Contents

1 ZEN (blue Edition) - Experiment Feedback

This tutorial contains general information on the usage of the ZEN Blue Feedback Experiments and some test examples for typical use cases. All of them can be tested on a simulates system without any hardware.

1.1 Definition

Experiment feedback (conditional or adaptive experiments) allows the definition of specific rules and actions to be performed during an experiment. This allows changing the course of an experiment depending on the current system status or the nature of the acquired data on runtime. Moreover, it is possible to integrate certain tasks like data logging or starting an external application, directly into the imaging experiment. Typically, but not exclusively, such an experiment connects the image pickup with an automatic image analysis.

1.2 Key Features

- Create **smart experiments** with Experiment Feedback or Experiment Designer and modify the acquisition **"On-the-fly"** based on **Online Image Analysis**, Hardware Changes or External Inputs (e.g. TTL Signals).
- **Adaptive Acquisition Engine** allows modifying running experiments according to the rules defined inside the python Feedback Script.
- Python Script can incorporate data from current system status and results from the **Online Image Analysis** on runtime during the experiment.
- **Data Logging** or starting an **External Application** (Python, Fiji, MATLAB, ...), directly from within the imaging experiment is possible.

1.3 General Workflow

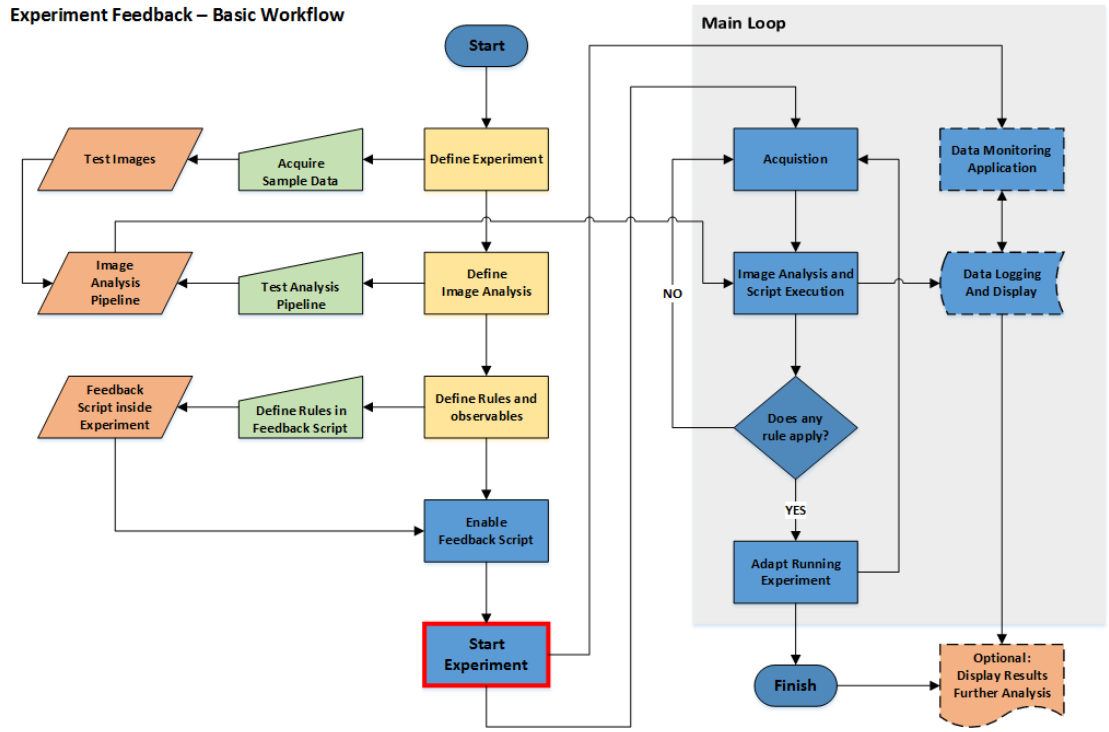


Figure 1: General Workflow Experiment Feedback

This diagram is one possibility to illustrate the typical workflow using a diagram. It contains the most important steps.

- **Define Experiment** – Setup and Configure the actual image acquisition experiment to the get the desired image data (time lapse, z-stack, multi-channel, tile acquisition, ...) - once defined it is recommended to **Acquire Sample Data**, which are required to setup and test the image analysis pipeline.

- **Define Image Analysis** – Setup the image analysis pipeline (use wizard or OAD macro for advanced topics) for the latter use inside the feedback script - only the specified parameters can be accessed from with the script later and it is strongly recommended to **Test Image Analysis Pipeline** to ensure the created results are meaningful.

- **Define Rules and Observables** - this step is all about making up one's mind on how the script should actually work. What must be observed and how should the experiment react upon a certain event - once the main idea becomes clear one can start to **Define Rules in Feedback Script**.

At this point one can start the experiment and watch the output. The general concept behind this workflow can be described as a loop, which is the actual acquisition. Every time "something", usually an image acquisition took place the script will be executed. The rules will be check 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. Which is the best moment to do so depends on the application.

1.4 Adaptive Acquisition Engine

The script itself is only executed if something "***HasChanged***". It is important to note that the script run will be only triggered, if an observable has changed or is used inside the script. Typically this can be:

- New image acquisition
- XYZ-Move
- Status of Trigger Port is altered
- Filter, Objective, Light Source modifications
- Change of Incubation

If "nothing is going on" the script will not be executed unless one uses the ***RunRepetion-Script*** command in conjunction with a timer (advanced use case).

2 The Experiment Feedback Script

In order to activate an Experiment Feedback Script one needs to enable the checkbox **Experiment Feedback**. Only when activated the respective tool window will become apparent.

2.1 Synchronization Options

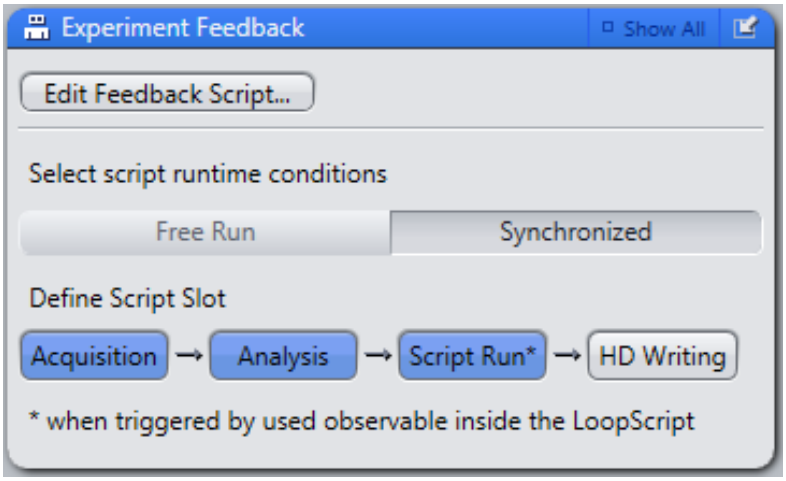


Figure 2: Options for the Synchronized acquisition

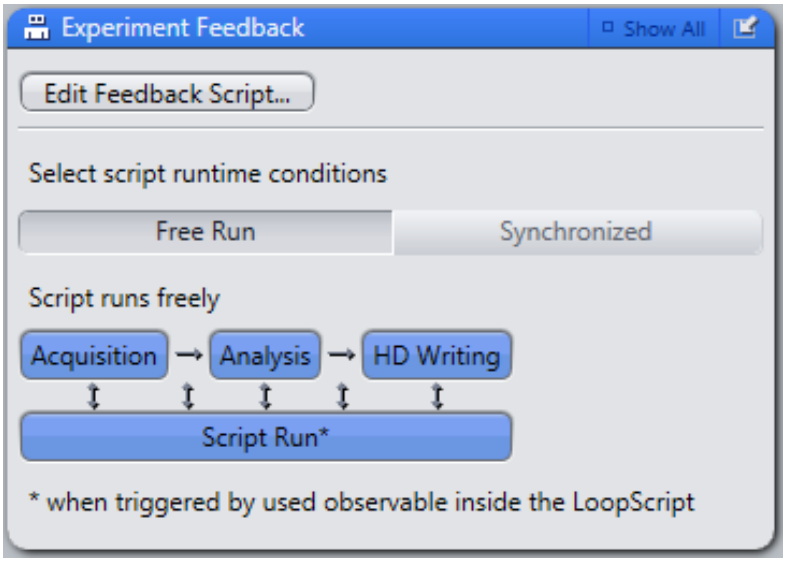


Figure 3: Options for the Free Run mode

The tool window to configure the script and the synchronization option can be found on the acquisition page. From inside this windows one has access to the following functionality:

- Edit the Feedback Script itself inside the script editor.
- Choose the script runtime conditions – **Free Run** or **Synchronized**.
- Allow additional loop script runs for hardware and environment variables.

Especially the script runtime conditions are important. The available slots for the execution order can be adjusted via clicking on the blue buttons.

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

2. Option 2 - The online image analysis is started after the acquisition and only when it is finished the script run will be triggered. Therefore it is guaranteed, that all analysis results exist and can be used inside the feedback script. After the analysis is finished the image sub-block will be written to disk.

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 script run will be triggered. Therefore it is guaranteed, that all analysis results exist and the image data is stored on disk before the script run is triggered. This might be important in case the script starts an external application to analyze the data.

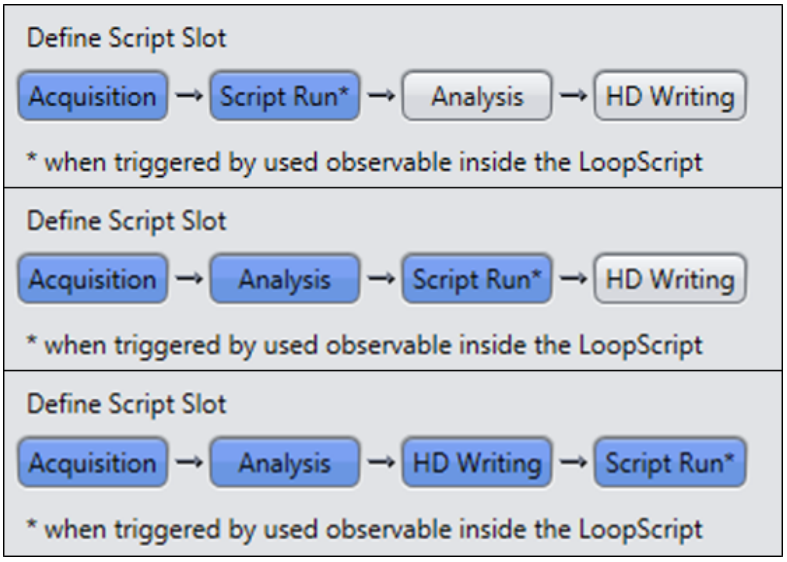


Figure 4: Differences between the three Synchronization Options

2.2 The Feedback Script Editor

The editor itself is used to modify the feedback script, which is stored as part of the experiment file. It has three distinct sections.

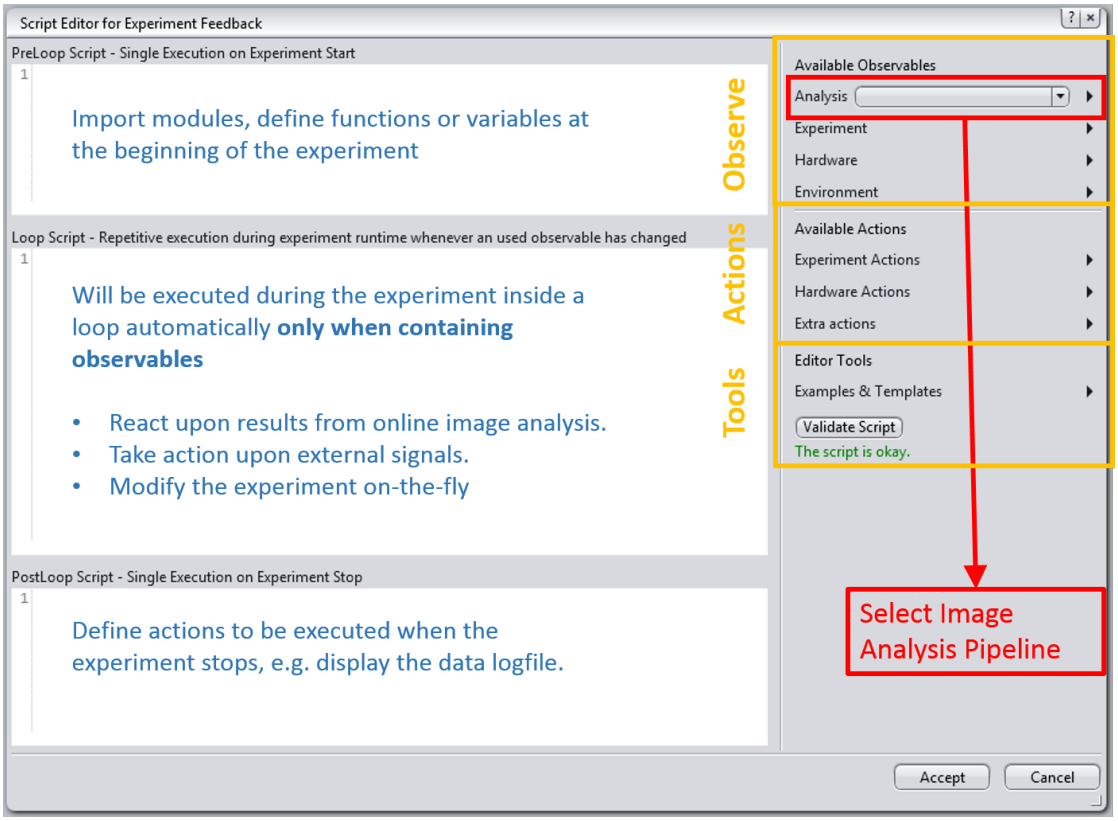


Figure 5: Experiment Feedback Script Editor

Note: It is important to note that the Loop Script itself will be only executed, if it contains an observable, who's values has changed during the acquisition or if the observable was used.

3 Prerequisites

3.1 Install Python

To display the results we will use a Python distribution, which is a logical choice since feedback experiment scripts and ZEN macros are written in Python anyhow. There are various possibilities to install Python and the required modules. The easiest way is probably to install a complete distribution, for instance the Anaconda distribution, which can be downloaded here:

- **Anaconda -** https://www.continuum.io/downloads

Just use the full install option and follow the instructions. **Of course you are free to use the Python distribution of your choice.**

3.2 Useful Python Links

- **IPython** - www.ipython.org
- **WinPython** - http://sourceforge.net/projects/winpython/
- **The Python Tutorial** - http://docs.python.org/2/tutorial/
- **Official IronPython** - http://ironpython.net/
- **MatPlotLib** - http://matplotlib.org/

And keep in mind, the internet is your friend here...

3.3 Install Fiji

The release version of Fiji can be downloaded from here: http://fiji.sc/Downloads

- Use the 64-bit version of Fiji only.
- Place the Fiji folder here: C:\Users\Public\Documents\Fiji
- Make sure, your Fiji is always up-to-date.

3.4 Simulated Sample Camera

This 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 in order to use them for a simulated experiment including a real online analysis.

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

If configured correctly inside the MTB one can specify the image folder inside ZEN here:

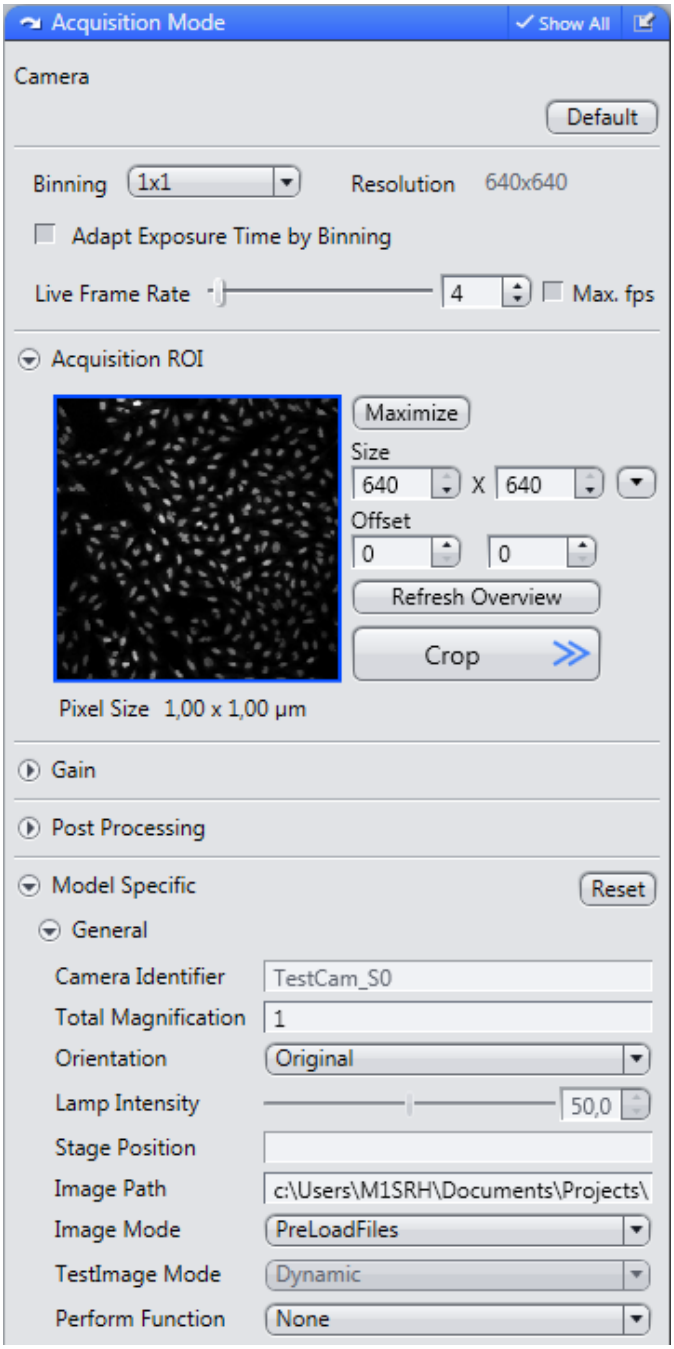


Figure 6: ZEN - Sample Camera User Interface

3.5 ZEN Experiment Feedback Script Reader

Since the actual feedback script is part of the experiment, it is not trivial to juts load the feedback script itself. Currently the script is part of the complex *.czexp file, which is a XML file. Fortunately it is not too difficult (but not trivial) to parse an XML file using Python. One can use the Python application **ZEN_Script_Macro_Reader.py** (Python must be installed, otherwise use the **Zen_Script_Macro_Reader.exe**) to extract the feedback script from the XML file.

