API':'pyside'})
# To ensure that only libraries that we have control of are used
env['PATH'] = os.path.join(os.path.dirname(sys.executable), 'Library', 'bin')

import matplotlib.pyplot as plt
import matplotlib.animation as animation
from matplotlib import style
import optparse
import numpy as np


# configure parsing option for command line usage
parser = optparse.OptionParser()

parser.add_option('-f', '--file',
    action="store", dest="filename",
    help="query string", default="No filename passed")

parser.add_option('-b', '--block',
    action="store", dest="block",
    help="set to False for non interactive behaviour", default="True")

# read command line arguments
options, args = parser.parse_args()
savename = options.filename[:-4] + '.png'
print('Filename: ', options.filename)
print('Savename: ', savename)

block = True
if options.block == 'False':
```

```
        block = False

# define plot layout
style.use('fivethirtyeight')
fig = plt.figure(figsize=(10,8))
ax1 = fig.add_subplot(1,1,1)


def animate(i):

    try:
        graph_data = np.genfromtxt(options.filename, delimiter='\t')

        xs = graph_data[:,0] # get frame Nr.
        ys1 = graph_data[:,1] # get absolute Scratch area
        ys2 = graph_data[:,2] # get Sratch area in percent of Frame Area
        ys1_max = np.max(ys1, axis = 0) #get maximum Scratch Area
        ys1_percent = ys1/ys1_max*100   #normalize Scratch Area


        #labels and legend for plot
        ax1.clear()
        plt.title('Wound Healing Assay')
        plt.xlabel('Frame Nr.')
        plt.ylabel('Scratch Area [%]')
        ax1.plot(xs, ys1_percent, label = 'Percent of Initial Wound Area')
        ax1.plot(xs, ys2, label = 'Percent of Frame Area')
        ax1.legend(loc='upper right')
        #save plot
        plt.savefig(savename)
    except:
        print('No file loaded')


