Semester Project

Laboratory of Applied Photonics Devices

Mice ex-vivo retina projector design, implementation and acquisition synchronization with electrical electrode array

Student:
Victor Tiberghien (250380)

Supervisor:
Babak Rahmani
Professor:
Christophe Moser

Lausanne, Fall 2020

Ecole Polytechnique Fédérale de Lausanne

Contents

List of Figures

1 Introduction

The objective of this project is to build a setup in order to characterize the electrical responses of an ex-vivo retina excited by projecting different images on it. The setup requires having a perfect synchronisation between the projected images and the electrical impulses captured. Indeed, retina neurons response to light stimulation is very fast, in the order of a few milliseconds. Another important aspect that needs to be taken into consideration is that the experiment with the ex-vivo retina needs to be done at a relatively high speed. Consequently, the setup needs to be reliable, without any measurements loss, at high sampling rates. Moreover, as the experiment with the retina is conducted with large quantities of images (up to $10'000$), the system must be able to manage autonomously the flux of images to be projected by the intended devices.

The following report is divided into multiple sections. First of all, the optical setup as well as the optical components are briefly introduced. Then, the different electronic devices used for this experiment are presented: the stimulus generator to control the speed of the experiment by generating a trigger signal, the Digital Micromirror Device (DMD) to project images on the sample and the electrodes amplifier on which the sample is placed to capture the electrical responses. Afterwards, the scripts written to control and capture data from the devices are discussed. Finally, the multiple tests carried out to verify the proper functioning of the setup as well as their results are discussed.

2 Optical setup

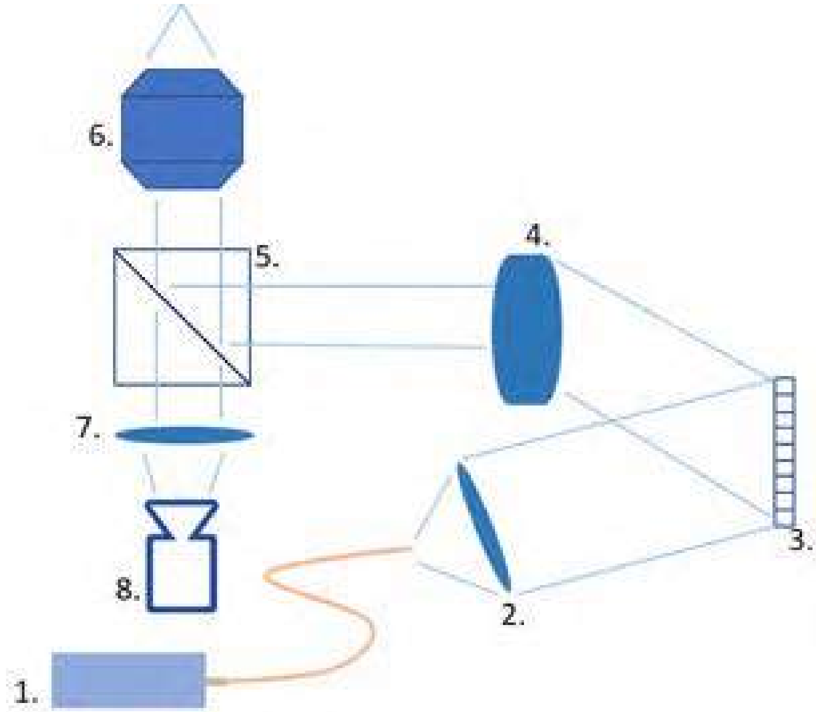

Figure 1: Complete optical setup

1. White LED from *THORLABS*, number MWWHF2. This is the light used for the experiment. It is coupled to the optical system using a multimode fiber. A white light implies a broad spectrum that needs to be corrected to avoid chromatic aberrations. Chromatic aberrations occur when different wavelengths focus at different distances.
2. Achromatic Doublet, f=40$[mm]$ from *THORLABS*, number AC254-040-A-ML. This lens collimates the light coming from the fiber on the surface of the DMD. Also, this specific lens allows to correct the chromatic aberrations by using two lenses to bring together the blue light and the red light.
3. Digital Micromirror Device (DMD), see section 3.3.
4. Tube lens, $f = 200[mm]$ *THORLABS*, number TTL200-A. Those lenses are designed to be used with infinity-corrected objectives.
5. Cube-Mounted Non-Polarizing Beamsplitter from *THORLABS*, number CCM1-BS013/M. Beamsplitters are used to split a light beam into two separate beams or can be also used in reverse to combine two beams into a single one. In this particular case, it is used to project images on the sample and at the same time, to observe it with a CCD camera.
6. Objective $2.5x/0.06$ infinity corrected from *Zeiss*. In an infinity corrected optical system, the image created by the objective is set to infinity. A specific tube lens needs to be placed after the objective in order to produce an intermediate image.
7. Achromatic Doublet $f = 45[mm]$ from *THORLABS*, number AC254-045-A-ML. Lens to couple the camera to the optical system.
8. CCD Camera, 1024x768 resolution from *THORLABS*, number DCU223M. Camera used to observe the sample.

3 Devices

3.1 Stimulus Generator

In order to generate simultaneous stimulus to synchronize the DMD and the electrodes amplifier, one can use a stimulus generator. The model used for the setup is the STG4004 from *Multichannel Systems*. The device is able to generate stimulus with all kind of shapes and amplitudes. It can generate analog or transistor-transistor logic (TTL) pulses. In the context of this experiment, TTL signals were used such that they can trigger the DMD and the electrodes amplifier. Those pulses are made of two states: logic state HIGH which corresponds to a $3.3[V]$ output signal and a logic state LOW which corresponds to a $0[V]$ output signal. The main parameters that can be configured are the ON_{time}, the OFF_{time} as well as the number of pulses. Those can be visualized in Figure 3.

The STG4004 is connected to the computer via USB2.0 and connected to the DMD and electrodes amplifier via two *Sync Out* BNC connectors located at the back of the device. This particular model can generate stimulus up to $25[kHz]$.

Figure 2: Stimulus generator device

3.1.1 Programming

The stimulus generator operates in download mode, meaning that the stimulus are first created on the computer and then transferred to the device. Once the transfer is over, the stimulus can be generated either by pressing the play button on the device or by sending the start command via software with a computer.

The interface between the computer and the STG4004 is achieved via a Python class located in the *MCS_ devices.py* file. The class is made after the inheritance of the class loaded with the *Dynamic Link Library*(DLL) file.
When an instance of the class is created, the constructor is automatically called. The latter will first look for connected devices and make sure that the STG is connected.
Once connected, the stimulus generator is first cleared of previous data. Then, each output is individually configured. All 4 SYNC OUT outputs are activated, while the analog outputs are deactivated. Then, the same stimulus is created for the all SYNC OUT outputs according to the desired parameters (T_{low}, T_{High}, number of repetition). Finally the stimulus is transferred to the device.
A separated command is then sent in order to start the stimulus.

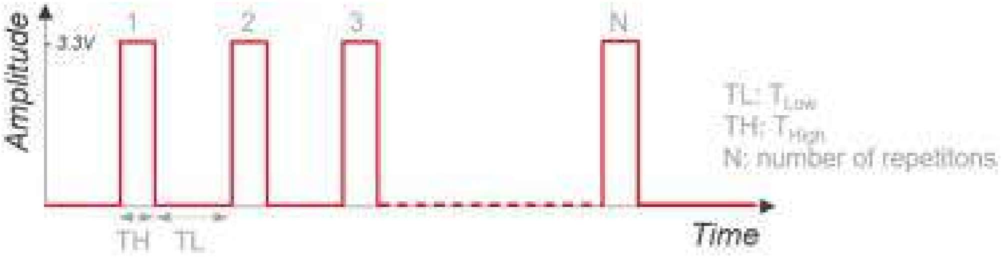

Figure 3: Trigger signal

3.2 Electrodes amplifier

The electrodes recording used for the setup is the USB-MEA256-System from *Multichannel Systems*. It captures and amplifies the signal coming from the retina thanks to a Microelectrodes array (MEA) composed of 252 electrodes and 4 reference electrodes as it can be seen in Figure 4. The raw signal coming from the electodes is then digitalized in real time by the integrated analog / digital converter. The latter being able to reach a sampling rate of up to $40[kHz]$ per channel. The voltage range of those electrodes is $\pm 3.7[mV]$ with 16 bits resolution which corresponds to a resolution of $113[nV]$. Additionally, the device is equipped with digital inputs than can be used to receive the triggering signal coming from the stimulus generator in order to synchronise the recording with the stimulation of the sample.

Figure 4: Electrodes amplifier

3.2.1 Programming

The interface between the computer and the electrodes amplifier is achieved through the python class MCS_MEA located in the $MCS\_devices.py$ file. At the creation of an instance, the constructor firstly check if the device is connected to the computer and if it is the case, it connects to it.

Afterwards, the device is configured with the desired parameters. First of all, the sampling frequency of the electrodes is set, it can go from $1[Hz]$ up to $40[kHz]$. Then, the number of channels to activate is chosen. Here, since we want all 252 electrodes plus 4 analog inputs, all the 256 channels are activated. Furthermore, the digital input used for the triggering signal is also activated. We end up with a total of 257 values for one sample. The data format of the measurements is also configured. For the current setup, it is set to unsigned integers of 16 bits. Then, the buffer needs to be set up. It contains all the samples that have been recorded but not yet sent to the computer. It's a kind of waiting queue based on the principle of First In First Out (FIFO). For this experiment, the queue size is set to hold up to 10^6 samples which is close to the maximum available memory.