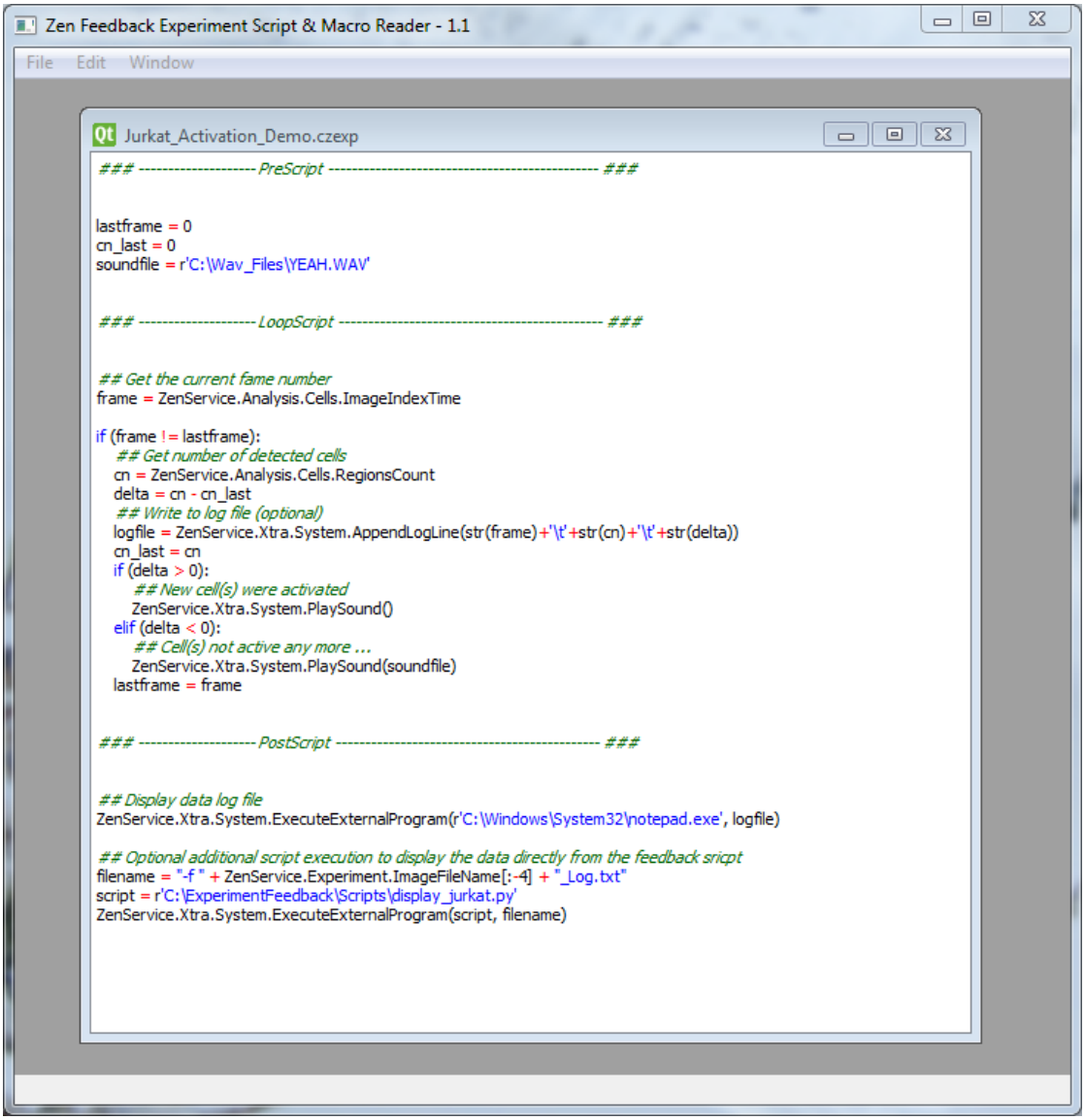


Figure 7: ZEN Experiment Feedback Script Reader

This application is not officially supported (since it has nothing to do with ZEN itself) or provided, but it illustrates furthermore the advantages of using Python for a huge variety of different tasks.

4 Application Examples

This is a collection of examples to illustrate some of the capabilities of the feedback experiments.

4.1 Count cells, log data and start external acquisition to display results

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 to finally display the results of the online analysis.

Idea or Task:

- Define an experiment
- Include an image analysis
- Run the experiment and do an analysis every frame
- Display the results with an external application

Definition of Feedback Experiment

For this example one can use the sample image folder ...\Testimages\96_well. This contains 96 single TIFF images with cells where the nucleus is stained with DAPI.

So just 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 will keep it simple and just set up a simple time lapse experiment called EF_Plot_Cell_Count_Simple.czexp with the following parameters:

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

When finished do not forget to make a snap shot in order to create some sample data. The task is to acquire the data, analyze the number of cells per image, log the data into a file and finally display the and save output.

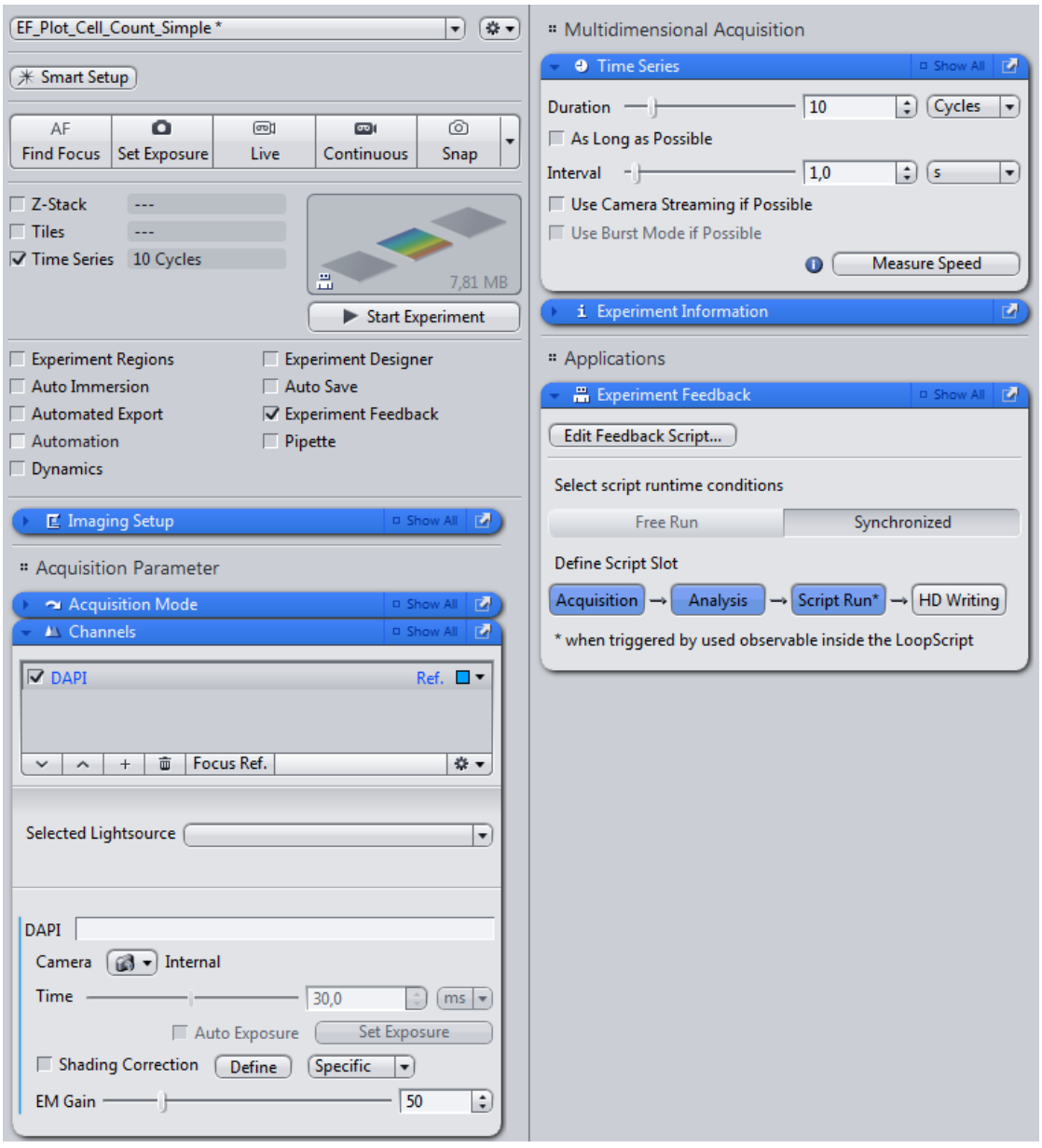


Figure 8: ZEN - Setup Example Experiment

Definition of Image Analysis Pipeline

The next logical step is to create an image analysis pipeline. (This example assumes that the user is at least familiar with the image analysis wizard). So only the really crucial steps are described. And keep in mind, that no interactive steps from the wizard can be used for the feedback experiment scripts. For this example one should create a measurement program called **Count_Cells_DAPI**.

- Define the object and region class – in this case **SingleCell** and **Cells**

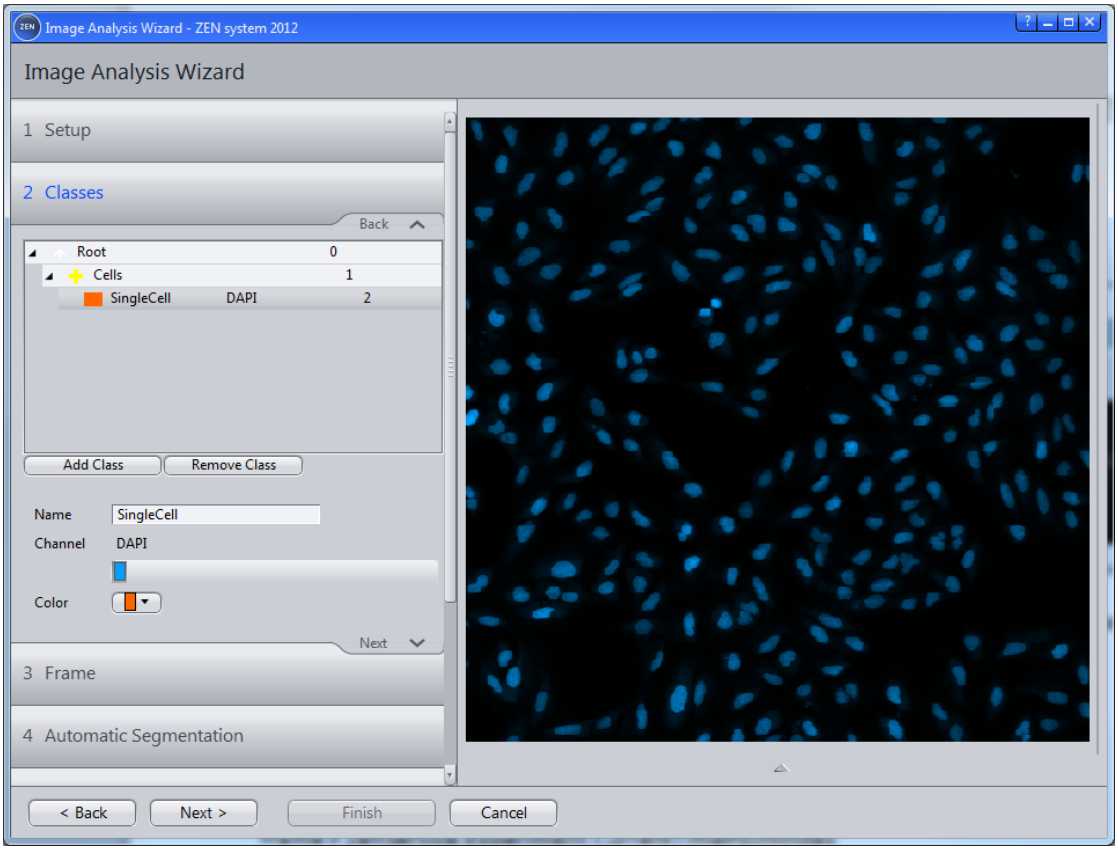


Figure 9: Image Analysis Pipeline - Class Definition

- Adjust the segmentation in order to count only the single cells. It does not really matter to get the precise number of cells, since this is usually not crucial, e.g. it does not really matter, if one counts 237 or 240 cells...

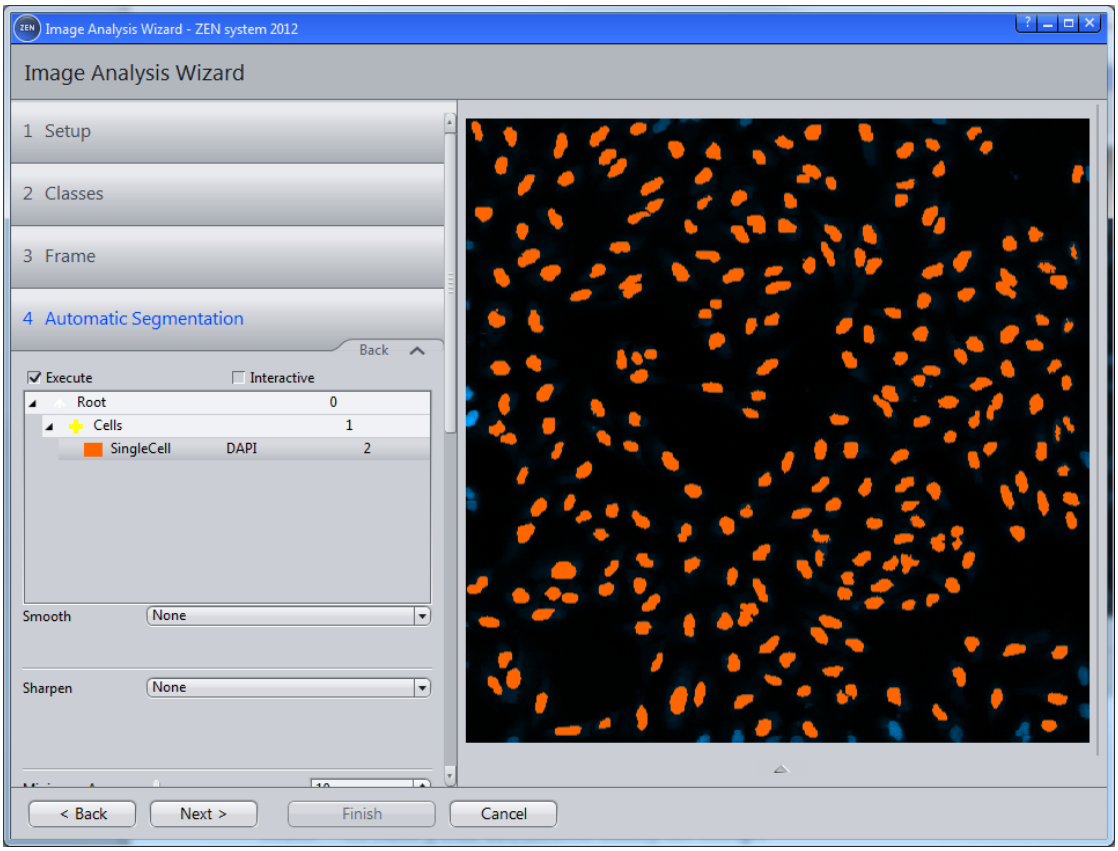


Figure 10: Image Analysis Pipeline - Segmentation

- Define parameters for all objects (Region Parameter) and for the single objects (Regions Parameter) to be accessed later on inside the feedback script.

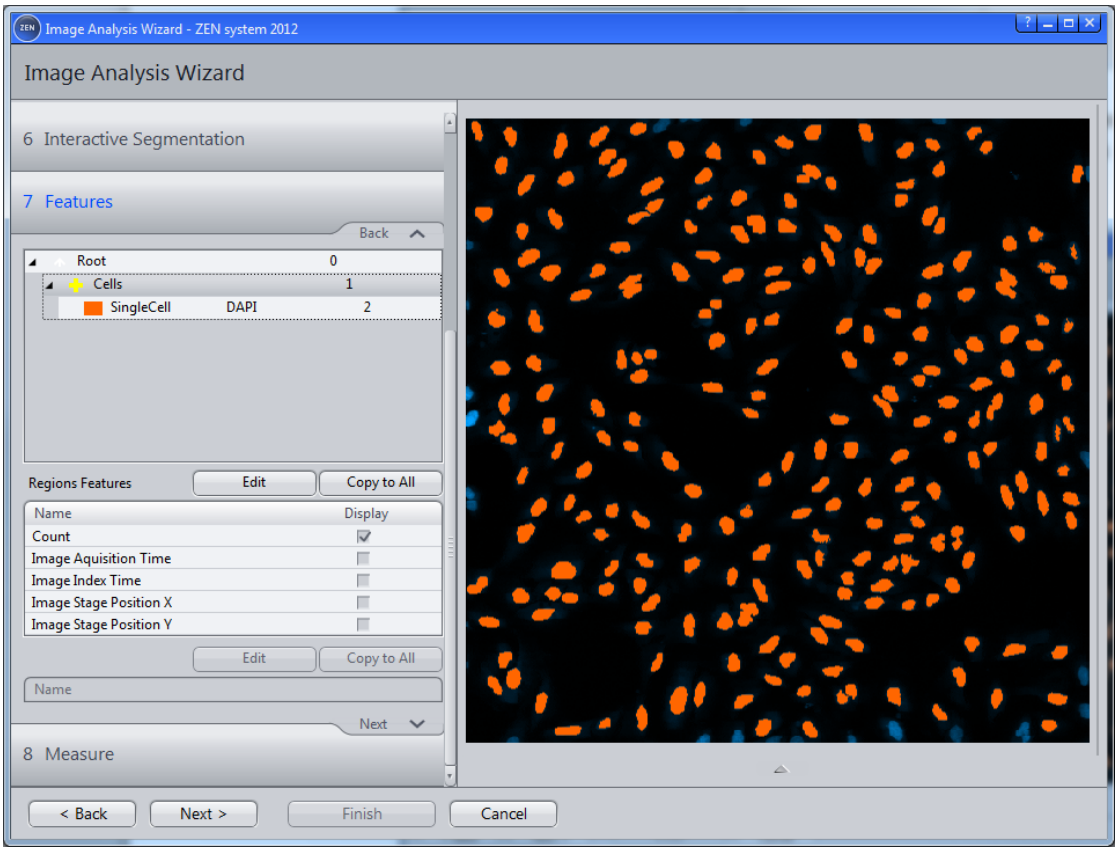


Figure 11: Image Analysis Pipeline - Select Measurement Parameter

- Finalize the image analysis pipeline and check the results.

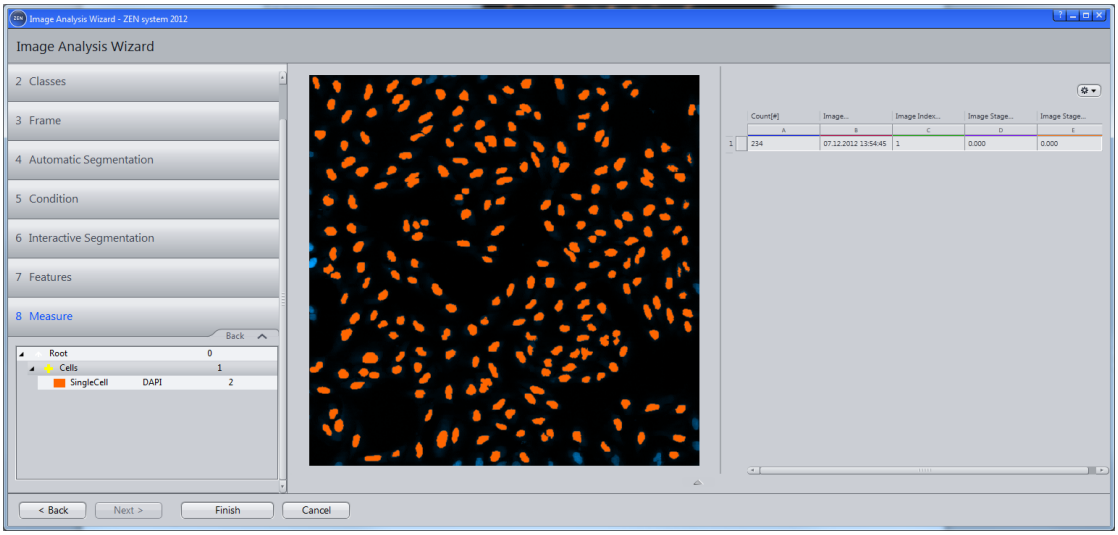


Figure 12: Image Analysis Pipeline - Finalize and Check results

- Re-run the analysis pipeline using the Analyze button and check the results.