ani = animation.FuncAnimation(fig, animate, interval=1000, repeat=False)
plt.show(block=block)
plt.pause(0.1)
```

Run the Feedback Experiment and watch the results

Activate the experiment feedback in the **Acquisition** tab and click on **Start Experiment**. During the experiment the online display will be shown.

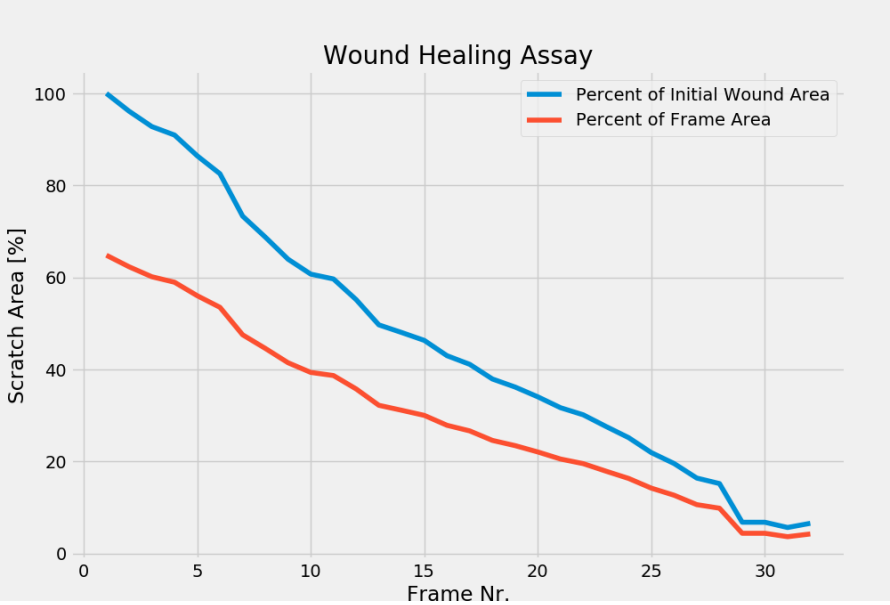

Figure 47: The resulting online data display showing the absolute area and the change.

4.12 Online Tracking

This example illustrates the possibility to use the online image analysis to track an object using a rather simple algorithm. One application example could be to keep the object of interest in the center of your field of view.

Idea or Task:

- Define a time-lapse acquisition.
- Use the ZEN 3.5 (blue edition) image analysis to figure out which particle has the highest mean intensity
- Use the XY-coordinates of this particle to move the stage.

The basic steps used for this example are:

- Detect the objects using an automatic threshold.
- Get all intensities of all detected objects.
- Find the index of the object with the highest mean intensity for every frame.
- Use the index get the XY-coordinates for the brightest object out of an array.
- Move the center of the stage to this position.

Definition of Feedback Experiment

For this example you can use the sample image data in the folder **...\Testimages\FakeTracks**. The folder contains 30 single TIFF images with artificial objects. Define a time series experiment with the following parameters:

- Cycles = 30
- Time Interval = 500 ms
- Channel 1 = EGFP

Definition of the Image Analysis Pipeline

In order to track the objects you need to set up an image analysis that determines the XY-stage coordinates of each detected object. For this example you can use a threshold-based object detection. There is no link between objects between adjacent frames implemented. Therefore this simple analysis might lead to a jump of the objectID.