Each time a new packet of samples is recorded and sent to the computer, a thread function is automatically called within the script. A thread function is a function that can be run simultaneously to the main script when a certain event occurs (here, reception of a new packet). The packet size is configured to contain a number of samples of half the sampling frequency, $\#samples = sampling\_frequency/2$. It means that during the recording process, the thread function is called every 500[ms]. This value has been found after several tries and failures. The callback function receives a single table, data[ ], containing the measurements coming from the device and saves it in the *.csv* file. The structure of data[ ], can be seen in Table 1.

Samples	Analog 0	Analog 1	...	Analog 255	Digital IN
Sample 1	data[0]	data[1]	...	data[255]	data[256]
Sample 2	data[257]	data[258]	...	data[512]	data[513]
...	...	...	...	...	...
Sample 10	data[2570]	data[2571]	...	data[2825]	data[2826]

Table 1: Structure of an array sent by the electrodes amplifier

3.3 Digital Micromirror Device

A digital micromirror device is an array of binary micromirrors that can be actuated individually using an electrocapacitive actuation. It can also generate grayscale images by toggling on and off the mirrors at high frequencies determined by pulse-width modulation. The DMD chip used for the experiment is the model *DLP7000BFLP* manufactured by *Texas Instrument*. It contains 1024 by 768 mirrors that represent the pixels. Each mirror tilts with angles of ±12° relative to the flat surface. This particular model is made to be used with visible light, that is wavelengths in the range of 400[nm] to 700[nm].

The chip itself comes with a controller that provides an interface with a computer. The model used is the *V4100 board* by *Vialux*. It allows, among other things, to store images to be displayed as well as tuning different parameters such as the *picture time* or the hardware trigger. This particular model has a 16[$Gbit$] on board DDR2 RAM intended to store images. The controller enables also the possibility to display grayscale images of 255 different values (8[$bits$]). In our configuration, the DMD is controlled via the trigger signal sent by the stimulus generator. In order to achieve this, one can use the pins *TRIGGER_IN* and *GND* located on the Multi-Purpose I/O Molex connector of the controller.

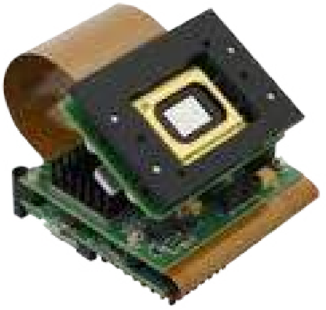

Figure 5: Digital Micromirror Device and its controller

3.3.1 Programming

The interface between the DMD controller and the computer is achieved through the Dynamic Link Library *alpV42.dll*. This file contains all the necessary functions required to configure and operate the DMD. In order to access those functions, the python class *DMD* located in the file *communication.py* is used. This class is an adaptation and modification of a version made by Matthias Müller-Schrader in 2015 at ETHZ.

At the creation of an instance of the class, the constructor looks for the DMD serial number to ensure that the communication is well established. Then, the DMD needs to be configured in the *slave mode*. In this configuration, the DMD projection loop waits for a trigger event before the next picture of the sequence is displayed. It is also important to define the trigger event. In our case, it was chosen to work with the rising edge of the triggering signal, that is when the signal goes from 0 to 1 ($0[V]$ to $3.3[V]$). Figure 6 displays how the signal *TriggerIn* triggers the *PictureTime*. The *PictureTime* is the time during which a picture is displayed on the DMD. This is an important parameter that needs to be modified by the user in the main *Python* file. The minimum possible *PictureTime* depends on the format of the images. For binary images it is $44[\mu s]$ while for $8-bits$ grayscale images it is $3.4[ms]$. It turns out to be the limiting factor for the experiment images frequency.

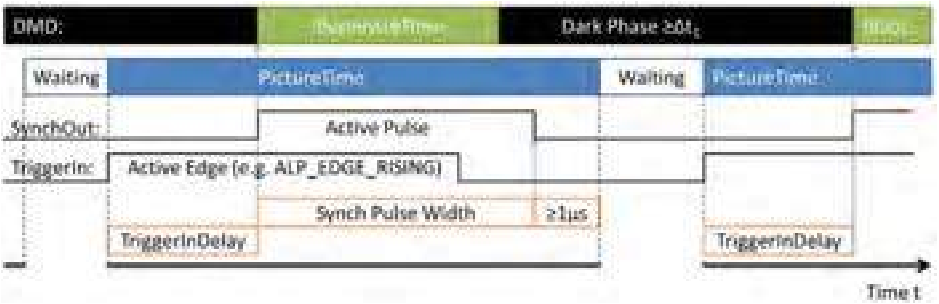

Figure 6: DMD timing in *slave* mode

On the DMD, the pictures are organized in sequences. Once a sequence of pictures is created (see Section 4.1), it is uploaded via USB2.0 on the DMD on board RAM. Finally, when the upload is finished, the DMD can be put in a standby mode, meaning that it waits for the trigger events so that it can display the picture one after the other. While waiting for trigger events, the script is paused until the end of the sequence. Even if the script is paused, if a new packet of samples is sent by the electrodes amplifier, the computer will still be able to process it because the data is treated by a thread function that can run in parallel.

3.4 Montage

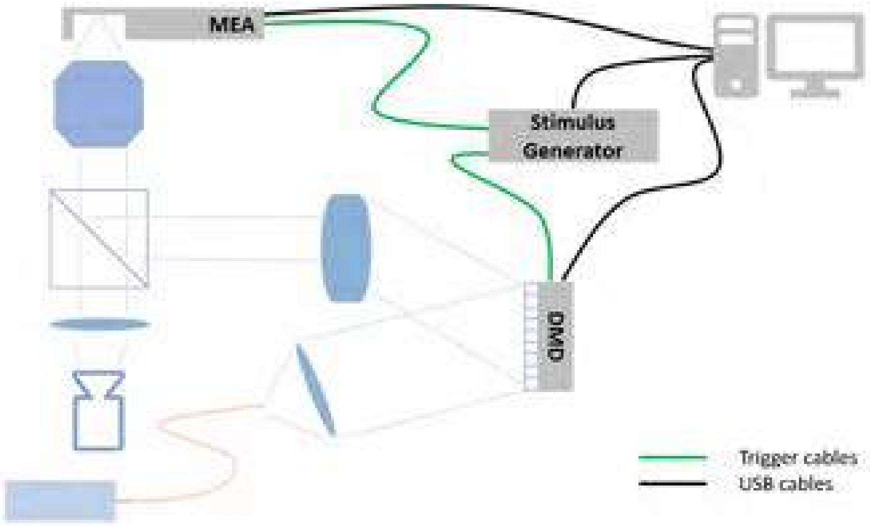

Figure 7: Overall montage

4 Overall script

4.1 Images preparation

For this experiment, grayscale images are needed. Consequently, each pixel of an image has an $8-bits$ value ranging from 0 to 255. Before uploading the images to the DMD, these need to be compiled in order to be understood by the DMD. This is achieved by the function *import_ and_ compile_ images().* The latter looks for a certain type of image extension in a certain folder, for instance *.png*. Then, knowing the total number of images, it divides them into several packets such that the number of images in a single package doesn't exceed the maximum number of images that the on-board DMD memory can hold. This maximum capacity can be approximated knowing the on-board RAM of the module:

$$\#\text{MAX\_IMAGES} = \frac{\text{On-board RAM[bits]}}{\#\text{bits per pixel} \cdot \#\text{pixels in an image}} = \frac{16 \cdot 10^9}{8 \cdot (1024 \cdot 768)} = 2543 \text{ images} \tag{1}$$

Finally, each packet containing a list of 2D arrays representing the images needs to be resized. This is achieved using the function *compilePicture* located in the file *communication.py*. The latter transforms each 2D array into a single 1D array and arranges them one after the other. This final resulting 1D array contains the individual $8-bits$ pixels of each image. Its length is equal to: $1024 \cdot 768 \cdot \#$images in the packet. Figure 8 shows this last step of the compilation.

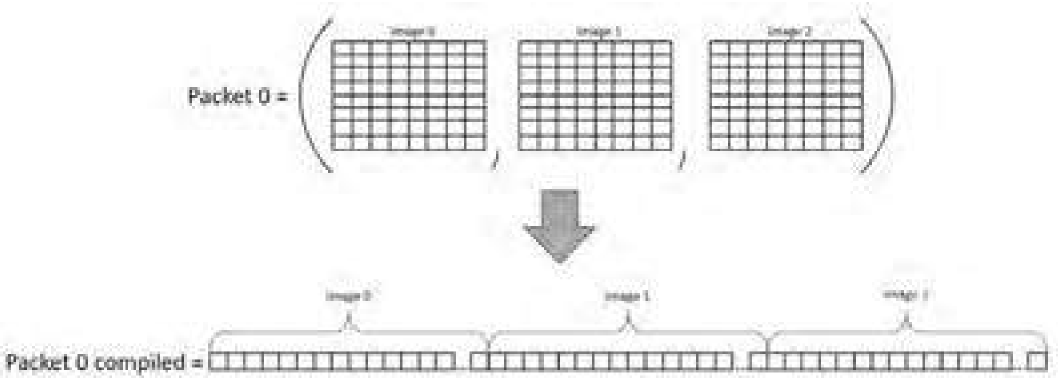

Figure 8: Images compilation

4.2 Images projection & data acquisition

Once all the images are compiled, the data file *.csv* that will contain all the measurements is created and prepared to receive incoming data. A timestamp is added to the file in order to keep a trace of when the experiment was performed.