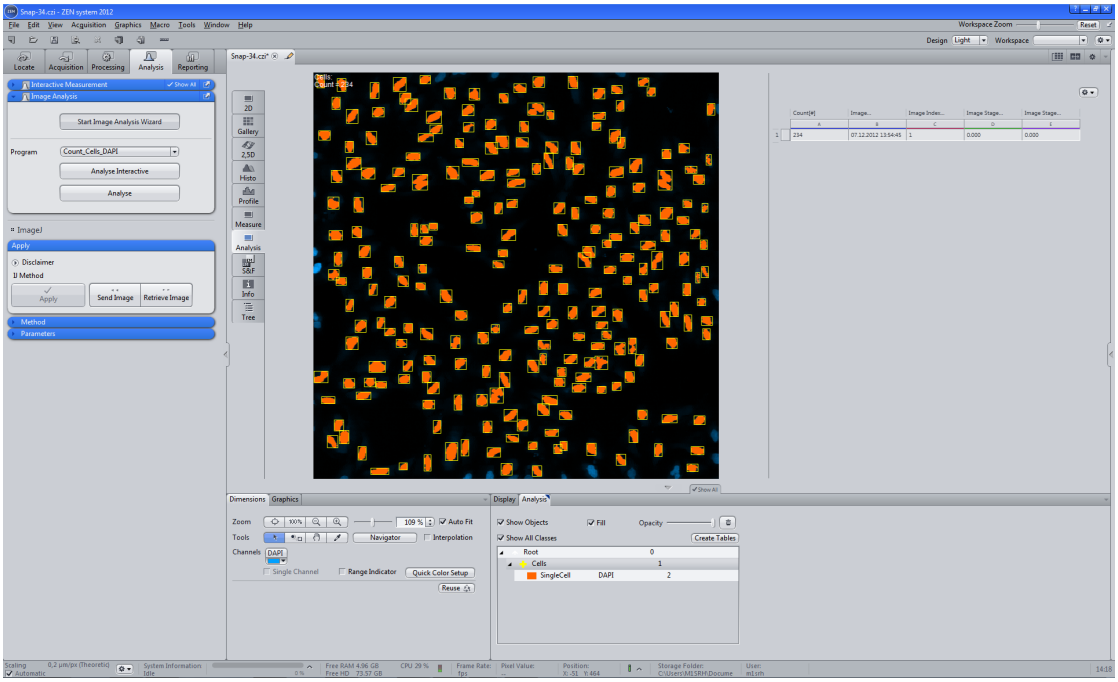


Figure 13: Image Analysis Pipeline - Re-Run Measurement

Creation of Feedback Script

So keep in mind the main idea: Count the number of cells for every frame, log the data and display the result at the end. The first step here is to create the actual feedback script. Go to the Automation Tab and press the **Edit feedback Script...** button. The empty script editor opens up. Now one can start writing. For this example the script is shown below.

It is important to point out that there is no correct solution how to write a feedback script that fulfills the task. Whatever works is fine, so the solution shown here is just one out of a few possible solutions. One of its main purposes is to point into the "right direction" and show some useful hints and tricks.

For this example we choose the synchronization settings shown in figure **8**. 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 when the image analysis is finished.**

The first important step is to select the correct image analysis pipeline in order to be

able to use the respective parameters from the image analysis inside the script.

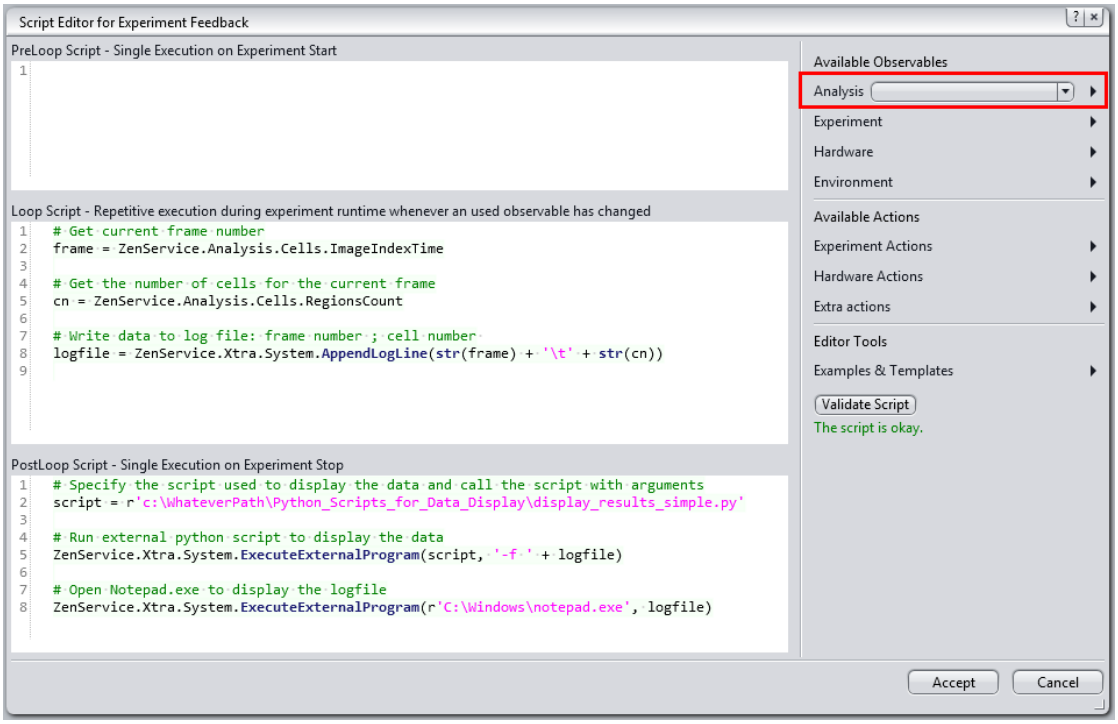


Figure 14: Select the Image Analysis and create the script

- **Single Execution on Experiment Start**

The first part of the script editor is usually used to initialize variables, so one can use them later on. In our case we initialize a frame counter which we will use later to ensure that every frame is analyzed only once.

- **Repetitive Execution on Experiment Start**

This the main loop of the script executed during experiment runtime whenever an observable has changes or is used inside the script. The number of detected objects will be passed to the script and will be written into a logfile. The final feedback script is shown below.

```
###   PreScript   ###



###   LoopScript ###

# Get current frame number
frame = ZenService.Analysis.Cells.ImageIndexTime

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

# Write data to log file: frame number ; cell number
logfile = ZenService.Xtra.System.AppendLogLine(str(frame) + '\t' + str(cn))

###   PostScript   ###

# Specify the script used to display the data and call the script with arguments
script = r'...\Python_Scripts\display_results_simple.py'

# Run external python script to display the data
ZenService.Xtra.System.ExecuteExternalProgram(script, '-f ' + logfile)

# Open Notepad.exe to display the logfile
ZenService.Xtra.System.ExecuteExternalProgram(r'C:\Windows\notepad.exe', logfile)
```

The 1st column contains the frame number, the 2nd contains the number of counted cells. Both columns are separated by tabs.

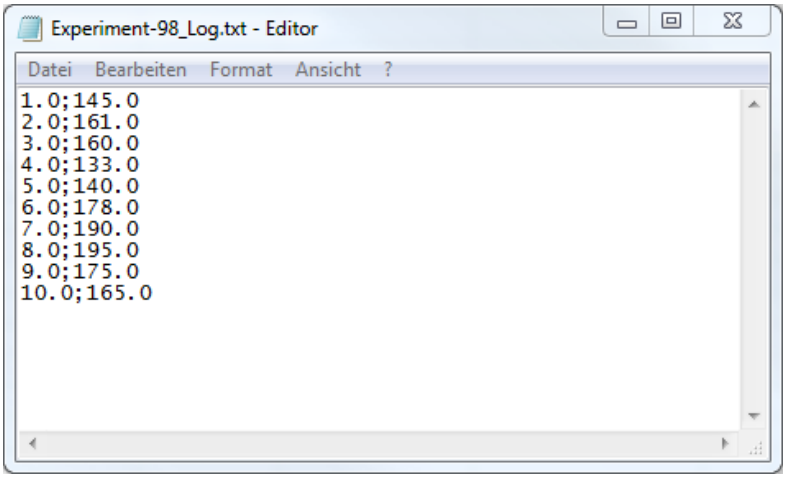


Figure 15: Logfile create by the Experiment feedback Script

- **Single Execution on Experiment Stop**

When the experiment is finished one wants to display the data right away and save the graph. It really depends on the application what is a good way to display the data, so this example is just a suggestion. And it is important to understand, that at this point, the user leaves the ZEN world.

As already mentioned it makes sense to use Python as well in order to display the data in order to stick to one scripting language, but this is of course a matter of taste.

Creation of Python Script for the Data Display

Once the format of the data log file is defined, one 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 out the example code on how to create the selected plot ...

To create a Python script one can use every text editor (Notepad++ etc.) or (recommended) use Spyder or PyCharm to edit and even test the script. This How-To will not explain the Python script in detail, but some points are important to have in mind

- The provided Python script is just an example
- The correct way to import the data depends on the analysis output

Remark: To make this work, the location of Python scripts must be added to the environmental variable PATH in order to make it "visible" when the script is called from within ZEN.

The main tasks of this example script are:

- Enable command line parsing - one can "call" the script and provide the filename of the out data file as an argument.
- Read in the data correctly.
- Display the data in an appropriate way.

And of course once the data are imported into Python, the door is wide open for all kinds of subsequent analysis and further data processing.

```
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")

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

# 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()
```

Run the Feedback Experiment and watch the results

Activate the experiment feedback under **Automation** and start the experiment. For this example a simple time series is recorded. Every frame is analyzed and the data are written into a specified file.

If everything worked fine 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. Here the 1st column represents the frame number and the 2nd column the number of detected cells. Both values are separated by a tab. This has to be taken into account when writing the Python script used for displaying the acquired data.

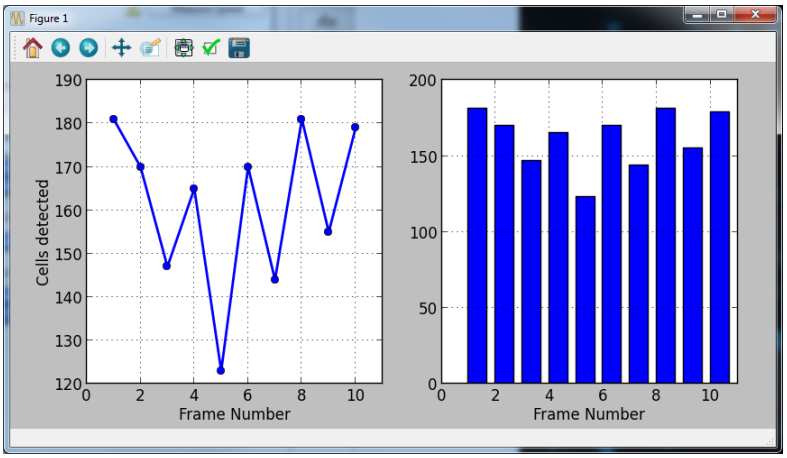


Figure 16: Logfile create by the Experiment feedback Script

This is the actual figure created by the Python script and below the saved figure (PNG-File).

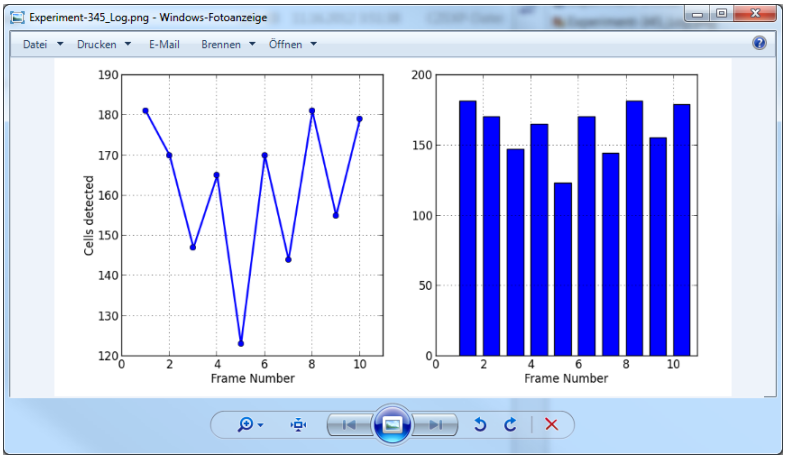


Figure 17: Logfile create by the Experiment feedback Script

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 logical next step would be to try the data display online ("as the data arrive") or to include further analysis steps.

4.2 Acquire Image Data, Open in Fiji and apply macro to data

The main purpose of this workflow is to demonstrate the possibilities to automate and customize an experiment.

Idea or Task:

- Define an experiment
- Include an image analysis
- Run the experiment and open the resulting CZI file automatically inside Fiji
- Further processing inside Fiji

Definition of Feedback Experiment

For this example this task is quite simple since the main idea is actually not really the definition of a feedback rule based on some observables, but to automate a workflow.

- Acquire Image Data
- Transfer Image data directly to Fiji
- Apply Fiji Macro directly after the data transfer

Those three steps will be included directly into the ZEN Experiment itself. So just think of any useful experiment. For the example we will just use a simple Z-Stack.

Create the Fiji macro

Especially Fiji (and ImageJ as well) has very powerful scripting capabilities. For further information and examples, please look here:

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

To create a macro for Fiji one can use every text editor (Notepad etc.) or you can use Fiji´s built-in script editor (this is where the following screen-shots are taken from) . For a starter we will used a very simple macro, which actually has the following purpose:

- Read the filename of the CZI file created by ZEN via command line
- Import the CZI file using the Bio-Formats library (preserves most of the metadata like, image dimensions and stage coordinates)
- Use a virtual stack to display the file - saves a lot of memory since only the current frame is opened

Here one can see an example for a typical Fiji macro. The first version is the simple solution – just import then CZI file and display it.

```
1  // Author:        Sebastian Rhode
2  // Date:        04.10.2012
3
4  name = getArgument;
5  if (name=="") exit("No argument!");
6
7  run("Bio-Formats Importer", "open=[" + name + "] autoscale color_mode=Default open_all_series view=Hyperstack
     ↪    stack_order=XYCZT");
8  // get dimension from original image
9  getDimensions(width,height,channels,slices,frames);
10 // apply additional operation
11 run("Z Project...", "start=1 stop=" + slices + " projection=[Max Intensity]"); // do maximum intensity projection
12 run("Fire"); // apply special LUT
```

- Line 4: Read the filename.
- Line 5: Print error message in case there was no filename.
- Line 7: Open CZI file using Bio-Formats and configure the display options.
- Line 11-12: Do a Z-projection and apply a special LUT.

Definition of Feedback Experiment

For this example one can use the sample image folder ...\Testimages\HeLa_ZStack. This contains 15 single TIFF images with cells where the mitochondria are stained with MitoTracker Red.. So just define a Mito Red channel and test if the sample camera gives the correct output. Next step is to set up a Z-Stack with 15 planes. And finally save the experiment EF_Open_Fiji_after_End.czexp with the following parameters:

- Z-planes = 15
- Channel = MitoDAPI

Creation of Feedback Script

For this example no special image analysis is required, so there is no need to set up an image analysis pipeline. The task one has to keep in mind are:

- Start Fiji when the acquisition is finished.
- Hand over the correct file name.
- Directly apply the macro on the imported dataset.

The crucial point is the possibility to start Fiji via command line with options as shown here:

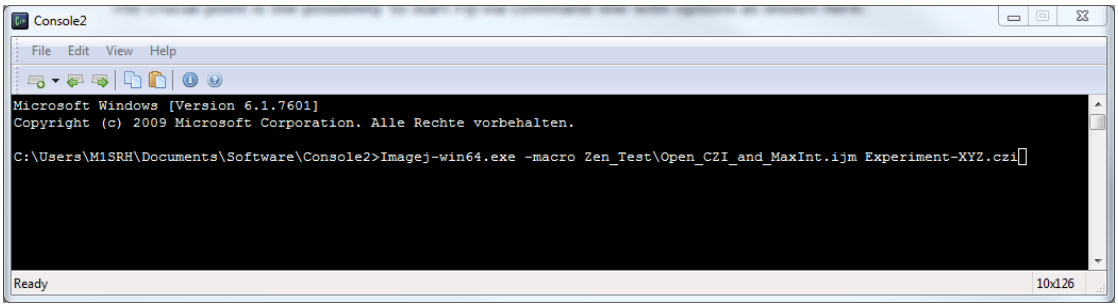


Figure 18: Test the Fiji macro from command line

This functionality can be easily integrated in the feedback script as shown below:

```
1  ###   PreScript   ###
2
3
4
5  ###   LoopScript ###
6
7
8
9  ###   PostScript   ###
10
11 # get the name of the current image data set
12 filename = ZenService.Experiment.ImageFileName
13 # use the absolute path --> this works always
14 exeloc = 'C:\Users\Public\Documents\Fiji\ImageJ-win64.exe'
15  # specify the Fiji macro one wants to use
16 macro = r'-macro Zen_Test\Open_CZI_and_MaxInt.ijm'
17 # 'glue' together the options
18 option =  macro + ' ' + filename
19 # start Fiji, open the data set and execute the macro
20 ZenService.Xtra.System.ExecuteExternalProgram(exeloc, option)
```

Run the Feedback Experiment and watch the results

Activate the experiment feedback under Automation and start the experiment. For this example a simple Z-Stack is acquired. When finished Fiji will start, import the CZI file and apply a MaxIntensity projection followed by a LUT.

If everything worked fine one should get the following result:

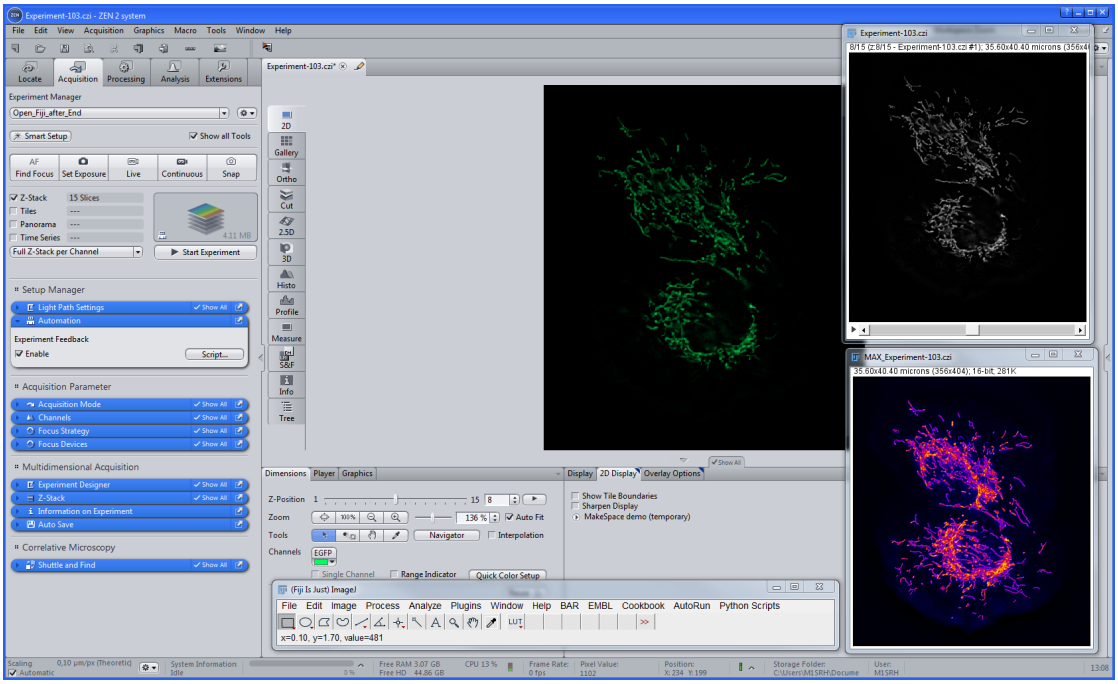


Figure 19: Fiji open CZI automatically and runs macro

This example demonstrates the possibility to call an external application, in this case Fiji from inside an experiment to do something. As an example we just imported the CZI file. And should be seen as a starting point, since it opens up the possibility to extend the macro with further steps:

- Image analysis
- Filtering
- 3D stitching
- Tracking

Of course those tasks can be also done manually afterwards, when one imports the CZI file manually and works from there on inside the Fiji software. But nevertheless it is always a good idea to automate what can be automated, especially in the field of microscopy.