- Do some image pre-processing.
- Detect the objects using an automatic threshold.
- Get all intensities of all detected objects.

The crucial steps of setting up the image analysis are show below. (You can also use the **EF_tracking.czias** image analysis setting.) Use a simple thresholding for object detection (Fig. 48).

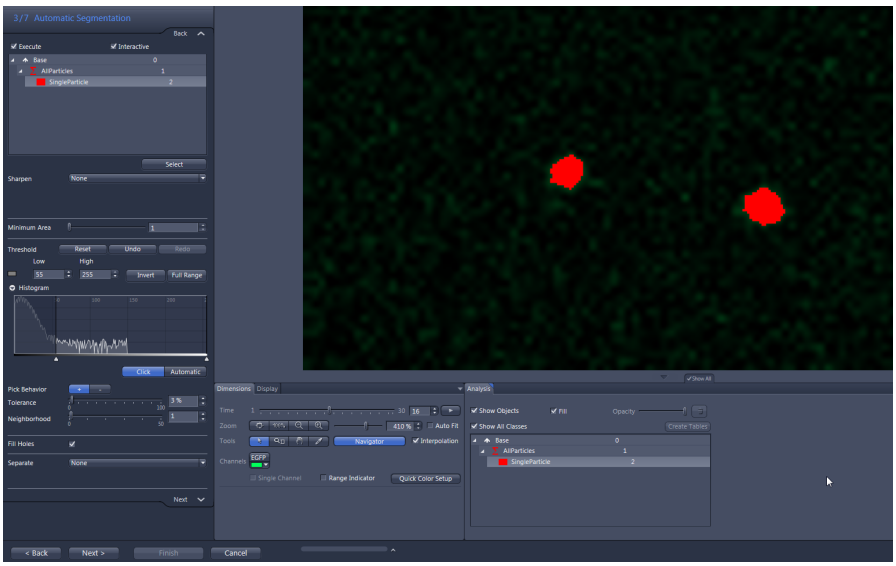

Figure 48: Simple thresholding is used to detect the objects.

Define the following features for **Regions**:

- **Count**
- **Image Stage Position X Image Center**
- **Image Stage Position Y Image Center**

and be sure to also define the following features for **Region**:

- **ID**
- **Intensity Mean Value of channel 'EGFP'**
- **Bound Center X Stage**
- **Bound Center Y Stage**

for later use in the feedback experiment.

The result of the image analysis for one frame of the data set is shown in Fig. 49. The ID the mean intensity and the X- and Y-coordinates of each detected objects are measured.

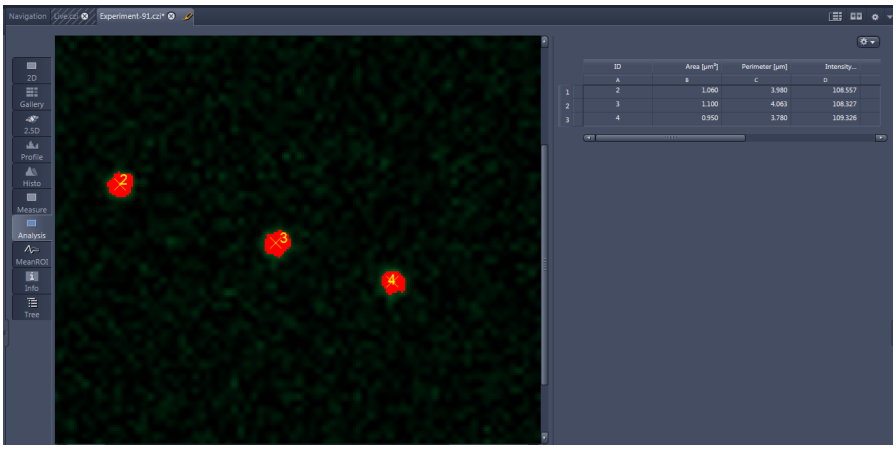

Figure 49: Results of the image analysis for an example frame.

Creation of the Feedback Script

The main tasks for this script can be described as follows:

- Initialize the required variables.
- Define a function that creates a string for the logfile out of an array (optional).
- Get the array containing all the XY-coordinates and intensities for **all single objects**.
- Find the brightest particle and the respective XY-coordinates using the array index.
- Move the stage to place the brightest object in the center of the field of view.

For further information read the comments inside the script, which explain the single steps in more detail. The corresponding experiment is called **EF_12_Track_Objects.czexp**.