The script then enters a for loop according to the number of packets required to project all the images. Next, all the devices need to be configured according to the experiment parameters: the electrodes sampling frequency, the DMD picture time and the trigger frequency. The number of repetitions of the trigger signal is given by the number of images in the current packet. Then, all the images contained in the current packet are uploaded as a sequence to the DMD and the latter is put on standby, waiting for the trigger signal. The data acquisition is started and finally, the *start* command is sent to the stimulus generator. The script starts receiving data from the electrodes amplifier and, at the same time, waits for the end of the sequence which is signaled by the DMD when all the pictures in the sequence have been projected. Once finished, the data acquisition is stopped and the DMD memory is freed. Finally, at the end of the loop, the script is paused again so that all the remaining data on the memory of the electrode amplifier can be properly sent and stored on the computer before acquiring data of the next packet.

Once every packet is displayed and all the corresponding data is stored in the data file, all the devices are properly disconnected and a *.txt* file containing the experiment paramerters is created with the same timestamp as the data file.

4.3 Data processing

The data containing the measurement needs some processing before being able to plot it. At first, the data was processed in real time, in the callback function, as it was coming from the electrodes amplifier. Each time a new set of samples arrived, it would convert it and reshape the array such that each line correspond to an individual sample. The problem with that method is that when the sampling frequency of the measurement exceeds a certain threshold situated around $1[kHz]$, the amount of samples into a single set is so large that the computer doesn't have the time to fully process this set before the next one arrives. This resulted in incomplete data *.csv* file with incomplete or missing samples.

The solution found for this problem is that instead of processing that data in real time as it comes, the data is now saved directly in the file without any kind of processing. The arrays *data[]* containing the sets of samples are saved line by line in the file. This allows to reduce the computing time and thus to save the incoming data at high frequencies without any loss. However, at the end of the experiment, the data still needs to be processed. This is achieved using a separated script *process_ data.py*.

The script performs two different steps simultaneously. The first step is to reshape the data file such that each line corresponds to a different sample and not a whole set as it is the case when the data is coming from the electrode amplifiers. The second step is to convert the raw numerical values of the measurements such that they can be expressed in $Volts$. The data format used to encode the measurements is *uint16*, which corresponds to 16 bits unsigned integers that go from 0 to $2^{16} = 65536$. A sample consists of three different kinds of measurement: there are 252 analog values coming from the MEA with a range of $\pm 3.7mV$, 4 analog values from the additional analog inputs with a range of $\pm 4.096V$ and 1 digital value coming from the digital IN. Only the analog values need to be converted. The following formulas illustrate how the conversion is performed:

$$MEA\_value[mV] = \frac{raw\_value}{65535} \cdot (2 \cdot 3.7) - 3.7 \tag{2}$$

$$Analog\_value[V] = \frac{raw\_value}{65535} \cdot (2 \cdot 4.096) - 4.096 \tag{3}$$

The result of this processing is saved in a new *.csv* file with the suffix *_processed* as well as a new timestamp.

4.4 Data visualisation

In order to visualize and plot the measured data correctly, the *Matlab* script *Data_visualization.m* is used. The latter reads all the measurement values from the *.csv* file. In order to plot the measurements versus time, the sampling frequency of the measurements is extracted from the *.txt* file containing the experiment parameters. In order to plot in $[ms]$, one can simply apply the following formula in order to compute the time step between each sample:

$$time\_step[ms] = \frac{1}{sampling\_frequency[Hz]} * 1000 \tag{4}$$

Figure 9 shows the typical data obtained after a test experiment with the following parameters:

- Electrodes sampling frequency: $10[kHz]$
- DMD picture time: $10[ms]$
- Trigger high time: $1[ms]$
- Trigger low time: $16[ms]$

- Number of images: 6

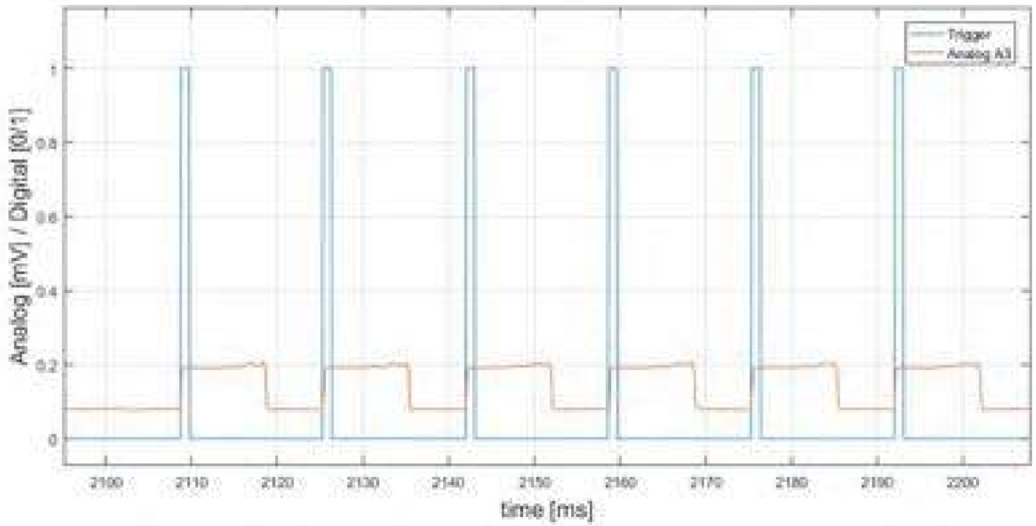

Figure 9: Graph obtained after the projection of 6 images on a photodiode

5 Tests & results

The approach used in order to test the complete setup composed of the stimulus generator, the electrodes amplifier and the DMD was to divide it into multiple sub-problems. Initially, each device was handled individually with its own dedicated *Python* script.

The first component addressed was the stimulus generator. The communication between the computer and the stimulus generator as well as the correct configuration by the python script was tested by connecting an oscilloscope directly at one of the *Sync out* output of the stimulus generator. By sending logic signals with variable periods and visualizing the results on the oscilloscope, one could validate the correct performance of the device. This test allowed to solve several problems regarding the allocation of the outputs.

Afterwards, the electrode amplifier was tested. Firstly, the good communication with the computer was assessed by configuring the device with random parameters and then retrieving these parameters using another function. Next, the validity of the device electrical measurements was assessed. By connecting one of the outputs of the stimulus generator to an analog input of the electrode amplifier, one could see if the signal generated corresponds to the signal received and processed by the electrodes amplifier. The result of such an experiment can be seen in Figure 10.

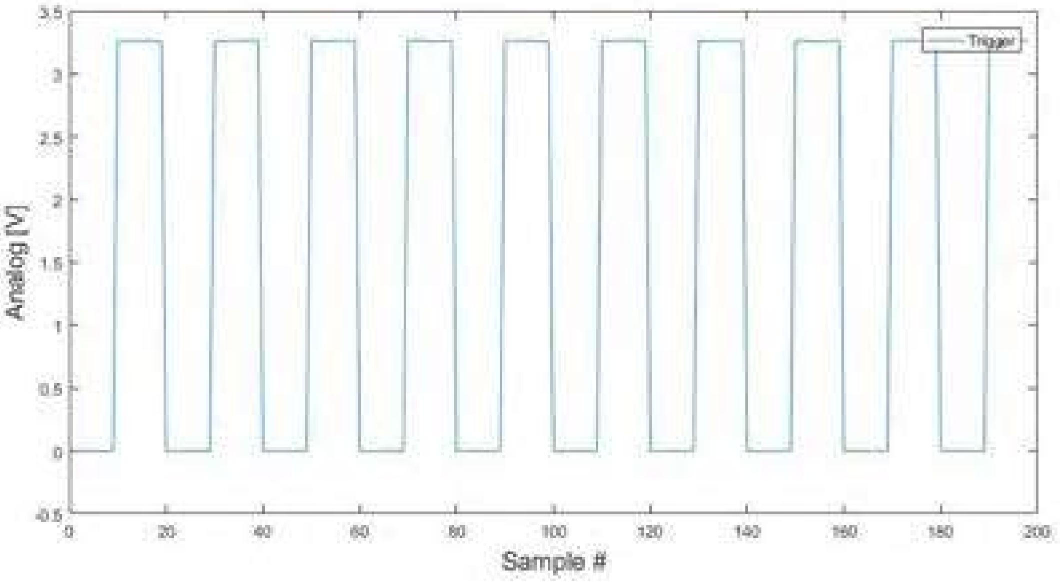

Figure 10: Experiment of sampling the analog value of a logical signal

Looking at Figure 10, one can see that the signal acquired by the electrodes amplifier corresponds to the signal generated by the stimulus generator. Indeed, the signal oscillates between $0[V]$ and $3.3[V]$ which corresponds to the logic state LOW and HIGH. Furthermore, tests at higher sampling frequencies, around $10[kHz]$ were conducted in order to determine the limits of the device.