4.3 Acquire Tile Images until a total number of objects in reached

The main purpose of this example is to further illustrate some capabilities and good practices when using a feedback script. Here it is useful to have a real microscope setup (motorized XY stage) and a real sample. With stained nuclei.

Idea or Task:

- Define an experiment
- Count number of objects
- Run the experiment and stop the acquisition upon an event
- Display the results with an external application

Definition of Feedback Experiment

Create an tile experiment and save it as **EF_Count_Cells_Tiles_DAPI.czexp** with the following experiment settings:

- Use 2.5X objective
- Tiles = 8x8 = 64 images (this will depend on the selected magnification)
- Channel = DAPI (or whatever the nuclei are stained with ...)

Definition of Image Analysis Pipeline

Here one can use a similar analysis pipeline as for the example 4.1in order to count the number of cells.

Creation of Feedback Script

This time the script is a bit more advanced in order to illustrate some typical scripting techniques. As usual it is useful to recall the general experiment workflow:

1. Acquire tile image and count number of objects
2. Calculate the total number of objects
3. Stop the experiment, if the desired number of objects was reached to save time and data
4. Create data log, display the data and save the plot

```
1  ###   PreScript   ###
2
3  cells_total = 0 # total number of cells
4  logfile = ZenService.Xtra.System.AppendLogLine('Tile\tCells\tTotal\tPosX\tPosY')
5  script = r'c:\Users\M1SRH\Documents\Projects\ExpFeedback_Current\Python_Scripts_for_Data_Display\display_results_tiles.py'
6
7  ###   LoopScript ###
8
9  # get cell number for the current tile
10 cn =ZenService.Analysis.Cells.RegionsCount
11 # get current tile number
12 tile = ZenService.Experiment.CurrentTileIndex
13 # sum up the number of cells
14 cells_total = cells_total + cn
15 # stop if the desired cell number was already reached
16 if (cells_total > 2000):
17     ZenService.Actions.StopExperiment()
18
19 # read the xy position of the current image
20 posx = ZenService.Analysis.Cells.ImageStageXPosition
21 posy = ZenService.Analysis.Cells.ImageStageYPosition
22 # write data into log file
23 logfile = ZenService.Xtra.System.AppendLogLine(str(tile)+'\t'+str(cn)+'\t'+str(cells_total)+'\t'+str(posx)+'\t'+str(posy))
24
25 ###   PostScript   ###
26
27 ZenService.Xtra.System.ExecuteExternalProgram(script, '-f ' + logfile)
28 ZenService.Xtra.System.ExecuteExternalProgram(r'C:\Program Files (x86)\Notepad++\notepad++.exe', logfile)
```

- **Single Execution on Experiment Start**

The new idea here the initialization of a variable called cells_total to have a container for the number of total objects. Be aware that this value is not available from the image analysis directly, since this is a custom parameter, which has to be calculated. In Line 3 a header line for the logfile is created to make it more readable later on.

- **Repetitive Execution on Experiment Start**

The crucial idea here is the task of summing up the total cell number for every acquired tiles. This is shown in line 12, but it requires the variable cells_total to be initialized before this line is executed the first time. In line 15 the actual experiment adaptation takes place – in this case the acquisition will be stopped, when the overall number of cells has exceeded 2000 (feel free to adapt it to your needs).

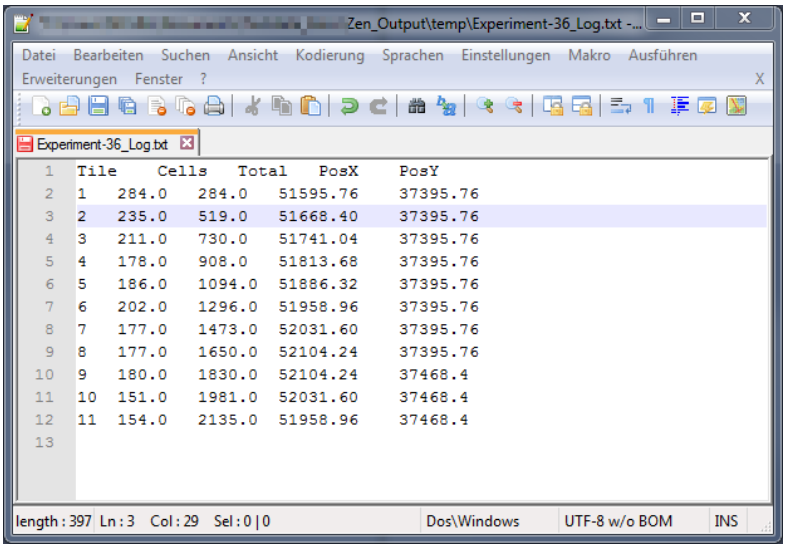


Figure 20: Logfile create by the Experiment Feedback Script

The 1st column contains the tile number, the 2nd the number of cells, the 3rd the total amount of cells so far. The last two columns are used to store the actual stage positions. Having this format in mind one can easily create a Python script to display the data later on.

- **Single Execution on Experiment Stop**

The last part of the script is just to start the Python script to display the data.

Creation of Python Script for the Data Display

Of course there are tone of useful ways to display the data, so the script below is just one possible example. It is more or less a variation of the script already used in chapter 4.1 earlier in this document.

```
1  import numpy as np
2  import optparse
3  import matplotlib.pyplot as plt
4
5  # configure parsing option for command line usage
6  parser = optparse.OptionParser()
7
8  parser.add_option('-f', '--file',
9      action="store", dest="filename",
10     help="query string", default="spam")
11
12 # read command line arguments
13 options, args = parser.parse_args()
14
15 savename = options.filename[:-4] + '.png'
16 print 'Filename: ', options.filename
17 print 'Savename: ', savename
18
19 # load data
20 data = np.loadtxt(options.filename, delimiter='\t', skiprows=1)
```

```
21
22  # create figure
23  figure = plt.figure(figsize=(12,4), dpi=100)
24  ax1 = figure.add_subplot(131)
25  ax2 = figure.add_subplot(132)
26  ax3 = figure.add_subplot(133)
27
28  # create subplot 1
29  ax1.bar(data[:,0],data[:,1],width=0.7, bottom=0, color='blue')
30  ax1.grid(True)
31  ax1.set_xlabel('Tile Number')
32  ax1.set_ylabel('Cells detected')
33
34  # create subplot 2
35  ax2.bar(data[:,0],data[:,2],width=0.7, bottom=0, color='green')
36  ax2.grid(True)
37  ax2.set_xlabel('Tile Number')
38  ax2.set_ylabel('Cells in total')
39
40  # create subplot 3
41  ax3.plot(data[:,3],data[:,4],'ro', markersize = 5)
42  ax3.grid(True)
43  ax3.set_xlim([data[:,3].min()-100,data[:,3].max()+100])
44  ax3.set_ylim([data[:,4].min()-100,data[:,4].max()+100])
45  ax3.set_xlabel('X Stage Position')
46  ax3.set_ylabel('Y Stage Position')
47
48  # add frame number directly to every data point
49  for X, Y, Z in zip(data[:,3], data[:,4], data[:,0]):
50      # Annotate the points
51      #ax3.annotate('{}'.format(int(Z)), xy=(X,Y), xytext=(5, 10), ha='right', textcoords='offset points')
52      ax3.annotate('{}'.format(int(Z)), xy=(X,Y), xytext=(0, 15), ha='right',
53                   textcoords='offset points',arrowprops=dict(arrowstyle='->', shrinkA=0))
54
55  # configure plot
56  figure.subplots_adjust(left=0.07, bottom=0.15, right=0.97, top=0.90,wspace=0.30, hspace=0.20)
57  # save figure
58  plt.savefig(savename)
59  # show graph
60  plt.show()
```

Run the Feedback Experiment and watch the results

Start the tile acquisition and watch the tile image growing. For this example the tile size was 4x4, but depending on the actual selected size, the image analysis and the sample, the acquisition will stop earlier. From the data log file in chapter 4.3 one can see that the tile acquisition stopped after 6 tiles out of 16 that were initially set up. The total number of cells after tile 6 was bigger than 1200, so the experiment was stopped skipping the remaining 10 tiles. Imaging this general idea for a complete 96 well plate ... A lot of time and amount of data can be saved the smart way.

The figure created by the Python script is shown below. The 1st subplot contains the number of cells per tile, the 2nd one the total cell number and the last one just the XY stage positions. Additionally the script also save this plot (not shown)

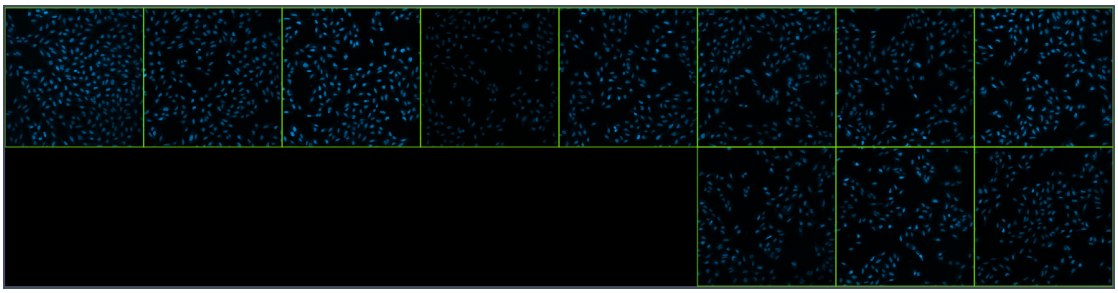


Figure 21: The acquisition was terminated after the the "stop" criteria was met

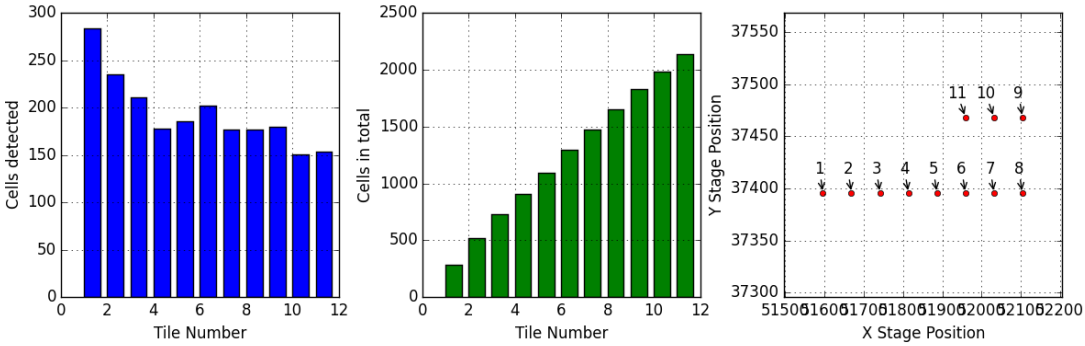


Figure 22: Display the content of the log file using a Python Script

This example demonstrates one the basic possibility to change the course of an ongoing experiment based on the results obtained by the online image analysis. Due the flexibility of the Python scripting language all kind of custom parameters can be calculated. Therefore the actual decision on how to adapt the experiment can be based on a real sophisticated decision tree if that is required by the application.

4.4 Displaying Data Online using a Heatmap

During the course of an experiment it can be very useful to already check some experiment or image analysis parameters during runtime in order to be able to tell if everything is "fine". This is a typical use case for online data display.

Idea or Task:

- Define an image acquisition - image per well
- Start the data display right away together with the experiment
- Count number of objects per well (or image)
- Constantly monitor the created data log file and display the data

Definition of Feedback Experiment

For this example one can use the sample image folder ...\CCD_Testimages_Folder\96_well again. It contains 96 single images with cells where the nucleus is stained with DAPI and allows to simulate a well plate, even if one has to real setup (Laptop only) available for demonstration purposes at a possible customers site.

- Positions = 96 (one image per well)
- Channel = DAPI (or whatever the nuclei are stained with ...)

For demonstration purposes this example "just pretends" to acquire images from 96 distinct wells with simulated hardware. The experiment itself is called EF_Count_Cells_96well_Heatmap_dy
Note that the stage motion follows a "meander" scheme.

Definition of Image Analysis Pipeline

The image pipeline used for this example reads out the wellID in order to assign the data to the correct well. The pipeline Count_Cells_DAPI_96well.czias. is used in this case.

Creation of Feedback Script

The feedback script itself is straight forward. The most important steps are:

1. Get the filename of logfile .
2. Define the well type for the external data display script and start it.
3. Get the number of objects per well and create the log file.

4. Display the data logfile.

```
###   PreScript   ###

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
script = 'dynamic_plot_96well_sync.py' ## this is a external python application
ZenService.Xtra.System.ExecuteExternalProgram(script, filename + params)

###   LoopScript ###

# get number of cells from current image
cn = ZenService.Analysis.AllCells.RegionsCount
# get the current well name, column idex, row index and position index
well = ZenService.Analysis.AllCells.ImageSceneContainerName
col = ZenService.Analysis.AllCells.ImageSceneColumn
row = ZenService.Analysis.AllCells.ImageSceneRow
# create logfile
logfile = ZenService.Xtra.System.AppendLogLine(str(well)+'\t'+str(cn)+'\t'+str(col)+'\t'+str(row))

###   PostScript   ###

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

Creation of Python Script for the Data Display

This time the script is a bit more advanced in order to illustrate some typical scripting techniques.

```
63      def onTimer(self, evt):
64
65          datain = np.loadtxt(options.filename, delimiter='\t', usecols=(1,2,3))
66
67          try:
68              for i in range(0,datain.shape[0]):
69                  # only read the last entry from the data --> this is the last data point
70                  cn = datain[i,0]
71                  col = datain[i,1]-1 # numpy is zero-based ...
72                  row = datain[i,2]-1
73                  # update well matrix at the correct position
74                  self.well[row, col] = cn
75
76              # do the plot
77              self.im = self.ax1.imshow(self.well, vmin = datain[:,0].min(),
78                                        vmax = datain[:,0].max(), cmap=cm.jet, interpolation='nearest')
79              self.fig.colorbar(self.im, cax=self.cax, orientation='vertical')
80              self.canvas.draw()
81
82          except:
83              print 'No data to display yet ...'
84
85          # save plot only when done and only once
86          if (datain.shape[0] == self.Nr * self.Nc and self.figure_saved == False):
87              self.fig.savefig(savename)
88              self.figure_saved = True
```

The part of the script above shows the important part of the script, where the data are imported and converted into a 2D array. The trick here is to use the well column and row index to assign the cell number to the correct well.

Run the Feedback Experiment and watch the results

Start the tile acquisition and watch heatmap fill up with the detected number of objects per well. The number of detected cells is represented as a color here. Red corresponds to a high number of objects, while blue is a low object count. The final result is automatically save as a PNG file inside the same folder.

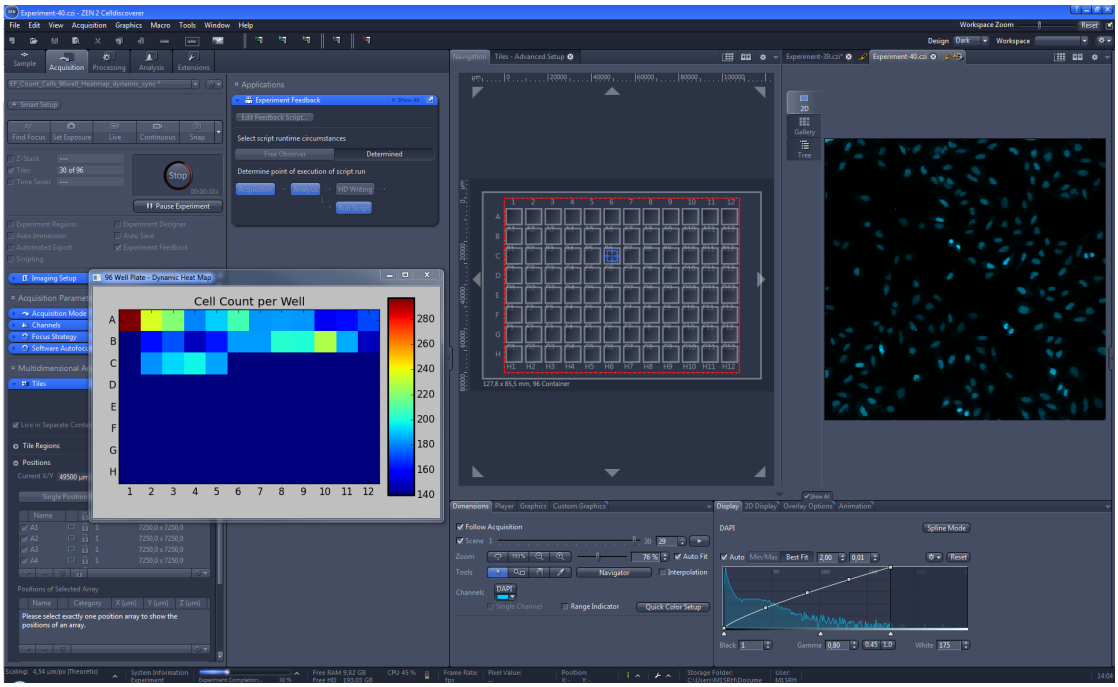


Figure 23: The heatmap is updated constantly during the course of the experiment by reading the logfile.

Finally the data logfile is displayed inside the text editor of your choice (Do not forget to adapt the feedback script, when using something else than Notepad++).

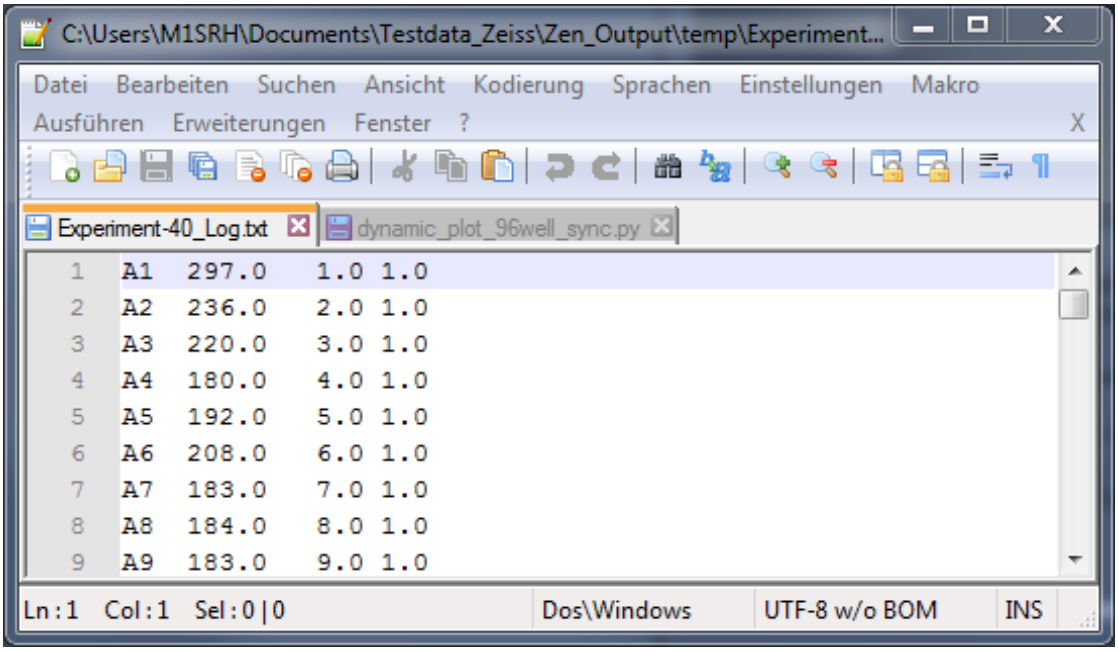


Figure 24: The data inside this logfile are used to create the online plot.

4.5 Jump to next Well

This experiment is basically a variant of the example 4.4. The main difference here is the idea to stop acquiring images within a specific well when a certain condition is met. Such a condition could be the total numbers of objects, e.g. cells, that were already detected.
Or one could also stop the acquisition when the object number after n tiles was still below a certain threshold, e.g. "the wells seems to be empty ..."

Idea or Task:

- Define a wellplate experiment with large number of tiles per well.
- Sum up the number of objects per well.
- jumpt to next well when a limit was reached.

Definition of Feedback Experiment

Basically the same as in example 4.4. The experiment used is called EF_JumpToNextWell.czexp. Important is to choose the Determined execution mode.

Definition of Image Analysis Pipeline

Here a slightly different image pipeline is used which also reads out the wellID in order to assign the data to the correct well. The pipeline Count_Cells_DAPI_Jump_Well.czias is used in this case.

Creation of Feedback Script

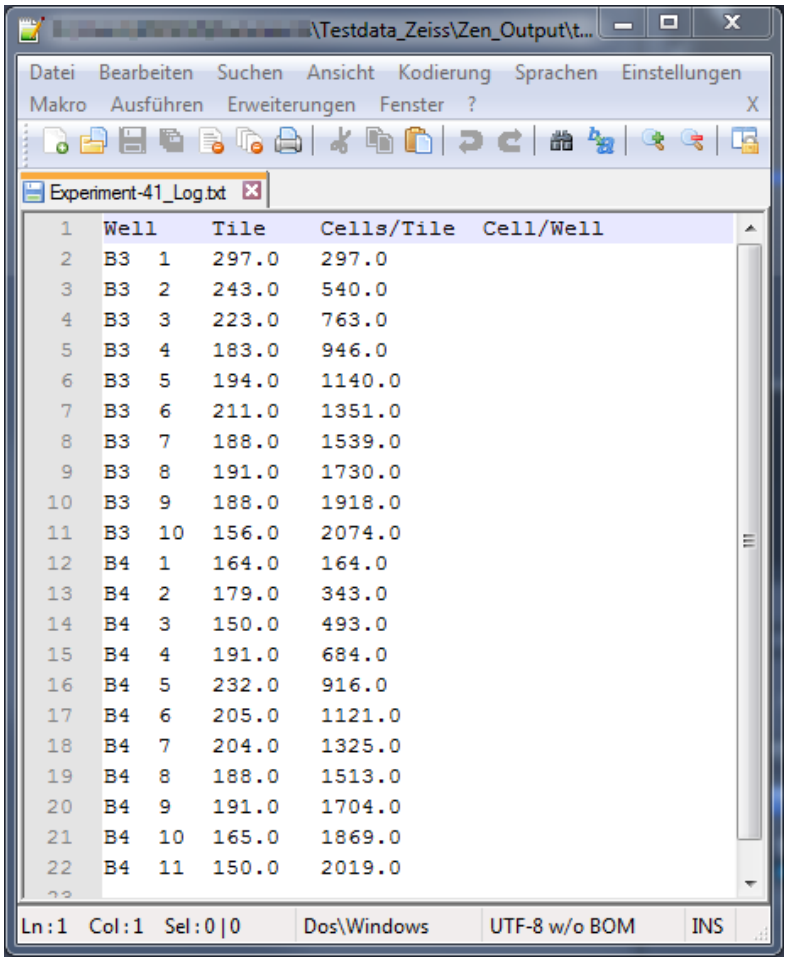
The feedback script itself is straight forward. The most important steps are:

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.
3. Jump to the next well when the limit was reached.
4. Reset the tile index.