```
### ------------------- PreScript ----------------------------------------------- ###


from System import Array

# define header for the oitput logfile to your needs - Timeframe - ObjectID - BoundCenterXStage - BoundCenterYStage -
↪  ImageStageXPosition - ImageStageYPosition
logfile = ZenService.Xtra.System.AppendLogLine('T\tObjID\tParticle X Pos\tParticle Y Pos\tStage Center X\tStage Center Y')

"""
# creates a readable string from all entries of an array (optional)
def createstr(arrayin):
    dim = len(arrayin)
    strout = ''
    for i in range(0,dim):
        if (i < dim-1):
            strout = strout + str(round(arrayin[i],2)) + '\t' # add tab at the end if it is NOT the last entry
        else:
            strout = strout + str(round(arrayin[i],2)) # no tab since it is the last entry
    return strout
"""


### ------------------- LoopScript ---------------------------------------------- ###


# get total number of objects and frame number
num_obj = ZenService.Analysis.AllParticles.RegionsCount
frame = ZenService.Experiment.CurrentTimePointIndex

```

```
30  # get current stage position XY of the image center
31  imgX = ZenService.Analysis.AllParticles.ImageStageXPosition
32  imgY = ZenService.Analysis.AllParticles.ImageStageYPosition
33
34  # get current object positions and intensity arrays for all detected objects
35  posx = ZenService.Analysis.SingleParticle.BoundCenterXStage
36  posy = ZenService.Analysis.SingleParticle.BoundCenterYStage
37  intensities = ZenService.Analysis.SingleParticle.IntensityMean_EGFP
38
39  # get ID of the brightest detected particle
40  ID = Array.IndexOf(intensities, max(intensities))
41
42  ## move the stage to the position of the brightest particle
43  ZenService.HardwareActions.SetStagePosition(posx[ID], posy[ID])
44
45  """
46  # create strings for all detected objects (optional for testing)
47  POSX = createstr(posx)          # array with all StageX positions
48  POSY = createstr(posy)          # array with all StageY positions
49  INTS = createstr(intensities)   # array with all intensities
50  """
51
52  # write positions to data log file
53  logfile = ZenService.Xtra.System.AppendLogLine(str(frame)+'\t'+str(ID+1)+'\t'+ str(posx[ID])+'\t'+str(posy[ID])+'\t'+
    ↪  str(imgX)+'\t'+str(imgY))
54
55
56  ### -------------------- PostScript ------------------------------------------- ###
57
58
59  # start Notepad (optional)
60  ZenService.Xtra.System.ExecuteExternalProgram(r"C:\Program Files (x86)\Notepad++\notepad++.exe", logfile)
```

Run the Feedback Experiment and watch the results