Finally, the last component to be tested was the DMD. Like the other devices, the DMD was initially tested individually. Once the communication has been properly established, the first aspect carried out concerned the projection of images. Using a secondary script, image samples were created on the computer to assess the capabilities of the projection. Figure 11 displays typical images that were uploaded and displayed on the DMD.

Figure 11: Images used to assess the capability of the DMD to project grayscale patterns

The correctness of the image projection was evaluated using another setup from the laboratory that has a camera pointed in the direction of the DMD that allows to visualize with a computer the projected image.

Another aspect investigated was whether the total storage capacity of the DMD corresponds to the theoretical one computed in section 4.1. Using a function from the DMD library, it turned out that the maximum $8-bits$ images that can be stored is 2730, slightly bigger than the computed one. Furthermore, the data rate from the computer to the DMD can be computed and tested. Since the connection uses a *High Speed* USB 2.0 cable, the maximum data rate is $480[Mbit/s]$. The time it would take to upload 2730 images to the DMD can be computed as follow:

$$Time[s] = \frac{\#images \cdot \#bits\_per\_images}{Data\_rate} = \frac{2730 \cdot 1024 \cdot 768 \cdot 8}{480 \cdot 10^6} = 35.8[s] \tag{5}$$

It turned out that the real uploading time was situated more around $15[s]$. The latter was measured using a simple timing function in the *Python* script. Those differences between theoretical values and real values can be explained by the fact that the images undergo some kind of compression before being uploaded to the DMD.

Once all the devices operated properly separately, they were connected together and their scripts were combined. An important aspect was to assess the reaction time of the DMD triggered by the stimulus generator. This was achieved using a GaP detector pointed towards the direction of the DMD and connected to an analog input of the electrode amplifier as seen in Figure 12. The detector used for this operation is the model PDA25K2 from *Thorlabs*. When triggered by the stimulus generator, the DMD displays an image which illuminates the detector. The latter responds with an increase in voltage.

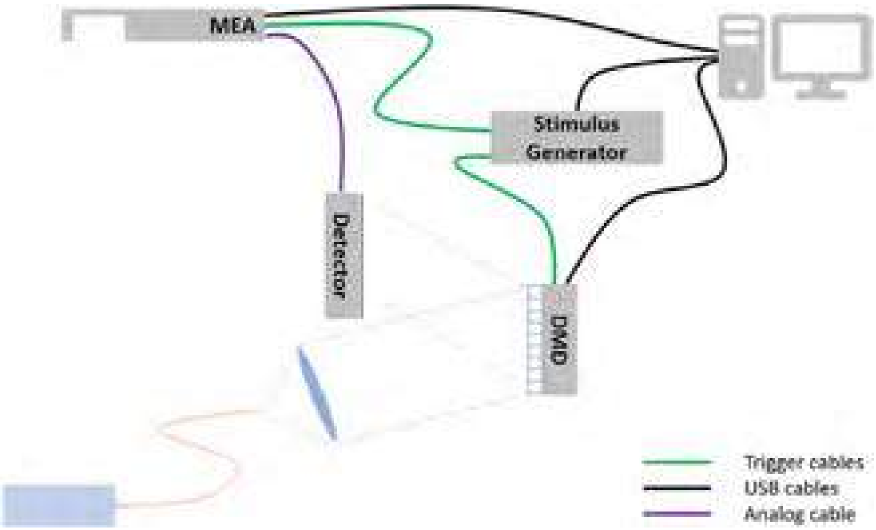

Figure 12: Montage with the photodetector connected to an analog input of the electrodes amplifier

This experiment allowed also to assess the temporal precision of setup. The parameters set and uploaded to the devices are the following:

- Electrode amplifier sampling frequency: $10[kHz]$
- DMD picture time: $10000[\mu s]$
- Stimulus generator frequency: $60[Hz]$
- Stimulus generator T_{HIGH}: $1000[\mu s]$

The result of such an experiment can be seen in Figure 13 and in Figure 14.

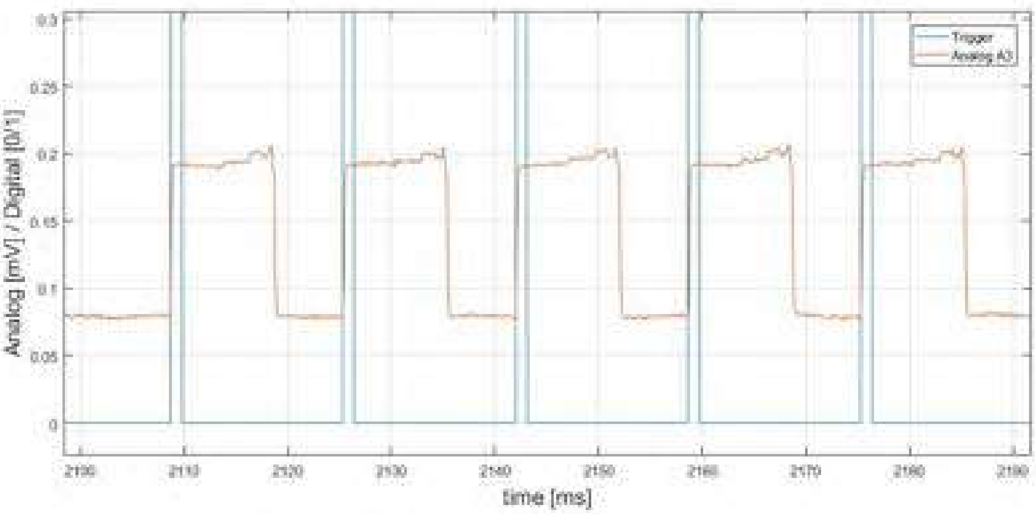

Figure 13: Values obtained after the projection of images on the photodetector

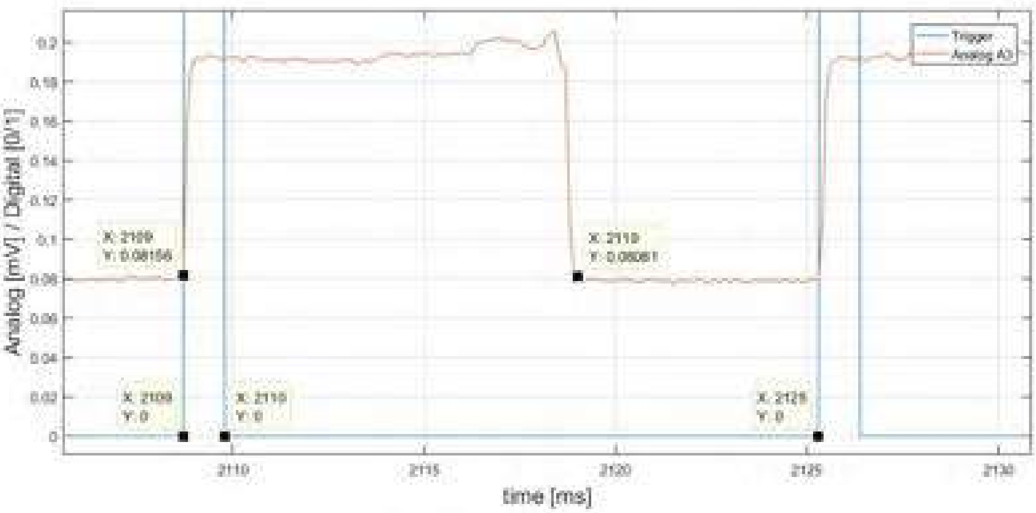

Figure 14: Values obtained after the projection of images on the photodetector with datatips measurements

Looking at the X_{values} of the *datatips* in Figure 14, one can observe that it corresponds to the previously specified parameters:

- DMD picture time: $2119[ms] - 2109[ms] = 10[ms]$
- Stimulus generator frequency: $1/(2125[ms] - 2109[ms]) \approx 60[Hz]$
- Stimulus generator T_{HIGH}: $2110[ms] - 2109[ms] = 1[ms]$

The average time delay between the photodiode response and the signal generator pulses is also an important factor for the precise execution of this experiment. In order to evaluate this value, the acquired data from the previous experiment is used with some added post-processing. Figure 15 displays multiple rising edges of the trigger signal followed by the rise of the photodiode voltage. For clarity purpose, the photodiode signal has been offset vertically to correspond to the *low* value of the trigger signal. Since the experiment is performed at a sampling frequency of $10[kHz]$, each data point is separated by a time step of $1/10000$ $[s]$. Looking at Figure 15, one can clearly see that the rising voltage of the photodiode (orange curve) occurs exactly at the same data point that the rising edge of the stimulus generator (blue curve). Therefore, it is safe to assume that the time delay between both devices is smaller than $100[\mu s]$, which means that all three devices (the stimulus generator, the electrodes amplifier and the DMD) are very well synchronized. To measure the time delay even more accurately, the sampling frequency should be further increased in order to have time steps smaller than $100[\mu s]$.