```
###   PreScript   ###

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

###   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

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

# stop if the desired cell number was already reached
if (cells_per_well > 2000):
    ZenService.Actions.JumpToNextContainer()

# 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

###   PostScript   ###

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

Run the Feedback Experiment and watch the results

The original experiment was configured to acquire 50 tiles per well. As one can see above the **JumpToNextContainer** command was called after 10 tiles acquired within in well B3. The accumulated number of detected cells was 2074 and therefore the jump command was called. Inside well B4 the required limit of 2000 cells was reached after 11 tiles and the experiment was finished here.



	Well	Tile	Cells/Tile	Cell/Well
2	B3	1	297.0	297.0
3	B3	2	243.0	540.0
4	B3	3	223.0	763.0
5	B3	4	183.0	946.0
6	B3	5	194.0	1140.0
7	B3	6	211.0	1351.0
8	B3	7	188.0	1539.0
9	B3	8	191.0	1730.0
10	B3	9	188.0	1918.0
11	B3	10	156.0	2074.0
12	B4	1	164.0	164.0
13	B4	2	179.0	343.0
14	B4	3	150.0	493.0
15	B4	4	191.0	684.0
16	B4	5	232.0	916.0
17	B4	6	205.0	1121.0
18	B4	7	204.0	1325.0
19	B4	8	188.0	1513.0
20	B4	9	191.0	1704.0
21	B4	10	165.0	1869.0
22	B4	11	150.0	2019.0

Figure 25: The data logfile shows the "jump" after the object limit was reached.

4.6 Adapt the Exposure Time during Image Acquisition

Idea or Task:

- Modify the exposure time while the experiment is running.

Definition of 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 based on a certain event. To test it one can just use a simple time lapse.

- Exposure Time = 50ms (one image per well)

The other settings really depend on your sample and hardware settings. The experiment example itself is called EF_Brighter_and_Brighter.czexp

Creation of Feedback Script

The feedback script itself is straight forward and fulfills the following tasks:

1. Initialize variable for exposure time with a certain value
2. Set new value depending on the progress of the experiment

```
1  ###   PreScript   ###
2
3  # define initial exposure time
4  exposure = 50
5
6  ###   LoopScript ###
7
8  # get the current time index
9  index  = 1 + ZenService.Experiment.CurrentTimePointIndex * 0.2
10 # increase the exposure time
11 ZenService.Actions.SetExposureTime(1, index*exposure)
12
13 ###   PostScript   ###
```

Run the Feedback Experiment and watch the results

For this example the rest is straight forward. Just start the experiment and use Gallery View to see the effects. If setup correctly one will see the increased brightness for every frame.

4.7 Do Something depending on the Experiment Block

This example is juts to demonstrate the usage of a feedback script in connection with heterogeneous experiments.

Idea or Task:

- Define an experiment
- Do something different for every block inside the experiment

Definition of Feedback Experiment

This example contains a simple time lapse, which is identical for all three experiment block. 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 based on a certain event. To test it one can just use a simple time lapse. The experiment is called **EF_Cell_Count_Blocks.czexp**.

Definition of Image Analysis Pipeline

Here one can use a similar analysis pipeline as for the example 4.1in order to count the number of cells.

Creation of Feedback Script

```
###   PreScript   ###

# enter blocks to analyze here
blocks2do = [1,3]
# enter blocks to do nothing
blocksnot2do = [2]
# create header for logfile
logfile = ZenService.Xtra.System.AppendLogLine('Frame\tCells\tBlock')

###   LoopScript ###

# get current block index
block = ZenService.Experiment.CurrentBlockIndex

if block in blocks2do:

    # get current frame number
    frame = ZenService.Experiment.CurrentTimePointIndex
    cn = ZenService.Analysis.Cells.RegionsCount
    logfile = ZenService.Xtra.System.AppendLogLine(str(frame)+'\t'+str(cn)+'\t'+str(block))

elif block in blocksnot2do:
    ZenService.Xtra.System.PlaySound()


###   PostScript   ###

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

At the very beginning one defines, for which block we want to carry out a certain task or action. During experiment runtime, one can just check the current block index and execute one part of the feedback script or the other.

Run the Feedback Experiment and watch the results

The resulting TXT file only contains information for the desired blocks.

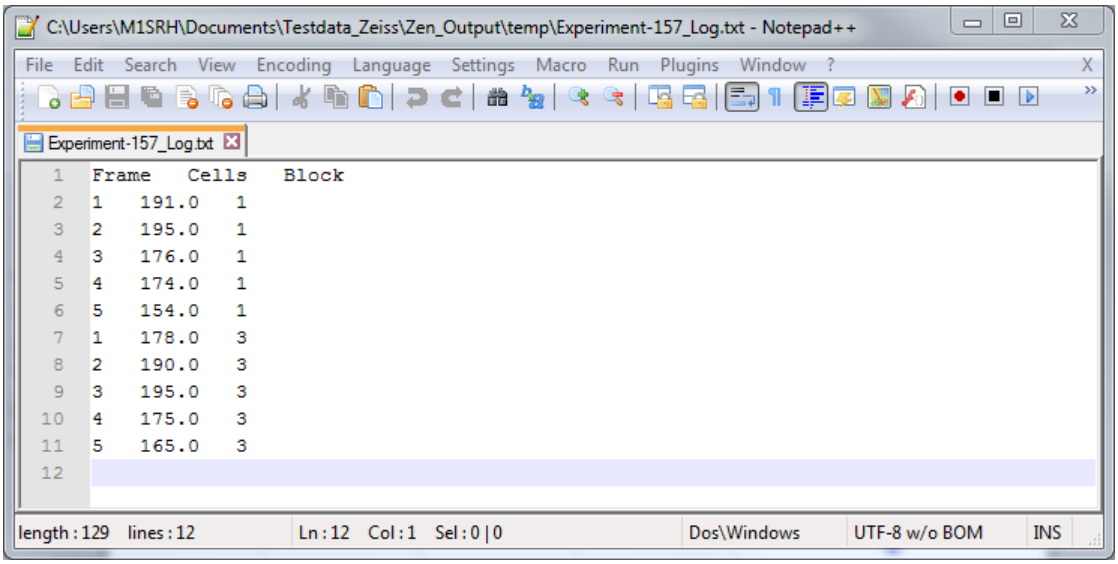


Figure 26: The resulting data logfile only contains information for block 1 and 3

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. Some knowledge of simple programming techniques is required to understand the idea of this feature.

Idea or Task:

- Define an time lapse acquisition.
- Locate the object of interest and define the desired Z-Stack.
- Acquire the complete time lapse for every z-plane inside the stack.

Usually ZEN does not allow to acquire a complete time lapse for every z-plane inside a z-stack. So one way to solve this problem is to move the focus position while repeating the time lapse itself.

Definition of Feedback Experiment

This script does not include an image analysis pipeline. Important is the use of the experiment designer. The number selected loops represent the number of planes of the desired z-Stack.

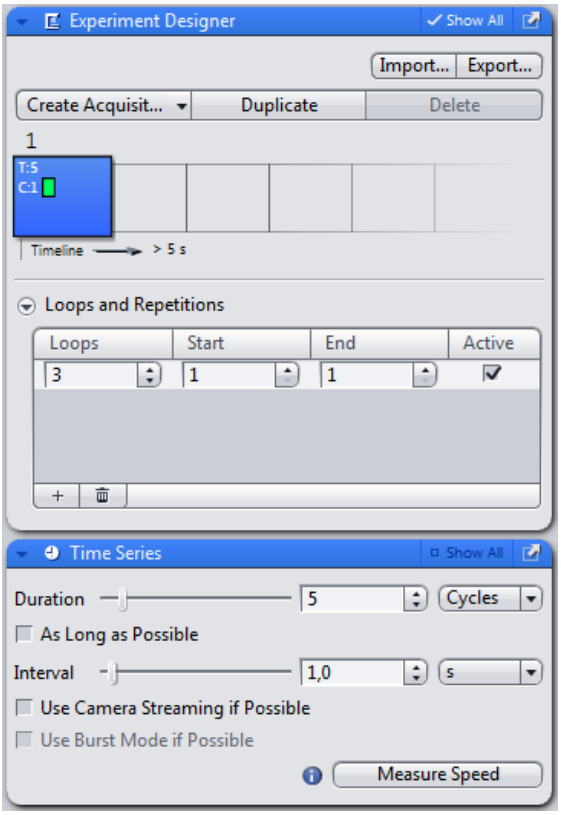


Figure 27: The data logfile shows the "jump" after the object limit was reached.

From the z-Stack tool one can easily extract the required parameters for the z-stack, but for the actual experiment the built-in Z-Stack tool cannot be used. The stack properties have to be defined manually inside the feedback script (number of planes, starting z-position and delta-Z). The corresponding experiment is called **EF_Timelapse_per_Z-Plane.czexp**.

Creation of Feedback Script

The most important task within the script is to make sure to move to the next Z-plane only after the desired number of time points was acquired.

- 3 Experiment-Designer Loops x 5 time points = 15 time point total.
- 5 time points per z-plane.
- Check, if 15 / Frame Number gives 1,2 or 3 - if YES, move Z-position.

```
###   PreScript   ###

loopcount = 0 # this variable is required to keep track of the current loop from the Experiment Designer
lastframe = 0
# the z-Stack has to be defined manually !
zstart = 0 # must be the coordinate of the 1st zplane in micron
deltaz = 5.0 # spacing for the z-stack in micron
timepoints = 5 # number of timepoints from Time Series Toolwindow!

###   LoopScript ###

# calculate the desired z-position
zpos = zstart + deltaz * loopcount

# get the current overall frame number
frame = ZenService.Experiment.CurrentTimePointIndex

# create a logile (this is optional)
logfile = ZenService.Xtra.System.AppendLogLine(str(loopcount+1)+'\t'+str(frame)+'\t'+str(zpos))

# only do the following for every new time point ...
if (frame != lastframe):

    # check, if the devision of overall frame / timepoints per plane = 0
    rest = divmod(frame, timepoints)
    # and if YES ...

    if (rest[1] == 0):
         # increase the loop counter
        loopcount = loopcount + 1
        # calc the new z-position for the next timelapse
        zpos = zstart + (deltaz * loopcount)
        # finally set new focus position
        ZenService.HardwareActions.SetFocusPosition(zpos)

    # update the frame to be ready for the next iteration
    lastframe = frame

###   PostScript   ###

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

Run the Feedback Experiment and watch the results

Just start the experiment as usual. Be sure to have the Experiment Feedback and the Experiment Designer enabled. All the image data end up in one CZI file, where the block slider corresponds to the z-dimension for this special experiment.

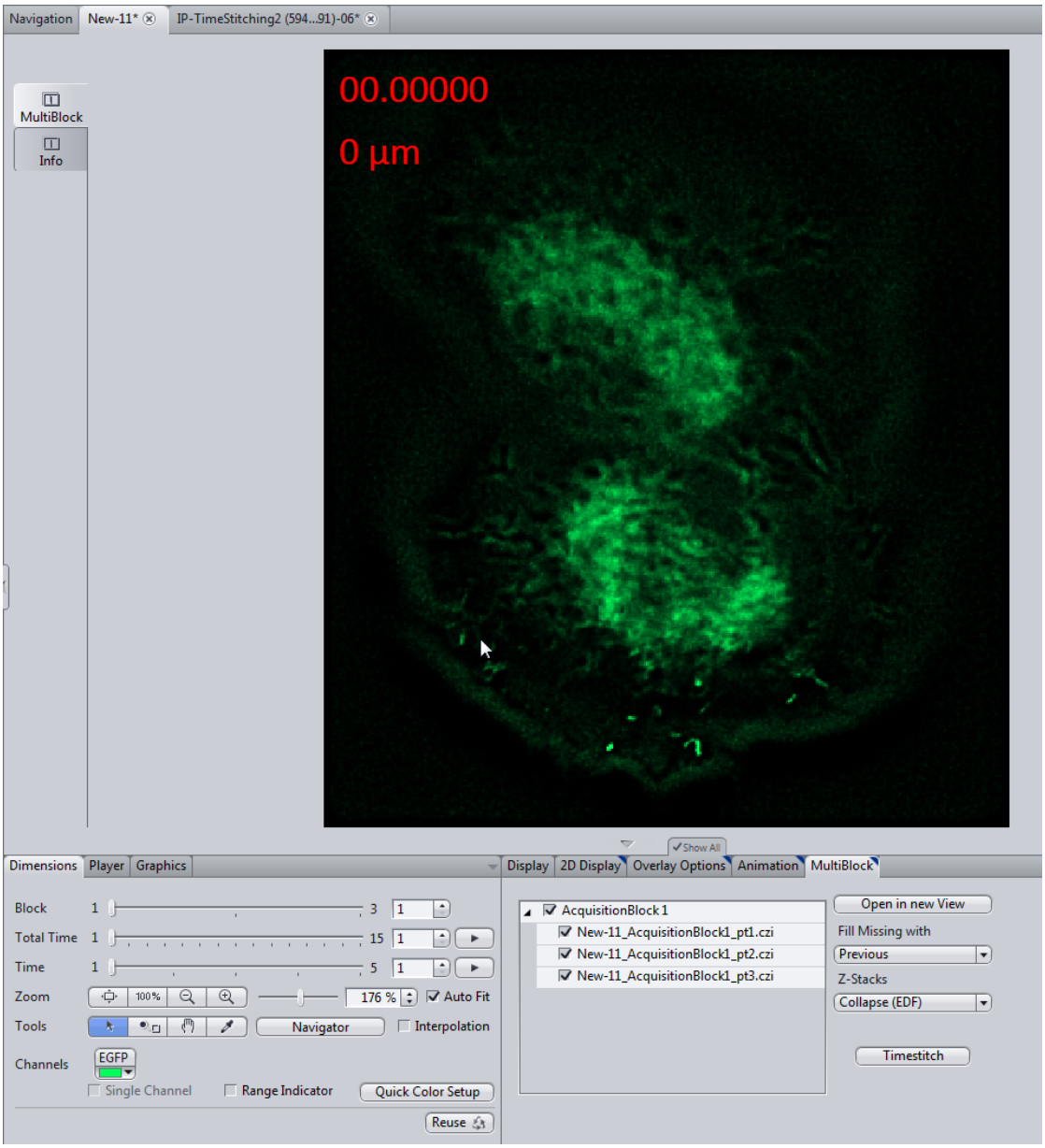


Figure 28: The resulting CZI s out of 3 acquisition blocks.

1. 1st Block = 1st Z-Plane **z=0µm** includes 5 time points.
2. 2nd Block = 2nd Z-Plane **z=5µm** includes 5 time points.
3. 3rd Block = 3rd Z-Plane **z=10µm** includes 5 time points.

Remark: The screen shot was produced using the TestCam feature, so the image data for every block "look" the same. It is only a simulated Z-Stack.

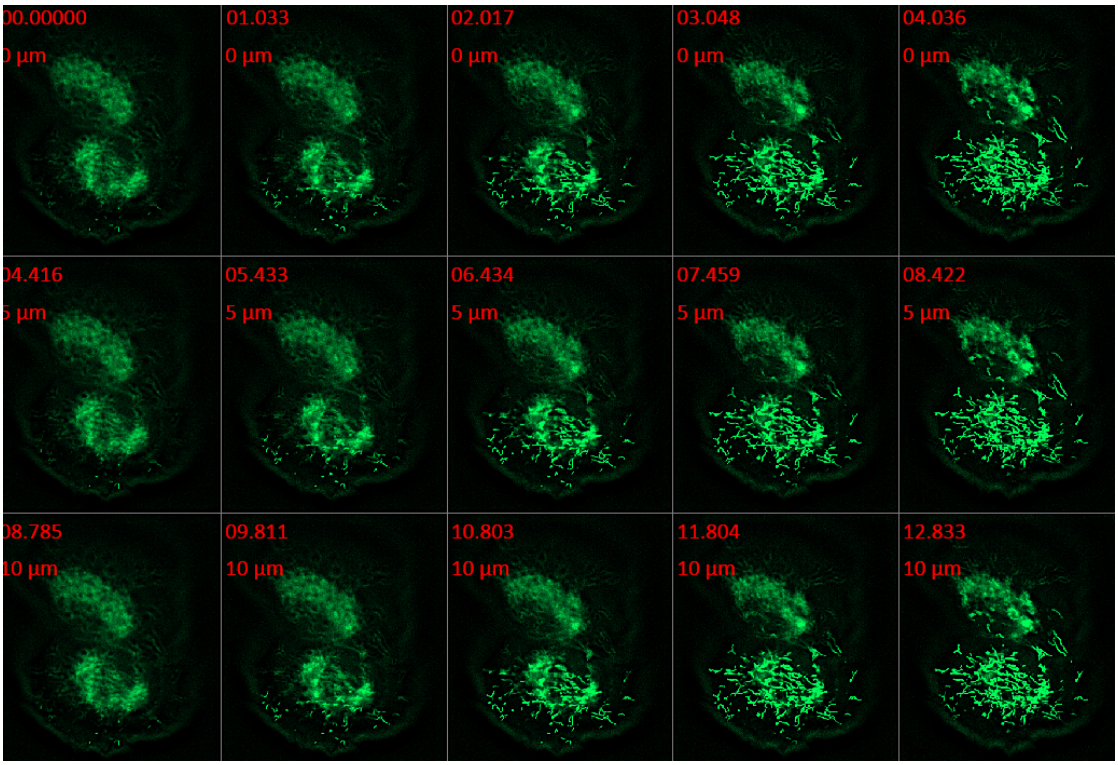


Figure 29: After TimeStitching the Gallery View reveals the correct result.