Activate the experiment feedback in the **Acquisition** tab and click on **Start Experiment**. As this is a simulated experiment with a simulated stage, the movement will just result in a shift of the overall image coordinate system. Within the **Gallery**-view you will be able to observe this as a black frame around the actual image.

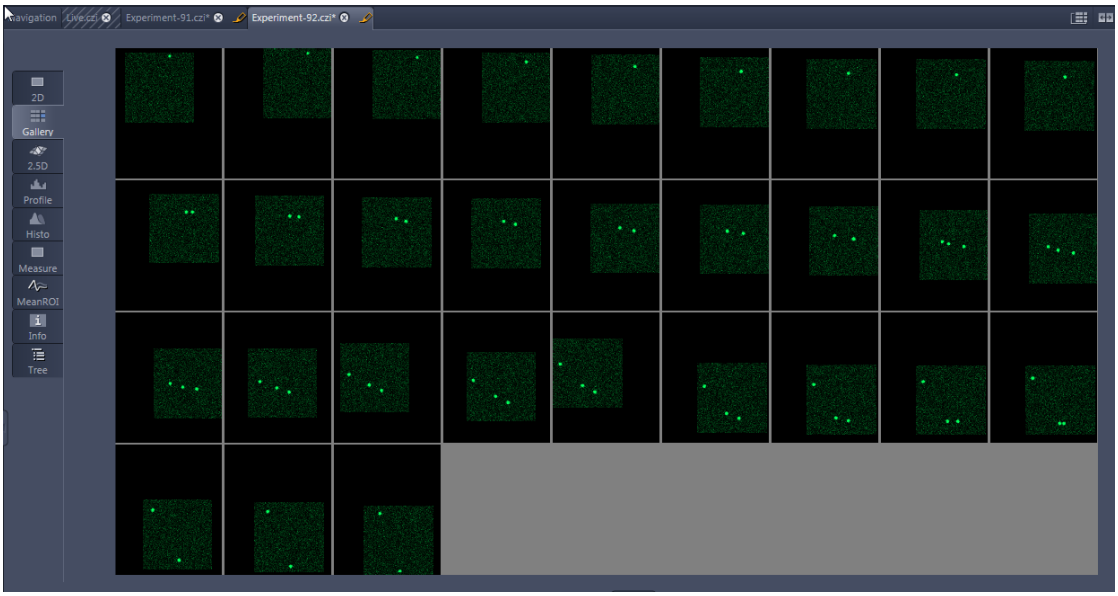

Figure 50: The resulting image data for simulated tracking in the **Gallery**-view.

5 Tools

5.1 ZEN Experiment Feedback Script Reader

The feedback script is stored as part of the experiment file, the *.czexp, an XML file that contains the complete settings to run an experiment. You can normally find these files at . . . \Documents\Carl Zeiss\ZEN\Documents\Experiment Setups\.
The feedback script reader **extract_script.py** is a tool that allows you to extract the feedback script from a *.czexp-file. It can be found at . . . \Feedback_Script_Reader. It is a Python script, demonstrating once more the advantages of using Phython for a huge variety of different tasks. The script parses the XML-file and extracts the experiment feedback script and stores it as *.py file. You can start the script from the command line via

▷ **python extract_script.py -f experiment_files_to_be_converted.czexp**

The Feedback Script Reader is not officially supported or provided by ZEISS, since it has nothing to do with ZEN 3.5 (blue edition) itself.

6 Script Commands

For a complete list of the available commands and their description please refer to the ZEN 3.5 (blue edition) online help.

7 Troubleshooting

When executing external scripts, such as Python code there are some potential pitfalls. If you are having troubles to run the examples as described, on your system you should check these two points:

- ***.py files must be associated with the respective python.exe**
- **are command line parameters passed to a python script**

Especially the second point can cause problems, even if it normally works out-of-the box. Those problems have nothing to do with the ZEN 3.5 (blue edition) software. If passing command line arguments via argparse (or optparse) does not work in Python, please check the registry if the following entries look like this:

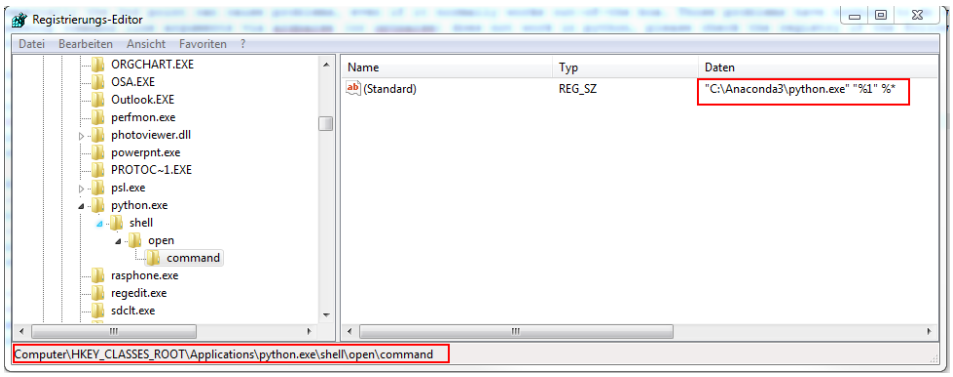

Figure 51: Registration Editor: Start Python from command line (1)

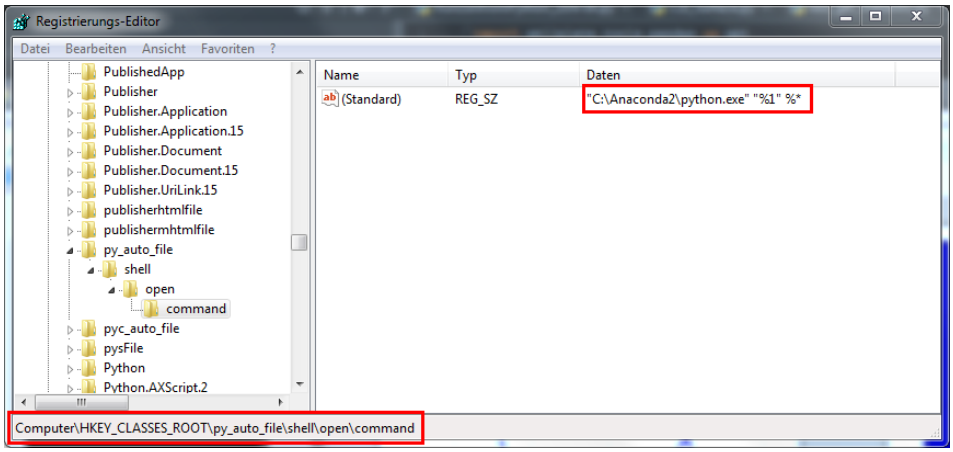

Figure 52: Registration Editor: Start Python from command line (2)

8 Disclaimer

Carl Zeiss Microscopy GmbH's ZEN 3.5 (blue edition) software allows to connect to the third party software Python. Carl Zeiss Microscopy GmbH undertakes no warranty concerning Python, makes no representation that Python will work on your hardware and will not be liable for any damages caused by the use of this extension. By running one of those examples you agree to this disclaimer.