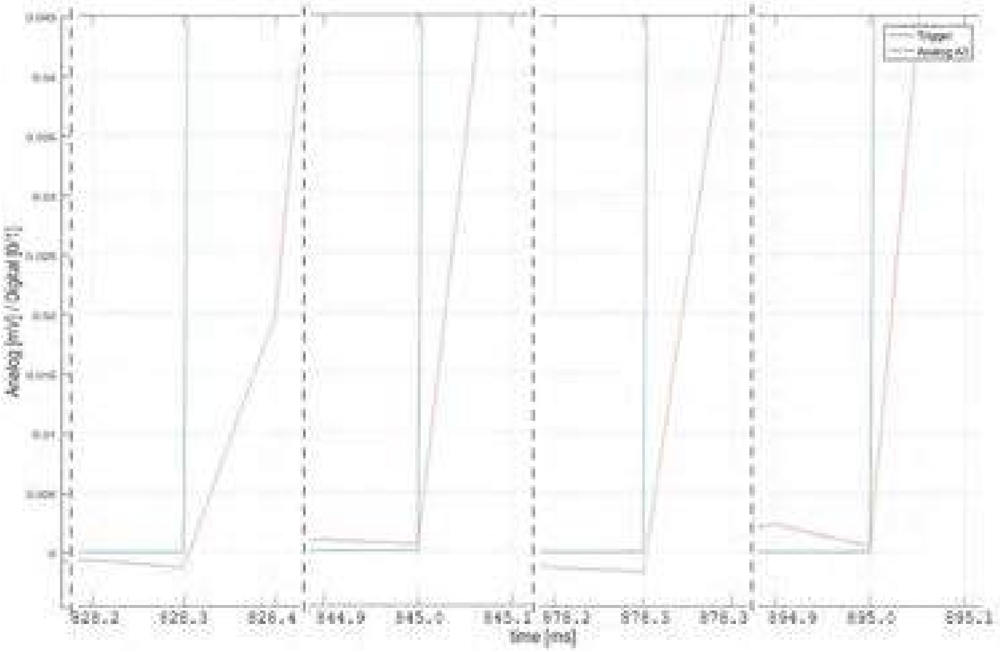

Figure 15: Time delay measurements

Another important point that needs to be investigated is the setup ability to display a large number of images successively. As it was seen in Section 4.1, the maximum number of images for 1 packet is limited to 2730. Thus, an other experiment needs to be conducted to determine the behaviour of the setup when projecting and sampling data with more than one packet of images. For this experiment the same parameters as the previous one were used and the number of images was 4600. The result of the processed data obtained with the samples of this experiment can be seen in Figure 16.

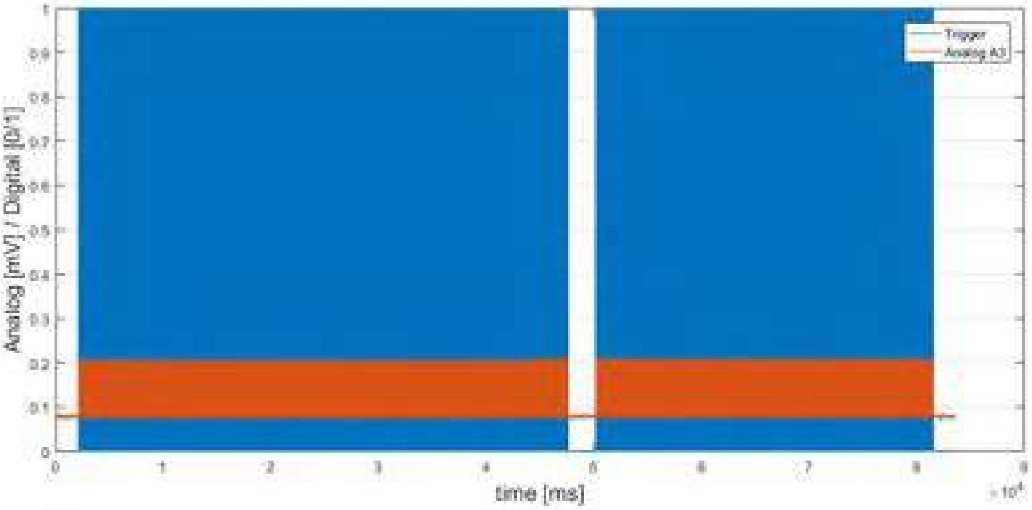

Figure 16: Graph obtained after the projection on the DMD of two packets of images; The first one with 2730 images and the second one with 1880 images

Looking at Figure 16, one can see that the total number of images was divided into two packets. Indeed, as said in section 4.1, if the number of images exceeds the available memory on the DMD, the experiment will be performed in separate steps. Here, the first packet contains the maximum number of images, 2730 and the second packet contains the rest, $4600 - 2700 = 1900$.

The last test performed on the setup was its ability to project a lot of images ($> 5000'$) and, at the same time, having a high sampling frequency on the electrode amplifier. It turned out that the bottle neck of this experiment is the data transfer from the electrode amplifier to the computer. Indeed, a single measurement contains $256 \cdot 16[bit] = 4096[bits]$. Since a new measurement is available each $1/sampling\_frequency$ $[s]$, it represents a data rate of $4096 \cdot sampling\_frequency$ $[bit/s]$. When the sampling frequency exceed $5[kHz]$, new measurements are generated faster than the computer can receive and store them. This is the reason why it was decided to carry out the data processing in a second step, in order to reduce the resources and time of live data saving as much as possible. Nevertheless, even with this technique, at very high sampling frequencies, new data was generated faster than the computer can receive and store it. However, the electrode amplifier is equipped with an internal memory which allows to store the samples before sending them to the computer. Therefore, when the acquisition of new data is stopped, the device still needs some time to send the rest of the data to the computer. When dealing with only one packet of images (less than 2730), this causes no problem. However, when working with many packets, the internal memory of the electrode amplifier tends to overflow, leading in a loss of data. The solution found to solve this problem is that between the projection of each packet, the data acquisition as well as the script itself are paused such that the rest of the data stored on the electrode amplifier memory can be received. The holding time depends on the sampling frequency. The higher it is, the longer the time between two consecutive packets. This solution therefore allows the projection of multiple packets at very high sampling frequencies, higher than $10[kHz]$.

6 Conclusion

The objective of this project was to develop a solution to connect different devices together in order to perform excitation by images projection and electrical acquisition of retina samples. The most important requirement was that the solution must ensure a good synchronization between the images projected by the DMD and the signal captured by the electrodes amplifier. Moreover, it needed to be robust to the projection of large quantities of images at high switching rates.

The produced *Python* scripts fulfill these objectives. Indeed, the setup provides a time delay of less than $100[\mu s]$ between the excitation and the response signal. Regarding the image switching rate, we are limited by the DMD actuation system itself. For $8-bits$ grayscale images, the maximum rate is $290[Hz]$. Another source of limitation occurs when a high sampling frequency is chosen on the electrodes amplifier. Indeed, with sampling frequencies higher than $5[kHz]$, the system is forced to pause between the projection of two packets of images while the remaining data sampled with the previous packet is sent to the computer. The higher the sampling frequency, the longer the time required between each packet.

Regarding the potential improvements of these scripts, if a solution is found to improve the data rate between the electrodes amplifier and the computer, the experiment could run at the maximum switching rate of the DMD, $290[Hz]$, without any interruption, even to refill new images. Indeed, the DMD internal memory could be split into multiple sequences such that while a sequence is being used to project images, the other one is refilled with new images from the computer. This parallel processing would make it possible to project more than $10'000$ images without any interruption to refill the DMD memory.

7 Appendices

In the following pages, scripts written for this project can be found in this order:

- The *Matlab* script allowing to visualize the data file *.csv*
- The main *Python* script managing the devices and controlling the experiment parameters
- The *Python* script allowing to process the raw data to produce data usable by the *Matlab* script.
- The *Python* library containing the classes to control the stimulus generator and the electrodes amplifier
- The *Python* library containing the class to control the DMD

```matlab
clc
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%% Needs to be modified with the correct file name%%%%%%%%
data_file = 'Experiment_02-Dec-2020_10H-37M_processed_02-Dec-2020_10H-38M.csv';
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% extracting the data
data = readtable(data_file);
size_of_data =size(data)

% exatraction of the sampling frequency from the corresponfing .txt file
newStr = split(data_file,'_processed');
txt_file = newStr(1)+'.txt';
fileID = fopen(txt_file,'r');
first_line = split(fscanf(fileID,'%s',2),':');
sampling_freq = str2double(first_line(2));
time_step_in_ms=(1/sampling_freq)*1000;


trigger = data.Var257; % the last column of data is the digital in (the trigger)

% deterimation of the time axis
time = transpose(0:time_step_in_ms:(length(trigger)-1)*time_step_in_ms);

plot(time, trigger)
grid on
hold on
plot(time, data.Var255)
xlabel('time [ms]','FontSize',15)
ylabel('Analog [mV] / Digital [0/1]','FontSize',15)
legend({'Trigger','Analog A3'})
```

```
"""
Laboratory of Applied Photonics Devices - EPFL
Fall 2020
Victor Tiberghien

Main script that works in those main parts:
 - Makes sure that all three devices are connected (Stimulus generator, electrodes amplifier and DMD)
 - First, it extracts the images from a directory and compiles them
 - Configure the DMD with the wanted parameters (slave mode, picture time,...)
 - Configure the electrodes amplifier (MEA256) by setting up the trigger signal and the outputs
 - Configure the stimulus generator (STG4004) with defined pulses duration and number of repetition
 - Start the acquisition of the MEA256. 256 electrodes (analog) values + 1 trigger (digital) value
 - Start the sequence of trigger pulses by the STG4004
 - Finally the program saves the raw data in a csv file "electrode_..._raw.csv"
 - The data needs then to be processed with the script process_data.py

"""
import communication
import MCS_devices
import time
from PIL import Image
import numpy as np
import glob
import math
import sys

MAX_IMAGES_MEMORY = 2730


def import_bin_file():
    print("Importing images...")
    print("This might take a while...")
    images = np.fromfile("train_labelsF.bin",  dtype='B')
    nbr_of_images=  int(len(images)/(1024*768))
    print(nbr_of_images, " images detected")
    number_of_packages = math.ceil(nbr_of_images / MAX_IMAGES_MEMORY)
    print("Number of packages needed: ", number_of_packages)
    list_of_seq = [None] * number_of_packages
    name_of_seq = [None] * number_of_packages
    for k in range(number_of_packages):
        name_of_seq[k] = 'seq' + str(k)
        k_seq = images[(k *(1024*768)* MAX_IMAGES_MEMORY):((k + 1)*(1024*768) * MAX_IMAGES_MEMORY)]
        print("Number of images in this packet is ", int(len(k_seq)/(1024*768)))
        list_of_seq[k] = k_seq
        print("Package", (k + 1), "over", number_of_packages, "completed")
        image_print = k_seq[(1024*768)*3:4*(1024*768)]
        image_print.resize(768,1024)
        imgagee = Image.fromarray(image_print)
        imgagee.save('table.png')

    return list_of_seq, name_of_seq, nbr_of_images


def import_and_compile_images():
    filelist = glob.glob(parameters["images_directory"])
    print("Importing images...")
    print("This might take a while...")
    images = np.array([np.array(Image.open(fname)) for fname in filelist], dtype=np.uint8)
    nbr_of_images = int(len(images))
    print(nbr_of_images, " images detected")
    number_of_packages = math.ceil(len(images) / MAX_IMAGES_MEMORY)
    print("Number of packages needed: ", number_of_packages)
    list_of_seq = [None] * number_of_packages
    name_of_seq = [None] * number_of_packages
    for k in range(number_of_packages):
        name_of_seq[k] = 'seq' + str(k)
        k_seq = images[(k * MAX_IMAGES_MEMORY):((k + 1) * MAX_IMAGES_MEMORY)]
        print("Number of images in this packet is ", len(k_seq))
        list_of_seq[k] = communication.compilePicture(k_seq, int(len(k_seq)))
        print("Package", (k + 1), "over", number_of_packages, "completed")

```

```python
    return list_of_seq, name_of_seq, nbr_of_images


def save_experiment_parameters(parameters_out):
    with open(parameters_out["file_name"], 'w') as f:
        print("electrodes_sampling_freq[Hz]: ", parameters_out["electrodes_sampling_freq[Hz]"], file=f)
        print("trigger_freq[Hz]: ", parameters_out["trigger_freq[Hz]"], file=f)
        print("trigger_Thigh[us]: ", parameters_out["trigger_Thigh[us]"], file=f)
        print("trigger_Tlow[us]: ", parameters_out["trigger_Tlow[us]"], file=f)
        print("images_directory: ", parameters_out["images_directory"], file=f)
        print("DMD_Picture_time[us]: ", parameters_out["DMD_Picture_time[us]"], file=f)
        print("nbr_of_images: ", parameters_out["nbr_of_images"], file=f)
    print("Data of this packet saved")


generator = MCS_devices.MCS_STG()
recorder = MCS_devices.MCS_MEA(MCS_devices.McsBusTypeEnumNet.MCS_USB_BUS)
dmd = communication.DMD()

parameters = dict()
########## Parameters to complete ##########
parameters["electrodes_sampling_freq[Hz]"] = 10000
parameters["trigger_freq[Hz]"] = 60
parameters["images_directory"] = 'images/*.png'
parameters["DMD_Picture_time[us]"] = 10e3
############################################

if parameters["DMD_Picture_time[us]"] > (1/parameters["trigger_freq[Hz]"])*1e6:
    print("DMD picture time bigger than T not possible")
    print("The program will close")
    input("Press enter to close the program...")
    sys.exit()

list_of_sequences, name_of_sequences, parameters["nbr_of_images"] = import_and_compile_images()
nbr_of_packets = int(len(list_of_sequences))

# configuration of the electrodes amplifier
recorder.file_to_save_data()
recorder.recorder_settings(parameters["electrodes_sampling_freq[Hz]"])

# for loop that will send and record batches of 2730 images (maximum number of images, the DMD can hold)
for i in range(nbr_of_packets):
    nbr_images_in_this_packet = int(len(list_of_sequences[i])/(768*1024))

    # configuration of the DMD with the wanted parameters
    dmd.controlProj('ALP_PROJ_MODE','ALP_SLAVE')
    dmd.controlDev('ALP_EDGE_RISING')
    dmd.allocSeq(name_of_sequences[i], nbr_images_in_this_packet)
    print("Starting loading image")
    start_loading = time.time()
    dmd.putSeq(name_of_sequences[i], list_of_sequences[i])
    print("The transfer took",(time.time()-start_loading), "seconds")
    print("All image loaded")
    dmd.timingSeq(name_of_sequences[i], int(parameters["DMD_Picture_time[us]"]))

    # configuration of the stimulus generator
    parameters["trigger_Thigh[us]"] = 1000
    parameters["trigger_Tlow[us]"] = int((1/ parameters["trigger_freq[Hz]"])*1e6) - 1000
    generator.trigger_settings(parameters["trigger_Tlow[us]"], parameters["trigger_Thigh[us]"], nbr_images_in_this_packet)

    # Beginning of the acquisition by the electrodes amplifier
    if (i==0):
        recorder.StartDacq()
    else:
        recorder.SendStartDacq()

    # Start the DMD, it will wait for a trigger comming from the stimulus generator
    dmd.startProj(name_of_sequences[i])
    time.sleep(1)

```

```python
    # Sending the "start" command to the stimulus generator
    generator.start_trigger()
    start_trigger = time.time()

    dmd.waitProj() #pauses the script until the actually playing sequence has finished
    dmd.freeSeq(name_of_sequences[i])

    time.sleep(5*(parameters["trigger_Thigh[us]"]+parameters["trigger_Tlow[us]"])/1e6)
    recorder.SendStopDacq()
    time.sleep(int(parameters["electrodes_sampling_freq[Hz]"]/300))

    print("Sequence", (i + 1), "over", nbr_of_packets, "displayed")

parameters["file_name"] = recorder.get_file_name()
save_experiment_parameters(parameters)
input("Press enter when all the data is arrived...")
recorder.StopDacq
recorder.Disconnect()
generator.Disconnect()
dmd.free()