4.9 Automatic Event Detection

This example illustrates the possibility to react upon a change of intensity inside automatically detected cells. Keep in mind that the intensity parameter can be easily exchanged for something else.

Idea or Task:

- Define an time lapse acquisition.
- Setup an image analysis pipeline to detect the events.
- Define the Feedback Script and specify the detection event inside script.
- Detect a significant change in the specified parameter and take an action.

Normally the user cannot really predict, when the activation event will happen And for the example the idea is to acquire the images slowly and speed up upon the activation event. The crucial point is, that one does not know beforehand, when this event will occur.

Definition of Feedback Experiment

For this example use the sample image folder **...\Testimages\Jurkat_Activation_Detection.** This contains 70 single TIFF images with Jurkat cells stained Fluo-4. Just define a single channel and test if the sample camera gives the correct output. Define your experiment with the following parameters:

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

This script requires an automated image analysis pipeline **Detect_Active_Jurkat_Cells_New.czias** which uses a simple threshold to detect activate cells.

Creation of Feedback Script

The main tasks for this 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.

- Optionally one can create a log file for testing purposes.

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

```
###   PreScript   ###

lastindex = 0
cn_last = 0
soundfile1 = r'C:\TFS\Doc\3-ZIS\3-Development\Discussions\ExperimentFeedback\Release_DVD\SoundFiles\PsychoScream.wav'
soundfile2 = r'C:\TFS\Doc\3-ZIS\3-Development\Discussions\ExperimentFeedback\Release_DVD\SoundFiles\YEAH.WAV'

###   LoopScript ###

# get parameters
frame = ZenService.Analysis.Cells.ImageIndexTime
cn = ZenService.Analysis.Cells.RegionsCount
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 (delta > 0): ## active cell number has increased
    #  play soundfile 1 --> this could be anythin else, e.g. sent a TTL to port XY
    ZenService.Xtra.System.PlaySound(soundfile2)

elif (delta < 0): ## active cell number has decreased
    # play soundfile 2 --> just a placeholder for a more meaningful action
    ZenService.Xtra.System.PlaySound(soundfile1)

###   PostScript   ###

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

# additional script execution to display the data directly from the feedback sricpt
filename = '-f ' + ZenService.Experiment.ImageFileName[:-4] + '_Log.txt'
script = r'c:\TFS\Doc\3-ZIS\3-Development\Discussions\ExperimentFeedback\Release_DVD\Python_Scripts_for_Data_Display\display_ju
    rkat.py'
ZenService.Xtra.System.ExecuteExternalProgram(script, filename)
```

Creation of Python Script for the Data Display

As usual this is an optional task, but it shows once more the flexibility of Python to display the data. The content of the data logfile is displayed as a graph which is also saved to disk.

```
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")

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

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


# load data
data = np.loadtxt(options.filename, delimiter='\t')
# create figure
figure = plt.figure(figsize=(12,6), dpi=100)
ax1 = figure.add_subplot(211)
ax2 = figure.add_subplot(212)

# create subplot 1
ax1.bar(data[:,0],data[:,1],width=0.8, bottom=0)
ax1.grid(True)
ax1.set_xlim([0,data[:,0].max() + 1])
ax1.set_ylim([0,data[:,1].max() + 0.5])
ax1.set_ylabel('Cells Avtive')

# create subplot 2
ax2.bar(data[:,0],data[:,2], width=0.8, bottom=0, color = 'red')
ax2.set_xlim([0,data[:,0].max() + 1])
ax2.set_ylim([data[:,2].min()-0.5,data[:,2].max()+0.5])
ax2.set_xlabel('Frame Number')
ax2.set_ylabel('Delta')
ax2.grid(True)
figure.subplots_adjust(left=0.10, bottom=0.12, right=0.95, top=0.95,wspace=0.10, hspace=0.20)

# save figure
plt.savefig(savename)

# show graph
plt.show()
```

Run the Feedback Experiment and watch the results

Activate the experiment feedback under Automation and start the experiment. As soon as a new cell "lights up" a sound file will be played. And also the opposite event is detected. When the intensity inside a cell drops below the threshold, a different sound file will be played.

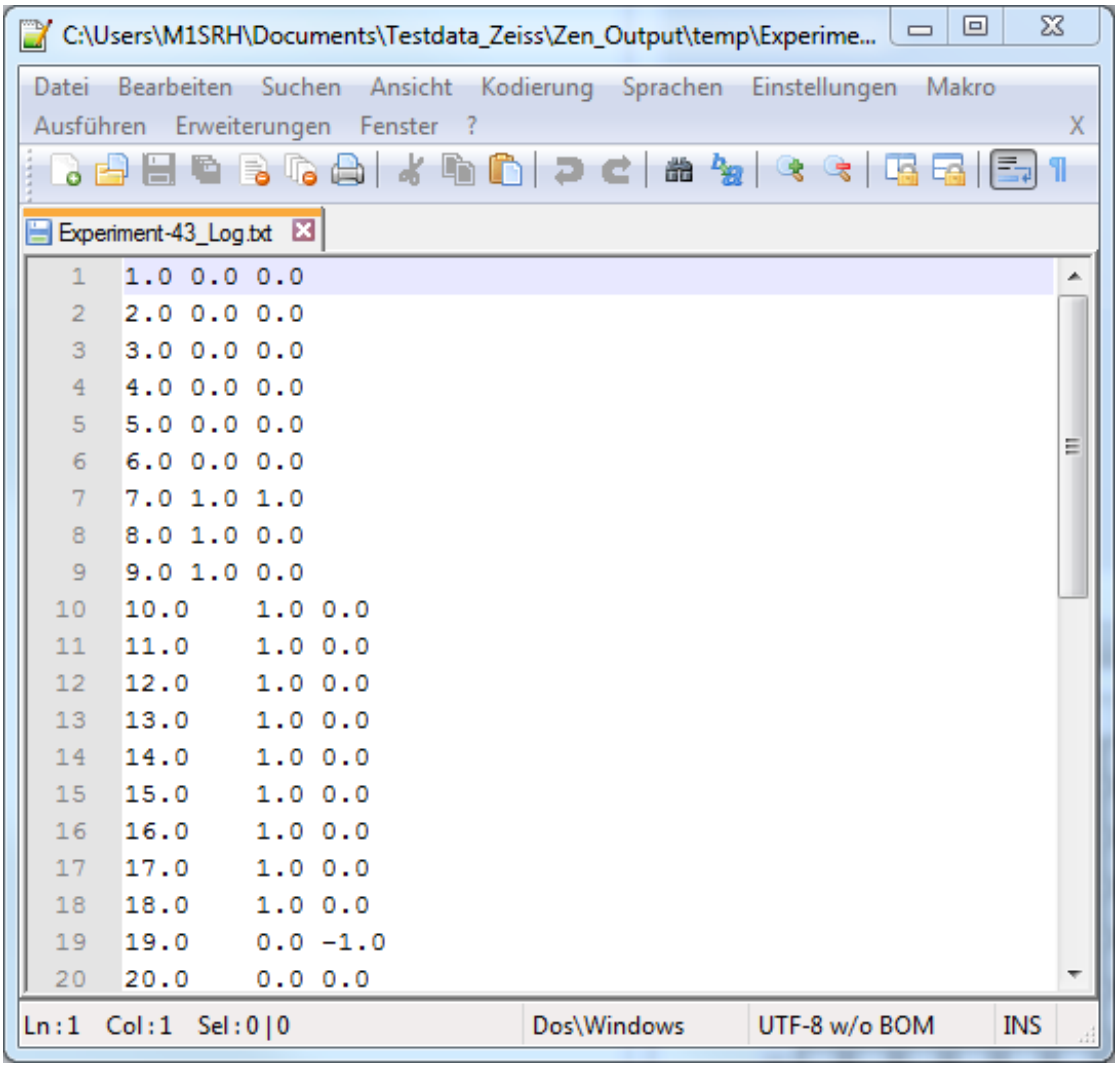


Figure 30: The resulting data log file shown (de)activation events.

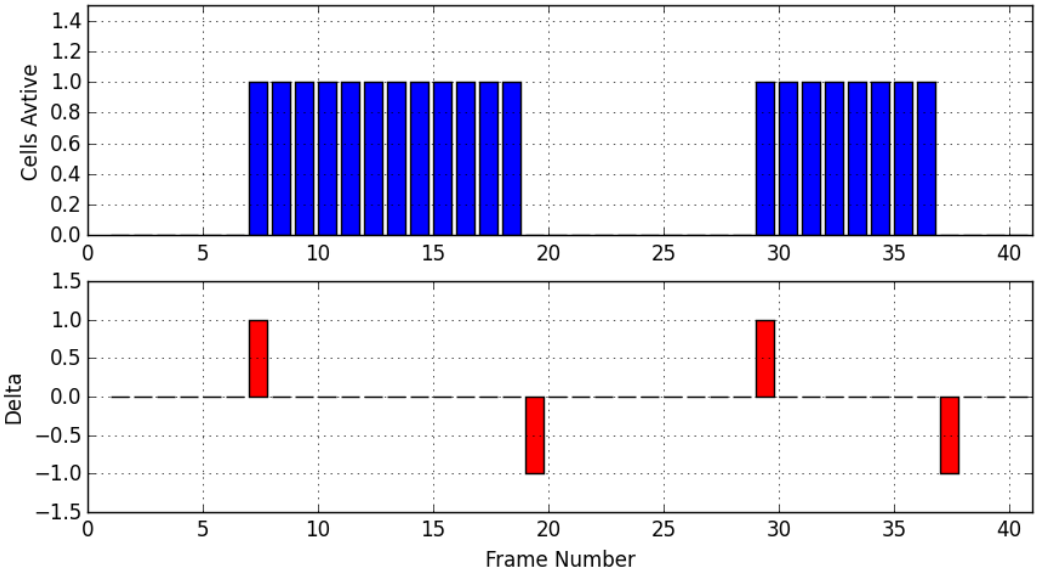


Figure 31: The data from the logfiles displayed as a graph.

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 ZEN image analysis to auto-detect the cells
- Get the parameters from the analysis, do some math and plot the data online.

Definition of Feedback Experiment

For this example use the sample image folder **...\Testimages\Calcium_340-380_100frames**. This contains 100 single TIFF images with artificial objects. Just define a one channel time series experiment to simulate the object movement:

- Cycles = 100
- Time Interval = 500ms
- Channel 1 = Fura WL1
- Channel 2 = Fura WL2

The image analysis pipeline to be used is **Online_Cell_Detect.czias**.

Creation of Feedback Script

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

- Initialize the required variables.
- Create a function that can divide System.Array.
- 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 please read the comments inside the script, which explain the single steps in more detail. The corresponding experiment is called **EF_Ratio_Feedback_Online-2.czexp**.

```
1 ### PreScript ###
2
```

```
from System import Array

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

options = '-f ' + ZenService.Experiment.ImageFileName[:-4] + '_Log.txt' + ' -p all'
script = r'c:\TFS\Doc\3-ZIS\3-Development\Discussions\ExperimentFeedback\Release_DVD\Python_Scripts_for_Data_Display\Dynamic_MeanROI_Cell_Detect.py'
ZenService.Xtra.System.ExecuteExternalProgram(script, options)

###   LoopScript ###

frame = ZenService.Analysis.AllCells340.ImageIndexTime
int340 = ZenService.Analysis.SingleCell340.IntensityMean_F2_340nm
int380 = ZenService.Analysis.SingleCell340.IntensityMean_F2_380nm
ratio, ratiostr = ArrayDiv(int340, int380)
logfile = ZenService.Xtra.System.AppendLogLine(str(frame) +'\t'+ratiostr)


###   PostScript   ###

# open logfile at the end (optional)
ZenService.Xtra.System.ExecuteExternalProgram(r'C:\Program Files (x86)\Notepad++\notepad++.exe', logfile)
```

Creation of Python Script for the Data Display

This time the script is a bit more advanced in order to illustrate some typical scripting techniques. The most important part in the **on_timer** function. It will be called every ... seconds and read the data, if the data set (logfile) has changed. In case of a change, the dataset is plotted.

Finally, if there are no more new data arriving, the plots are colorized and saved as an PNG file.

```
from pylab import *
import numpy as np
import optparse
import os
import time
from PyQt4.QtCore import *
from PyQt4.QtGui import *
# Qt4 bindings for core Qt functionalities (non-GUI)
from PyQt4 import QtCore
# Python Qt4 bindings for GUI objects
from PyQt4 import QtGui

# import the MainWindow widget from the converted .ui files
import ui_Ratio_Online_Plot2

frequency = 0.7 # update frequency in ms

class Ratio_Online_Plot(QtGui.QWidget, ui_Ratio_Online_Plot2.Ui_Ratio_Online_Plot):

    def __init__(self, parent=None):
        super(Ratio_Online_Plot, self).__init__(parent)
        self.setupUi(self)
        # add title & legends for plots
        self.setLegends()
        try:
            self.ct_last = os.stat(options.filename).st_mtime
        except:
            self.ct_last = 0.0

        self.nodatacount = 0


    def on_start(self):

```

```
        # create timer object
        self.timer = QTimer()
        # connect signals with slots
        self.connect(self.timer, SIGNAL('timeout()'), self.on_timer)
        # update timer every ... miliseconds
        self.timer.start(frequency * 1000)


    def on_timer(self):
        """
        Executed periodically when the monitor update timer is fired.
        """

        ## get file modification time
        ct = os.stat(options.filename).st_mtime
        ## check if the time is bigger than the last one
        if ct > self.ct_last:

            # read the actual data
            data = np.genfromtxt(options.filename, dtype=float, invalid_raise=False,delimiter='\t')
            # update ratio plot
            if (options.plotoption == 'all'):
                # plot all columns
                self.ratioplot.axes.plot(data[:,0],data[:,1:], 'r-', lw=2, label = 'Ratio')
            elif (options.plotoption != 'all'):
                # plot one column only - the index must be passed as a string
                cl = np.int(options.plotoption)
                self.ratioplot.axes.plot(data[:,0],data[:,cl:], 'ro-', lw=2, label = 'Ratio')

            # update plots
            self.ratioplot.draw()
            self.ct_last = ct

        ## check if the current btime is equal to the last one
        elif ct == self.ct_last:

            print 'NO NEW DATA ANYMORE'
            self.nodatacount = self.nodatacount + 1

        ## close app and save figure if ... time no change was detected
        if self.nodatacount > 10:

            ## read the data one more time and colourize the plot
            data = np.genfromtxt(options.filename, dtype=float, invalid_raise=False,delimiter='\t')
            self.ratioplot.axes.plot(data[:,0],data[:,1:], '-', lw=2, label = 'Ratio')
            self.ratioplot.draw()
            ## save figure before closing
            time.sleep(5)
            self.ratioplot.figure.savefig(savename)
            sys.exit()

    ## format the figure
    def setLegends(self):

        # add title & legends for ratio plot
        self.ratioplot.axes.set_title('Ratio CH1/CH2')
        self.ratioplot.axes.set_xlabel('Frame')
        self.ratioplot.axes.set_ylabel('Ratio')
        self.ratioplot.axes.grid(True)
        self.ratioplot.axes.hold(True)

def main():

    app = QApplication(sys.argv)
    form = Ratio_Online_Plot()
    form.show()
    form.on_start() # start data monitoring right away
    app.exec_()

if __name__ == "__main__":

    # 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.")
    # plot options
    # 'all' --> plot all columns
    # '3' --> plot column 3 only
    parser.add_option('-p', '--plot',
    action="store", dest="plotoption",
    help="query string", default="No plotoption passed.")
    # read command line arguments
    options, args = parser.parse_args()
    print 'Filename:', options.filename
    savename = options.filename[:-4] + '.png'
    print 'Savename: ', savename
    # this actually start the application
```

```
123     main()
```

Run the Feedback Experiment and watch the results

Activate the Experiment Feedback checkbox as usual and start the experiment. During the experiment the online display will as shown below.

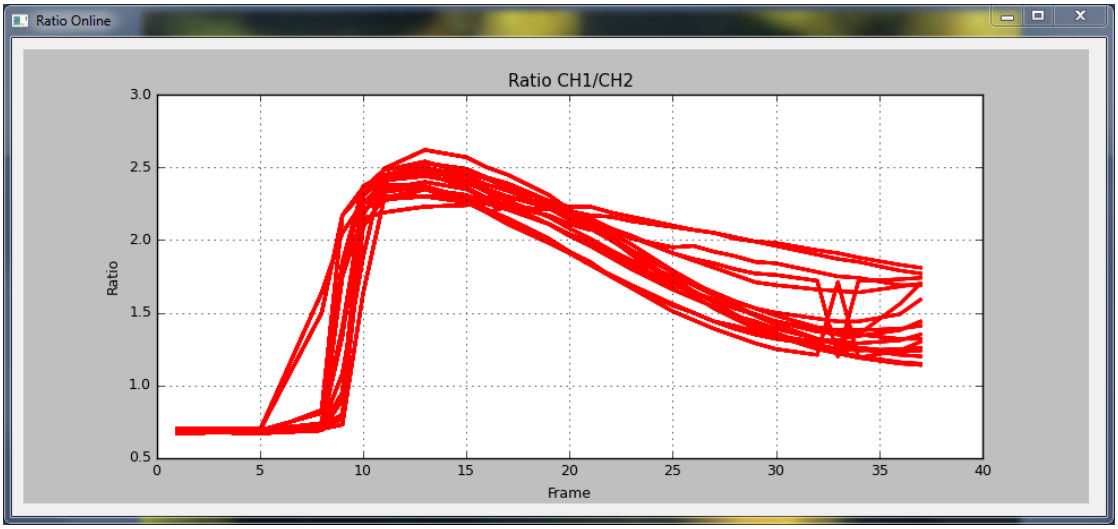


Figure 32: The resulting online data display showing the ratio.

This is the final PNG image file created by the python online plotting script. Note the difference to the screenshot above taken at frame 37. The final image contains the plots for all 100 frames.

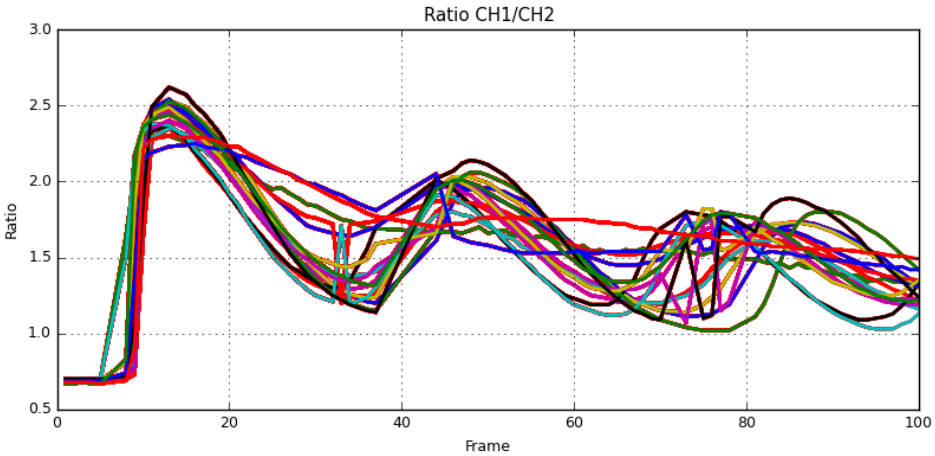


Figure 33: The ratio graph will be saved as a PNG file.

4.11 Online Tracking

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

Idea or Task:

- Define an time lapse acquisition.
- Use ZEN image analysis to figure out which particle is the "brightest"
- Use the XY coordinates of this particle to move the XY stage.

Definition of Feedback Experiment

For this example use the sample image folder ...\Testimages\FakeTracks. This contains 30 single TIFF images with artificial objects. Define your experiment with the following parameters:

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

In order to track the objects one must set up an image analysis pipeline to retrieve the xy stage coordinates of the object. For this example the actual tracking is just a threshold based object detection. There is no linking between objects between adjacent frames implemented. Therefore this simple analysis might lead to a jumping objectID. The basic steps used for this example are:

- Do some Image Pre-Processing.
- Detect the objects using an automatic threshold.
- Get all intensities of all detected objects.
- Find the index of the brightest object for every frame.
- Use the index get the XY coordinates for the brightest object out of an array.
- Move the center of the stage accordingly.

Parts of the used image analysis pipeline **tracking.czias** are show below.

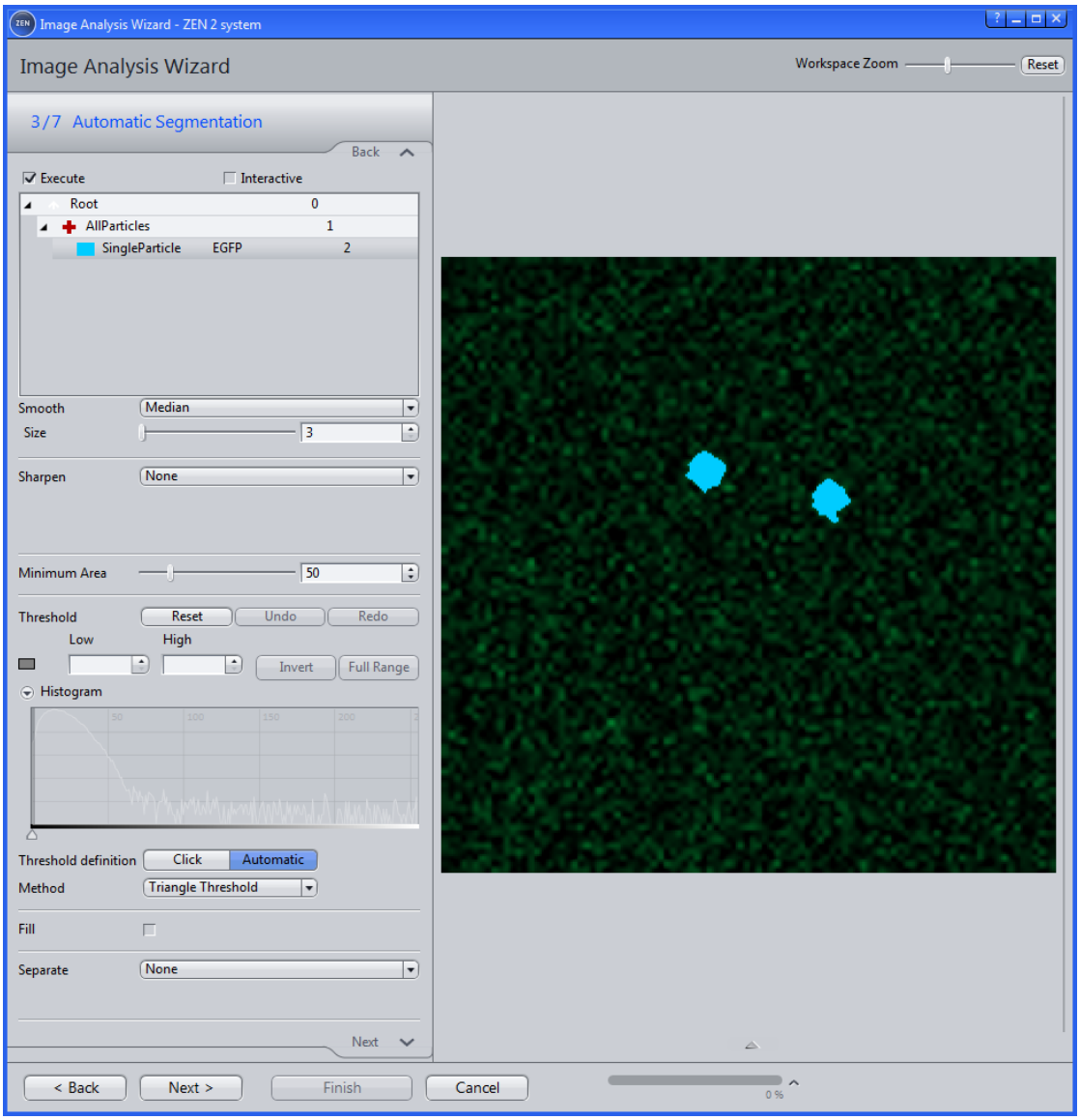


Figure 34: Simple thresholding is used to detect the objects.

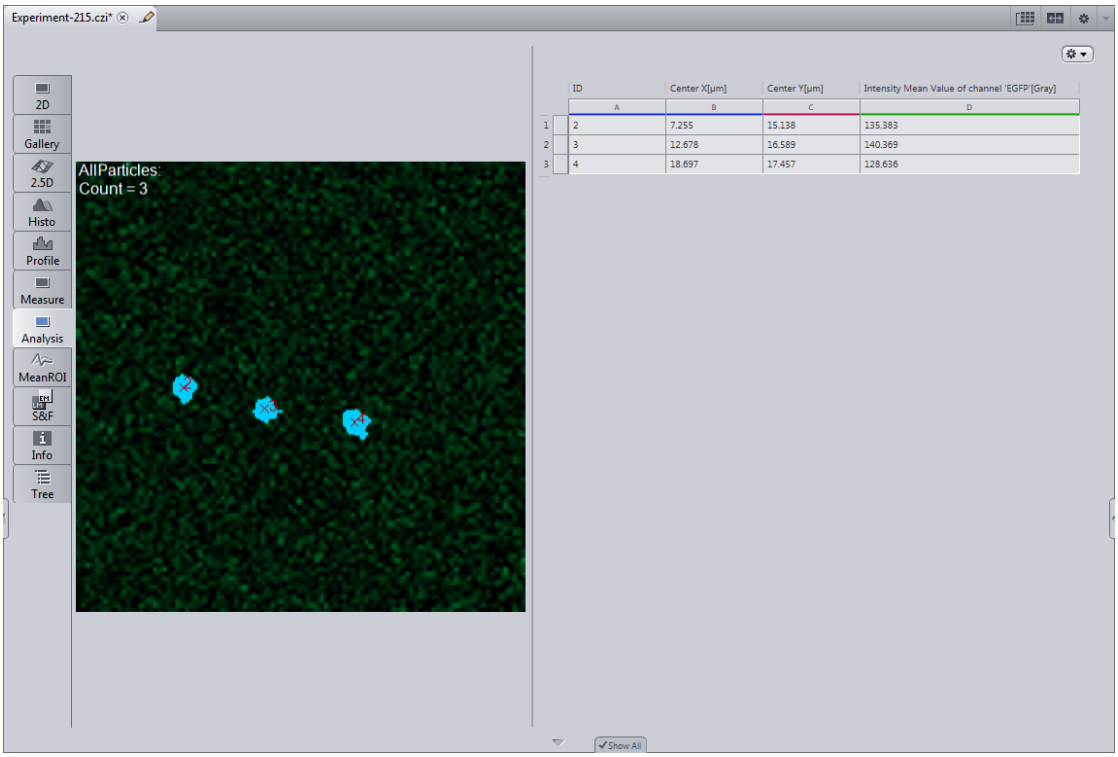


Figure 35: Results for the Image Analysis for an example frame.

Creation of 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 (this is actually optional).
- Get the array containing all the XY coordinates and intensities for **all single objects**.
- Find the brightest particle and the respective XY values using the array index.
- Move the stage to place the brightest object in the center of the field of view.

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

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


from System import Array

# define header for the oitput logfile to your needs - Timeframe - ObjectID - ObjStageX - ObjStageY
logfile = ZenService.Xtra.System.AppendLogLine('T\tObjID\tStageX\tStageY')

# creates a readable string from all entries of an array
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

# get current stage position XY of the image center
imgX = ZenService.Analysis.AllParticles.ImageStageXPosition
imgY = ZenService.Analysis.AllParticles.ImageStageYPosition

# get current object positions and intensity arrays for all detected objects
posx = ZenService.Analysis.SingleParticle.BoundCenterXStage
posy = ZenService.Analysis.SingleParticle.BoundCenterYStage
intensities = ZenService.Analysis.SingleParticle.IntensityMean_EGFP

# get ID of the brightest detected particle
ID = Array.IndexOf(intensities, max(intensities))

# move the stage to the position of the brightest particle -->
ZenService.HardwareActions.SetStagePosition(posx[ID], posy[ID])

# create strings for optional output for testing --> comment when not used:
POSX = createstr(posx)          # array with all StageX positions
POSY = createstr(posy)          # array with all StageY positions
INTS = createstr(intensities)   # array with all intensities

# write positions to data log file
logfile = ZenService.Xtra.System.AppendLogLine(str(frame)+'\t'+str(ID+1)+'\t'+'{:.2f}'.format(posy[ID])+'\t'+'{:.2f}'.format(po
 ↪    sy[ID]))


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


# this is all optional
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 under Automation and start the experiment. As this is a simulated experiment with a simulated stage the movement will be result in shift of the overall image coordinate system. Inside ZEN Blue one will see a black edge around the actual image.

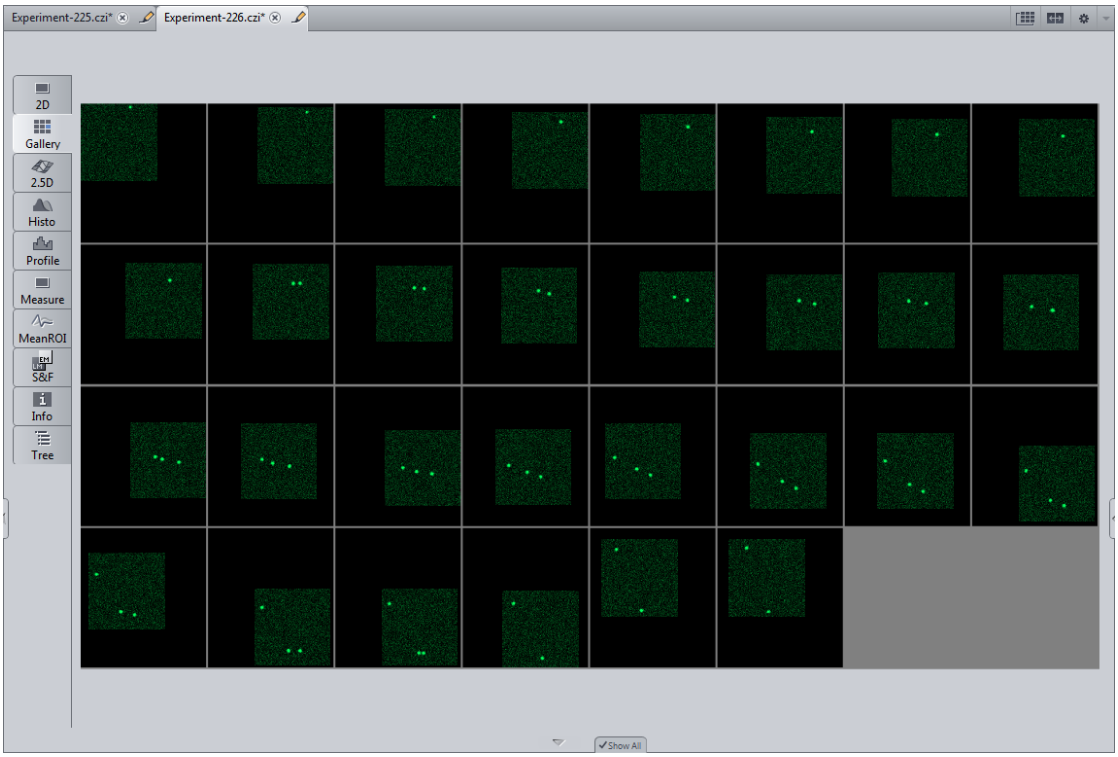


Figure 36: The resulting image data for simulated tracking.

5 Script Commands - Observables

It is crucial to understand that Observables are more than just values "that can be observed". They are required to trigger a script-run of the Loop Script. If there are no Observables used the script will not be executed at all.

5.1 Observables - Analysis

The available commands inside the analysis sections depend on the definition of the image analysis pipeline and the defined ROIs from Physiology. Therefore this is a dynamically created list that will always look different.

But it will only contain the shortcuts to those parameters, that are defined inside the CZIAS measurement file. Such a file can be created using the image analysis s wizard as described in 4.1 or using an OAD macro.

5.2 Observables - Environment

5.2.1 CurrentDateDay

Complete Command: **ZenService.Environment.CurrentDateDay**
Input:
Output: integer
Description: Returns the current day.

5.2.2 CurrentDateMonth

Complete Command: **ZenService.Environment.CurrentDateMonth**
Input:
Output: integer
Description: Returns the current month.

5.2.3 CurrentDateYear

Complete Command: **ZenService.Environment.CurrentDateYear**
Input:
Output: integer
Description: Returns the current year.

5.2.4 CurrentTimeHour

Complete Command: **ZenService.Environment.CurrentTimeHour**
Input:
Output: integer
Description: Returns the current hour.

5.2.5 CurrentDateTimeMinute

Complete Command: **ZenService.Environment.CurrentTimeMinute**
Input:
Output: integer
Description: Returns the current minute.

5.2.6 CurrentDateTimeMinute

Complete Command: **ZenService.Environment.CurrentTimeSeconds**
Input:
Output: integer
Description: Returns the current second.

5.2.7 FreeDiskSpaceInMBytes

Complete Command: **ZenService.Environment.FreeDiskSpaceInMBytes**
Input:
Output: double
Description: Returns the free disk space on the hard disk where the image experiment will is saved.

5.2.8 HasChanged

Complete Command: **ZenService.Environment.HasChanged(String observableId)**
Input:
Output: boolean
Description: Returns whether a certain observable has changed. On simple occurrences it is possible to use the HasChanged("observableId") method instead of a custom lock mechanism. Where available the HasChanged("observableId") method takes an observable name as parameter. HasChanged("observableId") returns true whenever the specified observable has changed since last script run; otherwise false. So it might not be suitable for situations when an observable changes very often and/or when really a singular execution is desired.

5.3 Observables - Experiment

5.3.1 CurrentBlockIndex

Complete Command: **ZenService.Experiment.CurrentBlockIndex**
Input:
Output: integer
Description: Returns the current experiment block index in heterogeneous experiments.

5.3.2 CurrentSceneIndex

Complete Command: **ZenService.Experiment.CurrentSceneIndex**
Input:
Output: integer
Description: Returns the index of the current position for multi-position experiments.

5.3.3 CurrentTileIndex

Complete Command: **ZenService.Experiment.CurrentTileIndex**
Input:
Output: integer
Description: Returns the current tile index.

5.3.4 CurrentTimePointIndex

Complete Command: **ZenService.Experiment.CurrentTimePointIndex**
Input:
Output: integer
Description: Returns the current time point index, e.g. the frame number from a time-lapse experiment.

5.3.5 CurrentTrackIndex

Complete Command: **ZenService.Environment.CurrentTrackIndex**
Input:
Output: integer
Description: Returns the current track number.

5.3.6 CurrentZSliceIndex

Complete Command: **ZenService.Experiment.CurrentZSliceIndex**
Input:
Output: integer
Description: Returns the current Z-Plane form the currently acquired stack.

5.3.7 ElapsedTimeInMinutes

Complete Command: **ZenService.Experiment.ElapsedTimeInMinutes**
Input:
Output: double
Description: Returns the elapsed time in minutes for the current experiment.

5.3.8 ElapsedTimeInSeconds

Complete Command: **ZenService.Experiment.ElapsedTimeInSeconds**
Input:
Output: double

Description: Returns the elapsed time in seconds for the current experiment.

5.3.9 ImageFileName

Complete Command: **ZenService.Experiment.ImageFileName**
Input:
Output: string
Description: : Returns the name of the current experiment as a String. The String ending will * .czi.

5.3.10 IsExperimentPaused

Complete Command: **ZenService.Experiment.IsExperimentPaused**
Input:
Output: boolean
Description: This returns the paused state of the current experiment, e.g. it will be True if the experiment is paused.

5.3.11 IsExperimentRunning

Complete Command: **ZenService.Experiment.IsExperimentRunning**
Input:
Output: boolean
Description: This returns the running state of the current experiment, e.g. it will be True if the experiment is running.

5.3.12 HasChanged

Complete Command: **ZenService.Environment.HasChanged(String observableId)**
Input:
Output: boolean
Description: Returns weather a certain observable has changed. On simple occurrences it is possible to use the HasChanged("observableId") method instead of a custom lock mechanism. Where available the HasChanged("observableId") method takes an observable name as parameter. HasChanged("observableId") returns true whenever the specified observable has changed since last script run; otherwise false. So it might not be

suitable for situations when an observable changes very often and/or when really a singular execution is desired.

5.4 Observables - Hardware

5.4.1 IncubationAirHeaterIsEnabled

Complete Command: **ZenService.Hardware.IncubationAirHeaterIsEnabled**
Input:
Output: Boolean
Description: Returns the state of the air heating system.

5.4.2 IncubationAirHeaterTemperature

Complete Command: **ZenService.Hardware.IncubationAirHeaterTemperature**
Input:
Output: Double
Description: Returns the current temperature setpoint.

5.4.3 IncubationChannelXIsEnabled (X=1-4)

Complete Command: **ZenService.Hardware.IncubationChannelXIsEnabled**
Input:
Output: Boolean
Description: Returns the state for a specific incubation channel.

5.4.4 IncubationChannelXTemperature (X=1-4)

Complete Command: **ZenService.Hardware.IncubationChannelXTemperature**
Input:
Output: Double
Description: Returns the temperature for a specific channel.

5.4.5 IncubationCO2Concentration

Complete Command: **ZenService.Hardware.IncubationCO2Concentration**
Input:
Output: Double
Description: Returns the current CO2 concentration setpoint.

5.4.6 IncubationCO2IsEnabled

Complete Command: **ZenService.Hardware.IncubationCO2IsEnabled**
Input:
Output: Boolean
Description: Returns the state of the CO2 incubation system.

5.4.7 IncubationO2Concentration

Complete Command: **ZenService.Hardware.IncubationO2Concentration**
Input:
Output: Double
Description: Returns the current O2 concentration setpoint.

5.4.8 IncubationO2IsEnabled

Complete Command: **ZenService.Hardware.IncubationO2IsEnabled**
Input:
Output: Boolean
Description: Returns the state of the O2 incubation system.

5.4.9 TriggerDigitalInX (X = ...)

The range of values for X depends on the IOcard configuration.

Complete Command: **ZenService.Hardware.TriggerDigitalInX**
Input:
Output: Boolean

Description: Returns the state of a specific TriggerDigitalIn port.

5.4.10 TriggerDigitalOutX (X = ...)

The range of values for X depends on the IOcard configuration.

Complete Command: **ZenService.Hardware.TriggerDigitalOutX**
Input:
Output: Boolean
Description: Returns the state of a specific TriggerDigitalOut port.

5.4.11 TriggerDigitalOutRLShutter

Complete Command: **ZenService.Hardware.TriggerDigitalRLShutter**
Input:
Output: Boolean
Description: Returns the state of the reflected light shutter, where True = Open and False = Closed.

5.4.12 TriggerDigitalOutTLShutter

Complete Command: **ZenService.Hardware.TriggerDigitalOutTLShutter**
Input:
Output: Boolean
Description: Returns the state of the transmitted light shutter, where True = Open and False = Closed.