```

```
 1 """
 2 Laboratory of Applied Photonics Devices - EPFL
 3 Fall 2020
 4 Victor Tiberghien
 5 
 6 Script that process the raw data generated by the main script experimentv2.py
 7 - input: name of the raw .csv file on line 52
 8 - output: .csv file in the working directory "electrode_..._processed_... .csv"
 9 
10 """
11 
12 import csv
13 from datetime import datetime
14 
15 
16 def save_data(electrodes_data, file_to_save):
17     with open(file_to_save, mode='a', newline='') as csv_file:
18         writer = csv.writer(csv_file)
19         row_of_data = electrodes_data
20         writer.writerow(row_of_data)
21 
22 
23 def data_treatment(file_to_open):
24     today = datetime.today()
25     now = today.strftime("%d-%b-%Y_%HH-%MM")
26     processed_time = 'processed_' + now
27     file_to_save = file_to_open.replace('raw', processed_time)
28 
29     with open(file_to_open, newline='') as csvfile:
30         spamreader = csv.reader(csvfile, delimiter=',')
31         row_nbr=0
32         print("Processing...")
33         for row in spamreader:
34             data = row
35             if len(row)%257==0:
36                 sample = int(len(row)/257)
37                 for j in range(0, sample):
38                     data_to_save = [None] * 257
39                     for k in range(0, 257):
40                         if 0 <= k <= 251:
41                             data_to_save[k] = (float(data[(j * 257) + k])/(65535/7.4))-3.7
42                         if 252 <= k <= 255:
43                             data_to_save[k] = ((float(data[(j * 257) + k])/(65535/8192))-4096)/1000
44                         if k == 256:
45                             data_to_save[k] = float(data[(j * 257) + k])
46                     save_data(data_to_save, file_to_save)
47             elif row_nbr != 0:
48                 print("missing data")
49             row_nbr = row_nbr+1
50     print("Processing done!")
51 
52 
53 file_name = 'Experiment_03-Dec-2020_12H-54M_raw.csv'
54 data_treatment(file_name)
55 
```

```python
"""
Laboratory of Applied Photonics Devices - EPFL
Fall 2020
Victor Tiberghien

Module that interfaces with the devices from Multichannels systems:
- Stimulus Genereator, STG4004
- Microelectrodes array, USB-MEA256

"""
import clr
import os
from System import Action
from System import *
import System
import sys
import csv
import time
import ctypes


path = str(repr(os.getcwd()))+'\\McsUsbNet.dll'
path = path.replace("'",'')
dll_ref = System.Reflection.Assembly.LoadFile(path)

from Mcs.Usb import CMcsUsbListNet
from Mcs.Usb import DeviceEnumNet

from Mcs.Usb import CMeaDeviceNet
from Mcs.Usb import McsBusTypeEnumNet
from Mcs.Usb import DataModeEnumNet
from Mcs.Usb import SampleSizeNet

from Mcs.Usb import CStg200xDownloadNet
from Mcs.Usb import McsBusTypeEnumNet
from Mcs.Usb import STG_DestinationEnumNet
from datetime import datetime


# class that controls that interface with the stimulus generator
class MCS_STG(CStg200xDownloadNet):
    def __init__(self):
        self.USB_location = self.looking_for_generator()
        self.Stg200xPollStatusEvent += self.PollHandler;
        self.Connect(self.USB_location)

    def looking_for_generator(self):
        deviceList = CMcsUsbListNet(DeviceEnumNet.MCS_DEVICE_USB)  # List of connected MCS devices
        print("found %d devices" % (deviceList.Count))
        for i in range(deviceList.Count):                           # Scan for USB devices
            listEntry = deviceList.GetUsbListEntry(i)
            print("Device: %s   Serial: %s" % (listEntry.DeviceName, listEntry.SerialNumber))
            if (listEntry.DeviceName == "STG4004"):                 # Looks for the stimulus generator
                generator_entry = i
        try:
            generator_entry
        except:
            print("Stimuli generator not detected!")
            print("The program will close")
            input("Press enter to close the program...")
            sys.exit()
        return deviceList.GetUsbListEntry(generator_entry)

    def PollHandler(self, status, stgStatusNet, index_list):
        print('%x %s' % (status, str(stgStatusNet.TiggerStatus[0])))

    def get_precision(self):
        voltageRange = self.GetVoltageRangeInMicroVolt(0);
        voltageResulution = self.GetVoltageResolutionInMicroVolt(0);
        currentRange = self.GetCurrentRangeInNanoAmp(0);
        currentResolution = self.GetCurrentResolutionInNanoAmp(0);
```

```python
        print('Voltage Mode:  Range: %d mV  Resolution: %1.2f mV' % (voltageRange / 1000, voltageResolution / 1000.0))
        print('Current Mode:  Range: %d uA  Resolution: %1.2f uA' % (currentRange / 1000, currentResolution / 1000.0))

    def trigger_settings(self, Thigh = 100000, Tlow = 100000, nbr_of_repetition=1):
        self.ClearSyncData(0);
        self.ClearSyncData(1);
        self.ClearSyncData(2);
        self.ClearSyncData(3);
        amplitude = Array[UInt16]([0, 1])  # setup the trigger pulse
        duration = Array[UInt64]([Thigh, Tlow])  # Duration of the low and high in microseconds
        channelmap = Array[UInt32]([0, 0, 0, 0])

        # bitmap of the sync out outputs to activate, 15 corresponds to 1111 which will activate all 4 sync out outputs.
        # In order to activate just 3, you have to enter 7 which corresponds to 0111
        syncoutmap = Array[UInt32]([15, 0, 0, 0])
        repetition = Array[UInt32]([nbr_of_repetition, nbr_of_repetition, nbr_of_repetition, 0])

        self.SetupTrigger(0,channelmap, syncoutmap, repetition)

        self.SendSyncData(0, amplitude, duration)  # Send the pulse configuration to the STG4004
        self.SendSyncData(1, amplitude, duration)
        self.SendSyncData(2, amplitude, duration)
        self.SendSyncData(3, amplitude, duration)


    def start_trigger(self):
        self.SendStart(1)

    def disconnect(self):
        self.Disconnect()


# class that controls that interface with the electrodes amplifier
class MCS_MEA(CMeaDeviceNet):
    def __init__(self, arg):
        self.USB_location = self.looking_for_recorder()
        self.ChannelDataEvent += self.OnChannelDatav2
        self.ErrorEvent += self.OnError
        self.Connect(self.USB_location)
        self.previous_state = True
        self.available_channels = 0
        self.file_data = 'Experiment_n.csv'
        self.counter = 0
        self.sampling_rate = 5000

    def looking_for_recorder(self):
        deviceList = CMcsUsbListNet(DeviceEnumNet.MCS_DEVICE_USB)  # List of connected MCS devices
        print("found %d devices" % (deviceList.Count))
        for i in range(deviceList.Count):  # Scan for USB devices
            listEntry = deviceList.GetUsbListEntry(i)
            print("Device: %s   Serial: %s" % (listEntry.DeviceName, listEntry.SerialNumber))
            if (listEntry.DeviceName == "USB-MEA256"):                          #Looks for the electrodes amplifier
                recorder_entry = i
        try:
            recorder_entry
        except:
            print("Electrodes amplifier not detected!")
            print("The program will close")
            input("Press enter to close the program...")
            sys.exit()
        return deviceList.GetUsbListEntry(recorder_entry)

    def OnError(self, msg, info):
        print(msg, info)

    def get_number_of_available_channels(self):
        self.available_channels = self.HWInfo().GetNumberOfHWADCChannels(0)
```

```python
        print("Number of channels availible", self.available_channels)

    # call back function that is called when a new packet of data is ready to be sent
    def OnChannelDatav2(self, x, cbHandle, numSamples):
        self.counter = self.counter + 1
        nbr_of_samples = int(self.sampling_rate/2) # nbr_of_sample before sending the data
        data, size = self.ChannelBlock_ReadFramesUI16(0, nbr_of_samples, Int32(0))
        print("Size:", size)
        print("size: %d numSamples: %d Data: %04x" % (size, numSamples, data[0]))
        self.save_data(self.counter, data)

    def get_counter(self):
        return self.counter

    def recorder_settings(self, sampling_r):
        self.sampling_rate = sampling_r
        self.SetNumberOfChannels(256)
        self.EnableDigitalIn(Boolean(True), UInt32(0))  # Enable the Digital-in on the MEA-256

        self.SetDataMode(DataModeEnumNet.Unsigned_16bit, 0)
        self.SetSamplerate(self.sampling_rate, 1, 0)  # Sample rate in Hz
        self.EnableChecksum(False, 0)
        print("Channels in Block: ", self.GetChannelsInBlock(0))
        self.SetSelectedData(self.GetChannelsInBlock(0), 1000000, int(self.sampling_rate/2), SampleSizeNet.SampleSize16Unsigned,
                                 self.GetChannelsInBlock(0))

    #creation of the .csv file in which the data will be saved
    def file_to_save_data(self):
        self.ClearBuffers()
        file = 'Experiment_n.csv'
        today = datetime.today()
        now = today.strftime("_%d-%b-%Y_%HH-%MM")+'_raw'
        self.file_data = file.replace('_n', now)
        with open(self.file_data, mode='w', newline='') as csv_file:  # Creation of the CSV file to save data
            writer = csv.writer(csv_file)

    def save_data(self, image_nbr, electrodes_data):
        with open(self.file_data, mode='a', newline='') as csv_file:
            writer = csv.writer(csv_file)
            row_of_data = electrodes_data
            writer.writerow(row_of_data)

    # returns the name of the data file in order to saved the parameters of the experiment in a corresponding .txt file
    def get_file_name(self):
        txt_file = self.file_data.replace('_raw.csv','.txt')
        return txt_file

    def disconnect(self):
        self.StopDacq()
        time.sleep(10)
        self.Disconnect()
```

```
"""
Laboratory of Applied Photonics Devices - EPFL
Fall 2020
Victor Tiberghien

Module that interface with the DMD from Vialux:
DMD model: DLP HI-SPEED V-MODULE
    - 0.7” XGA 2x LVDS (VIS) DMD for visible light
    - ALP-4.2 “high-speed”

Modified and completed from a version written by Matthias Müller-Schrader in 2015
- https://gitlab.phys.ethz.ch/mohanj/holography/-/blob/096ce42d18efc5f4eda14a53013a4cffb3220830/dmd/communication.py
"""


import ctypes
from PIL import Image
import sys
import os
import numpy as np
from ctypes import *

### Constants or controlling arguments, see documentation of ALP-4.2 high speed
ALP_DEFAULT=ctypes.c_int(0)
ALP_DEVICE_NUMBER = ctypes.c_int(2000)
ALP_VERSION = ctypes.c_int(2001)
ALP_TRIGGER_EDGE = ctypes.c_int(2005)
ALP_DEV_DISPLAY_HEIGHT = ctypes.c_int(2057)
ALP_DEV_DISPLAY_WIDTH = ctypes.c_int(2058)
ALP_AVAIL_MEMORY = ctypes.c_int(2003)
ALP_USB_CONNECTION = ctypes.c_int(2016)
ALP_PBC_TEMPERATURE = ctypes.c_int(2052)
ALP_BITPLANES = ctypes.c_int(2200)
ALP_BITNUM = ctypes.c_int(2103)
ALP_BIN_MODE = ctypes.c_int(2104)
ALP_PICNUM = ctypes.c_int(2201)
ALP_PICTURE_TIME = ctypes.c_int(2203)
ALP_ILLUMINATE_TIME = ctypes.c_int(2204)
ALP_ON_TIME = ctypes.c_int(2214)
ALP_OFF_TIME = ctypes.c_int(2215)
ALP_MIN_ILLUMINATE_TIME = ctypes.c_int(2212)
ALP_DATA_FORMAT = ctypes.c_int(2110)
ALP_TRIGGER_IN_DELAY = ctypes.c_int(2207)                          # in us
ALP_MAX_TRIGGER_IN_DELAY = ctypes.c_int(2210)                      # in us

ALP_DATA_BINARY_TOPDOWN = ctypes.c_int(2)
BIT_PLANES = ctypes.c_long(8)           # before, it was 1!


try:
    #dmd_dll = ctypes.CDLL('libDMD/x64/alpV42.dll')
    dmd_dll = ctypes.CDLL(r'alpV42.dll')
except Exception:
    print ("Error occured while loading DMD-ddl")
    sys.exit()


ALP_ERR = {                           #giving better error messages, see otherwise documentation
    0: 'ALP_OK',
    1001: 'ALP_NOT_ONLINE',
    1002: 'ALP_NOT_IDLE',
    1003: 'ALP_NOT_AVAILABLE',
    1004: 'ALP_NOT_READY',
    1005: 'ALP_PARM_INVALID',
    1006: 'ALP_ADDR_INVALID',
    1007: 'ALP_MEMORY_FULL',
    1008: 'ALP_SEQ_IN_USE',
    1009: 'ALP_HALTED',
    1010: 'ALP_ERROR_INIT',
    1011: 'ALP_ERROR_COMM',
```

```
    1012: 'ALP_DEVICE_REMOVED',
    1013: 'ALP_NOT_CONFIGURED',
    1014: 'ALP_LOADER_VERSION',
    1018: 'ALP_ERROR_POWER_DOWN',

    }

ALP_CONTRL_ARGS = {     # Arguments for AlpSeqControl() and AlpProjControl
    'ALP_BIN_MODE'          : ctypes.c_int(2104),
    'ALP_DATA_FORMAT'       : ctypes.c_int(2110),
    'ALP_FIRSTFRAME'        : ctypes.c_int(2101),
    'ALP_LASTFRAME'         : ctypes.c_int(2102),
    'ALP_SEQ_REPEAT'        : ctypes.c_int(2100),
    'ALP_PROJ_MODE'         : ctypes.c_int(2300),
    'ALP_PROJ_INVERSION'    : ctypes.c_int(2306),
    'ALP_PROJ_UPSIDE_DOWN'  : ctypes.c_int(2307),
    'ALP_MASTER'            : ctypes.c_int(2301),
    'ALP_SLAVE'             : ctypes.c_int(2302),
    'ALP_DEFAULT'           : ctypes.c_int(0),
    'NOT_ALP_DEFAULT'       : ctypes.c_int(1),
    'ALP_EDGE_FALLING'      : ctypes.c_int(2008),
    'ALP_EDGE_RISING'       : ctypes.c_int(2009),
    'ALP_BIN_NORMAL'        : ctypes.c_int(2105),
    'ALP_BIN_UNINTERRUPTED' : ctypes.c_int(2106),

}

ALP_INQ_ARGS = {
    1200      : 'ALP_PROJ_ACTIVE',
    1201      : 'ALP_PROJ_IDLE',
    2301      : 'ALP_MASTER',
    2302      : 'ALP_SLAVE',
    2008      : 'ALP_EDGE_FALLING',
    2009      : 'ALP_EDGE_RISING',
    2105      : 'ALP_BIN_NORMAL',
    2106      : 'ALP_BIN_UNINTERRUPTED',
    0         : 'ALP_DATA_MSB_ALIGN',
    1         : 'ALP_DATA_LSB_ALIGN',
    2         : 'ALP_DATA_BINARY_TOPDOWN',
    3         : 'ALP_DATA_BINARY_BOTTOMUP',

}

def compilePicture(img, nbr_images):
    print("Compiling images...")
    if isinstance(img, np.ndarray) and img.ndim == 2:
        img = [img]
    tArray = np.zeros(nbr_images*1024*768, dtype='B')
    image_counter = 0
    for arr in img:
        arr.resize(1, (1024 * 768))
        tArray[image_counter*1024*768:(image_counter+1)*1024*768] = arr
        image_counter = image_counter + 1
        string = '\rProgress: ' + str(int(100 * image_counter / len(img))) + '%'
        sys.stdout.write(string)
    print('')
    print(int(len(tArray) / 786432), "images compiled in this package")
    return tArray


class DMD():
    """ Class to communicate with the DMD.

    Each instance of this class can communicate with one DMD.
    During the initialisation, it tries to connect to the next aviable DMD.
    It is also possible to connect to a special DMD, specified by its serial number.

    After the usage, the DMD should be released using the DMD.free() method.

    """