5.4.13 HasChanged

Complete Command: **ZenService.Environment.HasChanged(String observableId)**
Input:
Output: Boolean
Description: Returns weather a certain observable has changed. On simple occurrences it is possible to use the HasChanged("observableId") method instead of a custom lock

mechanism. Where available the HasChanged("observableId") method takes an observable name as parameter. HasChanged("observableId") returns true whenever the specified observable has changed since last script run; otherwise false. So it might not be suitable for situations when an observable changes very often and/or when really a singular execution is desired.

6 Script Commands - Available Actions

6.1 Actions - Experiment

Depending on the present hardware the shown options might vary, e.g. it is only possible to set the laser intensity, if there is a laser engine configured.

6.1.1 ContinueExperiment

Complete Command: **ZenService.Actions.ContinueExperiment()**
Input:
Output:
Description: This will continue the current experiment in case it was paused.

6.1.2 JumpToBlock

Complete Command: **ZenService.Actions.JumpToBlock(int newBlockIndex)**
Input: integer newBlockIndex
Output:
Description: This will jump to the specified experiment block inside a heterogeneous experiment.

6.1.3 JumpToContainer

Complete Command: **ZenService.Actions.JumpToContainer(string containerId)**
Input: string containerId
Output:
Description: This will jump to the specified container of the sample carrier.

6.1.4 JumpToNextBlock

Complete Command: **ZenService.Actions.JumpToNextBlock()**
Input:
Output:
Description: This will jump to next acquisition block defined inside the Experiment

Designer tool.

6.1.5 JumpToNextContainer

Complete Command: **ZenService.Actions.JumpToNextContainer()**
Input:
Output:
Description: This will jump to next container inside the currently used sample carrier.

6.1.6 JumpToNextRegion

Complete Command: **ZenService.Actions.JumpToNextRegion()**
Input:
Output:
Description: This will jump to next defined region.

6.1.7 JumpToPreviousBlock

Complete Command: **ZenService.Actions.JumpToPreviousBlock()**
Input:
Output:
Description: This will jump to the previous acquisition block inside the experiment designer tool.

6.1.8 PauseExperiment

Complete Command: **ZenService.Actions.PauseExperiment()**
Input:
Output:
Description: This will pause the current running experiment.

6.1.9 ReadLEDIntensity

Complete Command: **ZenService.Actions.ReadLEDIntensity(int trackindex, double wavelength)**
Input: integer trackindex, double wavelength [nm]
Output: double intensity
Description: This will read the intensity for the specified track and LED.

6.1.10 ReadTLHalogenLampIntensity

Complete Command: **ZenService.Actions.ReadTLHalogenLampIntensity(int trackindex)**
Input: integer trackindex
Output: double intensity
Description: This will read the intensity of the TL Halogen lamp for the specified track.

6.1.11 SetExposureTime (1)

Complete Command: **ZenService.Actions.SetExposureTime(int channelindex, double exposure)**
Input: integer channelindex, double exposure [ms]
Output:
Description: This will set the exposure time of the camera for the specified channel.

6.1.12 SetExposureTime (2)

Complete Command: **ZenService.Actions.SetExposureTime(int trackindex, int channelindex, double exposure)**
Input:integer trackindex, integer channelindex, double exposure [ms]
Output:
Description: This will set the exposure time of the camera for the specified track and channel.

6.1.13 SetLEDIntensity

Complete Command: **ZenService.Actions.SetLEDIntensity(int trackindex, double wavelength, double intensity)**
Input: integer trackindex, double wavelength [nm]
Output:
Description: This will set the intensity for the specified track and channel.

6.1.14 SetLEDIsEnabled

Complete Command: **ZenService.Actions.SetLEDIsEnabled(int trackindex, double wavelength, bool isEnabled)**
Input: integer trackindex, double wavelength [nm], boolean is Enabled
Output:
Description: This will enable the specified LED for the specified track. The predefined intensity value will be used.

6.1.15 SetMarkerString

Complete Command: **ZenService.Actions.SetMarkerString(string markerText)**
Input:string makerText
Output:
Description: This will insert a marker string at the current time point.

6.1.16 SetTimeSeriesInterval (1)

Complete Command: **ZenService.Actions.SetTimeSeriesInterval(double interval, TimeUnit unit)**
Input: double interval, TimeUnit unit (e.g. TimeUnit.ms)
Output:
Description: This will set the interval of a time series to the specified value.

6.1.17 SetTimeSeriesInterval (2)

Complete Command: **ZenService.Actions.SetTimeSeriesInterval(double interval)**

Input: double interval [ms]
Output:
Description: This will set the interval of a time series to the specified value in [ms].

6.1.18 SetTLHalogenLampIntensity

Complete Command: **ZenService.Actions.SetTLHalogenLampIntensity(int trackindex, doublem intensity)**
Input: integer trackindex, double intensity
Output:
Description: This will set the intensity of the TL Halogen lamp for the specified track.

6.1.19 StopExperiment

Complete Command: **ZenService.Actions.StopExperiment()**
Input:
Output:
Description: This will stop the current running experiment (similar to pressing the **Stop** button.

6.2 Actions - Experiment - LSM

6.2.1 LSM - ReadAnalogInTriggerState

Complete Command: **ZenServiceLSM.Actions.ReadAnalogInTriggerState(int port)**
Input: integer port
Output:
Description: Returns the state of specified AnalogInTriggerIn port.

6.2.2 LSM - ReadAnalogOutTriggerState

Complete Command: **ZenServiceLSM.Actions.ReadAnalogOutTriggerState(int port)**
Input: integer port
Output:

Description:Returns the state of specified AnalogInTriggerIn port.

6.2.3 LSM - ReadDigitalGain

Complete Command: **ZenServiceLSM.Actions.ReadDigitalGain(int trackIndex, int channelIndex)**
Input: integer trackindex, integer channelIndex
Output: double digitalGain
Description:Returns the value for the DigitalGain for the specified track/channel.

6.2.4 LSM - ReadLaserIntensity

Complete Command: **ZenServiceLSM.Actions.ReadLaserIntensity(int trackIndex, int lineWavelength)**
Input: integer trackIndex, integer lineWavelength [nm]
Output: double intensity
Description: Returns the laser intensity [%] for the specified laser wavelength.

6.2.5 LSM - ReadMasterGain

Complete Command: **ZenServiceLSM.Actions.ReadMasterGain(int trackIndex, int detectorIndex)**
Input: integer trackIndex, integer detectorIndex
Output: double masterGain
Description: Returns the value for the masterGain for the specified track/dectector combination.

6.2.6 LSM - ReadPinholeDiameter

Complete Command: **ZenServiceLSM.Actions.ReadPinholeDiameter(int trackIndex)**
Input: integer trackIndex
Output: double pinholeDiameter [micron]
Description: Returns the value for the pinhole in [micron] for the specified track.

6.2.7 LSM - ReadScanDirection

Complete Command: **ZenServiceLSM.Actions.ReadScanDirection()**
Input:
Output: ScanDirections direction
Description: Returns the current scan direction.

6.2.8 LSM - ReadScanSpeed

Complete Command: **ZenServiceLSM.Actions.ReadScanSpeed()**
Input:
Output: int speed
Description: Returns the current scan speed.

6.2.9 LSM - ReadTLLEDIntensity

Complete Command: **ZenServiceLSM.Actions.ReadTLLEDIntensity(int trackIndex)**
Input: integer trackIndex
Output: double intensity [%]
Description: Returns the intensity value [%] for the TL-LED.

6.2.10 LSM - SetAnalogOutTriggerState

Complete Command: **ZenServiceLSM.Actions.SetAnalogOutTriggerState(int port, double state)**
Input: integer port, double state
Output:
Description: Sets the voltage value for the AnalogOutTrigger port.

6.2.11 LSM - SetDigitalGain

Complete Command: **ZenServiceLSM.Actions.SetDigitalGain(int trackIndex, int channelIndex, double digitalGain)**
Input: integer trackIndex, integer channelIndex, double digitalGain
Output:

Description: Sets the value for the DigitalGain for the specified track/channel.

6.2.12 LSM - SetLaserEnabled

Complete Command: **ZenServiceLSM.Actions.SetLaserEnabled(int trackIndex, int lineWavelength, bool isEnabled)**
Input: integer trackIndex, integer lineWavelength, bool isEnabled
Output:
Description: Sets the state for the specified laser inside the specified track.

6.2.13 LSM - SetLaserIntensity

Complete Command: **ZenServiceLSM.Actions.SetLaserIntensity(int trackIndex, int lineWavelength, double intensity)**
Input: integer trackIndex, integer lineWavelength [nm], double intensity [%]
Output:
Description: Sets the laser intensity for the specified laser inside the specified track.

6.2.14 LSM - SetMasterGain

Complete Command: **ZenServiceLSM.Actions.SetMasterGain(int trackIndex, int detectorIndex, double masterGain)**
Input: integer trackIndex, integer detectorIndex, double masterGain
Output:
Description: Sets the value for the MasterGain for the specified track/detector.

6.2.15 LSM - SetPinholeDiameter

Complete Command: **ZenServiceLSM.Actions.SetPinholeDiameter(int trackIndex, double pinholeDiameter)**
Input: integer trackIndex, double pinholeDiameter [micron]
Output:
Description: Adjusts the pinhole for the specified track.

6.2.16 LSM - SetScanDirection

Complete Command: **ZenServiceLSM.Actions.SetScanDirection(ScanDirections direction)**
Input: ScanDirections direction
Output:
Description: Adjust the ScanDirection to the specified value.

6.2.17 LSM - SetScanSpeed

Complete Command: **ZenServiceLSM.Actions.SetScanSpeed(int speed)**
Input: integer speed
Output:
Description: Adjusts the ScanSpeed to the specified value.

6.2.18 LSM - SetTLLEDIntensity

Complete Command: **ZenServiceLSM.Actions.SetTLLEDIntensity(int trackIndex, double intensity)**
Input: integer trackIndex, double intensity [%]
Output:
Description: Adjusts the intensity of the TL-LED to the specified value.

6.3 Actions - Hardware

6.3.1 ExecuteHardwareSetting

Complete Command: **ZenService.HardwareActions.ExecuteHardwareSetting(string settingNameInExperimentSettingsPool)**
Input: string settingNameInExperimentSettingsPool
Output:
Description: This will load a specific experiment setting from the settings pool. **Please be aware of the fact, that the settings will be overwritten within the next experiment loop, if the settings are includes in the regular experiment settings.**

6.3.2 ExecuteHardwareSettingFromFile

Complete Command: **ZenService.HardwareActions.ExecuteHardwareSettingFromFile(string hardwareSettingFilePath)**
Input: integer string hardwareSettingFilePath
Output:
Description: This will load a specific experiment setting from a file directly. **Please be aware of the fact, that the settings will be overwritten within the next experiment loop, if the settings are includes in the regular experiment settings.**

6.3.3 PulseTriggerDigitalOut

Complete Command: **ZenService.HardwareActions.PulseTriggerDigitalOut(string portLabel, double duration)**
Input: string portLabel, double duration
Output:
Description: This will produce a TLL pulse with a specified duration at the selected trigger port.

6.3.4 PulseTriggerDigitalOutX (X=7-8)

Complete Command: **ZenService.HardwareActions.PulseTriggerDigitalOutX(double duration)**
Input: double duration
Output:
Description: This will produce a TLL pulse with a specified duration at trigger port X.

6.3.5 ReadFocusPosition

Complete Command: **ZenService.HardwareActions.ReadFocusPosition(double PositionInMicrometer)**
Input: double PositionInMicrometer
Output:
Description: Get the current value of the Z-Drive. This function was placed under the category **HardwareActions**, and not under **Observables**, since they trigger script runs automatically when changing.

6.3.6 ReadStagePositionX

Complete Command: **ZenService.HardwareActions.ReadFocusPosition(double PositionInMicrometer)**
Input: double PositionInMicrometer
Output:
Description: Get the current value of the stage X-axis. This function was placed under the category **HardwareActions**, and not under **Observables**, since they trigger script runs automatically when changing.

6.3.7 ReadStagePositionY

Complete Command: **ZenService.HardwareActions.ReadFocusPosition(double PositionInMicrometer)**
Input: double PositionInMicrometer
Output:
Description: Get the current value of the stage Y-axis. This function was placed under the category **HardwareActions**, and not under **Observables**, since they trigger script runs automatically when changing.

6.3.8 SetFocusPosition

Complete Command: **ZenService.HardwareActions.SetFocusPosition(double PositionInMicrometer)**
Input: double PositionInMicrometer
Output:
Description: This will set the focus drive to the specified position. Currently the piezo drive is not accessible from the Experiment Feedback script.

6.3.9 SetIncubationAirHeaterIsEnabled

Complete Command: **ZenService.HardwareActions.SetIncubationAirHeaterIsEnabled(bool IsEnabled)**
Input: bool IsEnabled
Output:
Description: This will enable the AirHeater of the incubation system.

6.3.10 SetIncubationAirHeaterTemperature

Complete Command: **ZenService.HardwareActions.SetIncubationAirHeaterTemperature(double temperature)**
Input: double temperature
Output:
Description: This will set the temperature of the AirHeater to the specified temperature.

6.3.11 SetIncubationChannelIsEnabled

Complete Command: **ZenService.HardwareActions.SetIncubationChannelIsEnabled(int channel, bool isEnabled)**
Input: integer channel, boolean isEnabled
Output: double intensity
Description: This will enable the specified channel.

6.3.12 SetIncubationChannelTemperature

Complete Command: **ZenService.HardwareActions.SetIncubationChannelTemperature(int channel, double temperature)**
Input: integer channel, double temperature
Output:
Description: This will set the temperature of the selected channel to the specified value.

6.3.13 SetIncubationCO2Concentration

Complete Command: **ZenService.HardwareActions.SetIncubationCO2Concentration(double concentration)**
Input: double concentration
Output:
Description: This will set the CO2 concentration to the specified value.

6.3.14 SetIncubationCO2IsEnabled

Complete Command: **ZenService.HardwareActions.SetIncubationCO2IsEnabled(bool isEnabled)**
Input: boolean isEnabled
Output:
Description: This will enable the CO2 incubation.

6.3.15 SetIncubationO2Concentration

Complete Command: **ZenService.HardwareActions.SetIncubationO2Concentration(double concentration)**
Input: double concentration
Output:
Description: This will set the O2 concentration to the specified value.

6.3.16 SetIncubationO2IsEnabled

Complete Command: **ZenService.HardwareActions.SetIncubationO2IsEnabled(bool isEnabled)**
Input: boolean isEnabled
Output:
Description: This will enable the O2 incubation.

6.3.17 SetStagePosition

Complete Command: **ZenService.HardwareActions.SetStagePosition(double positionXInMicrometer, double positionYInMicrometer)**
Input: double positionXInMicrometer, positionYInMicrometer
Output:
Description: This will set the stage XY position to the specified value.

6.3.18 SetStagePositionX

Complete Command: **ZenService.HardwareActions.SetStagePosition(double positionXInMicrometer)**
Input: double positionXInMicrometer
Output:
Description: This will set the stage X position to the specified value.

6.3.19 SetStagePositionY

Complete Command: **ZenService.HardwareActions.SetStagePosition(double positionYInMicrometer)**
Input: double positionYInMicrometer
Output:
Description: This will set the stage Y position to the specified value.

6.3.20 SetTriggerDigitalOut

Complete Command: **ZenService.HardwareActions.SetTriggerDigitalOut(string portLabel, bool isSet)**
Input: string portLabel, boolean isSet
Output:
Description: This will set the selected TriggerDigitalOut port to the specified value.

6.3.21 SetTriggerDigitalOut7

Complete Command: **ZenService.HardwareActions.SetTriggerDigitalOut7(bool isSet)**
Input: boolean isSet
Output:
Description: This will set TriggerDigitalOut7 port to the specified value.

6.3.22 SetTriggerDigitalOut8

Complete Command: **ZenService.HardwareActions.SetTriggerDigitalOut8(bool isSet)**
Input: boolean isSet
Output:
Description: This will set the selected TriggerDigitalOut8 port to the specified value.

6.3.23 SetTriggerDigitalOutRLShutter

Complete Command: **ZenService.HardwareActions.SetTriggerDigitalOutRLShutter(bool isSet)**
Input: boolean isSet
Output:
Description: This will set the TriggerDigitalOutRLShutter port to the specified value.

6.3.24 SetTriggerDigitalOutTLShutter

Complete Command: **ZenService.HardwareActions.SetTriggerDigitalOutTLShutter8(bool isSet)**
Input: boolean isSet
Output:
Description: This will set the selected TriggerDigitalOutTKShutter port to the specified value.

6.4 Actions - Extra

6.4.1 AppendLogLineString (1)

Complete Command: **ZenService.Xtra.AppendLogLineString(string logMessage)**
Input: string logMessage
Output: string (contains the file path of the data log file)
Description: This will create a text file (ExperimentName_Log.txt) inside the folder save folder/temp and the file path to the log file is returned. When called multiple times the text file will be appended by one row every time the function is used. If used for data logging number have to be converted to a string before one can write them into the

log file and column separators have to be specified as strings as well.

6.4.2 AppendLogLineString (2)

Complete Command: **ZenService.Xtra.AppendLogLineString(string logMessage, string logFileName)**
Input: string logMessage, string logFileName
Output: string (contains the file path of the data log file)
Description: This is similar to the function above but allows to specify the file path where the text file will be saved.

6.4.3 ExecuteExternalProgram (1)

Complete Command: **ZenService.Xtra.ExecuteExternalProgram(string exeFilePath)**
Input: string exeFilePath
Output:
Description: This will start an external application (*.exe, *.py, *.mp3, ...). It may be required to add the location of the application the environmental variables, usually PATH, in order to let windows know where to look. Or it is always possible to use the absolute file path.

6.4.4 ExecuteExternalProgram (2)

Complete Command: **ZenService.Xtra.ExecuteExternalProgram(string exeFilePath, string arguments)**
Input: string exeFilePath, string arguments
Output:
Description: This is similar to the function above but it allows to specify arguments which can be passed to the application. An example could look like this: ZenService.Xtra.ExecuteExternalPr
win64.exe", "-macro Open_CZI_OME_complete.ijm Experiment-123.czi")

6.4.5 ExecuteExternalProgramBlocking (1)

Complete Command: **ZenService.Xtra.ExecuteExternalProgram(string exeFilePath, string arguments, int timeoutInMs)**
Input: string exeFilePath, string arguments, int timeoutInMs

Output: integer exitCode
Description: This **advanced** method will run an external program and blocks the subsequent script. When the Experiment Feedback runs in **Determined Mode** it also blocks the experiment until the external application is closed. It requires a paramater called timeoutinMs, which specifies after which time period the script will continue independent from the external applications. As an output it returns a so-called ExitCode (integer), which is an integer produced by the external application when exiting. This ExitCode has to be supported by the external application. Standard Windows programs (like Notepad, etc.) just return 0. When using a special application or s self-written one it is possible to set special own ExitCode.

6.4.6 ExecuteExternalProgramBlocking (2)

Complete Command: **ZenService.Xtra.ExecuteExternalProgram(string exeFilePath, int timeoutInMs)**
Input: string exeFilePath, int timeoutInMs
Output: integer exitCode
Description: This is the same method as above but with additional arguments for the external application. The timeout parameter is still required.

6.4.7 PlaySound (1)

Complete Command: **ZenService.Xtra.PlaySound()**
Input:
Output:
Description: This will play the system sound.

6.4.8 PlaySound (2)

Complete Command: **ZenService.Xtra.PlaySound()**
Input:
Output:
Description: This will play the specified wave sound file.

6.4.9 RunLoopScript

Complete Command: **ZenService.Xtra.RunRepetionScript()**
Input:
Output:
Description: This command will trigger a run of the Repetition Script or LoopScript. Is only meant to be used when running in the Determined Mode.

7 Figure Index

List of Figures

8 Disclaimer

Carl Zeiss Microscopy GmbH's ZEN software allows to connect to a the third party software Python. Therefore 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.