```

```python
    def __init__(self,serial_number = ALP_DEFAULT):
            #searching for DMD
        print ('searching for DMD')
        self.DevID = ctypes.c_int()       ### To store the device ID to communicate with DMD
        self.seq_ids = {}       ### To store sequenceIDs
        ret = dmd_dll.AlpDevAlloc(serial_number,ALP_DEFAULT,ctypes.byref(self.DevID))
        if ret != 0:
            print("DMD not detected")
            print("The program will close")
            input("Press enter to close the program...")
            raise Exception('Communication with DMD failed. Error %s'%ALP_ERR[ret])
            print("DMD not found")
            sys.exit()
        else:
            print ('Connected to DMD')
        ### determining resulution of DMD (for transforming pictures)
        self.disp_height = ctypes.c_int()
        ret = dmd_dll.AlpDevInquire(self.DevID,ALP_DEV_DISPLAY_HEIGHT,ctypes.byref(self.disp_height))
        self.disp_height = self.disp_height.value   # used c_int.value to get normal py int
        if ret != 0:
            raise Exception('Inspecting height failed. Error %s'%ALP_ERR[ret])
            input("Press enter to close the program...")
        self.disp_width = ctypes.c_int()
        ret = dmd_dll.AlpDevInquire(self.DevID,ALP_DEV_DISPLAY_WIDTH,ctypes.byref(self.disp_width))
        if ret != 0:
            raise Exception('Inspecting width failed. Error %s'%ALP_ERR[ret])
            input("Press enter to close the program...")
        self.disp_width = self.disp_width.value
        self.last_added_seq = None
        print("Diplay hight:", self.disp_height)
        print("Diplay width:", self.disp_width)

    def available_memory(self):
        memory=ctypes.c_long()
        ret = dmd_dll.AlpDevInquire(self.DevID, ALP_AVAIL_MEMORY, ctypes.byref(memory))
        if ret != 0:
            raise Exception('Inspecting left failed. Error %s' % ALP_ERR[ret])
            input("Press enter to close the program...")
        memory = memory.value
        print("Memory left on the DMD is: ", int(memory/8),"8 bits images")
        return memory

    def controlDev(self,tr_edge):
        """ Allows to set some properties to the DMD.

        Actually, it is only possible to change the trigger_edge, if the DMD is in the
        slave mode.

        **Implementation of ``AlpDevControl`` from the DLL.

        Parameters
        -----------------
        tr_edge : *int or str*
            Specifies the trigger edge. Can either be a number as specified in the
            DMD documentation or the string ``ALP_EDGE_RISING`` or ``ALP_EDGE_FALLING``.
        """
        if isinstance(tr_edge,str):
            c_tr_edge = ALP_CONTRL_ARGS[tr_edge]
        else:
            c_tr_edge = ctypes.c_int(tr_edge)
        ret = dmd_dll.AlpDevControl(self.DevID,ALP_TRIGGER_EDGE,c_tr_edge)
        if ret:         # ret == 0 is everything is ok.
            raise Exception('Changing trigger edge failed. Error %s'%ALP_ERR[ret])
            input("Press enter to close the program...")

    def free(self):
        """Allows to eject the DMD manually. Should always be done.

        **Implementation of ``AlpDevHalt`` and ``AlpDevFree`` from the DLL.

        Raises
```

```
        ----------------
        Exception :
            - If either ``AlpDevHalt`` or ``AlpDevFree`` returns an error
        """
        #ejecting the DMD manually
        ret1 = dmd_dll.AlpDevHalt(self.DevID)
        ret2 = dmd_dll.AlpDevFree(self.DevID)
        if(ret1+ret2==0):
            print ('DMD is free')
        elif ret1 != 0:
            raise Exception('Halting the DMD failed! Error %s'%ALP_ERR[ret1])
        else:
            raise Exception('Freeing the DMD failed! Error %s'%ALP_ERR[ret2])

    def inquireDev(self,conv=True):
        """ Helps inspecting the DMD.

        ** Implementation of some parts of ``AlpDevInquire`` from the DLL.

        By default, the values are returned a converted form.
        If ``converted`` is False, the values (except from display height and width)
        will be returned as they come from the DMD, i.e. as ctypes.c_int.

        Returns
        ----------------
        propts : *dict*
                Dictionary containing the properties. Keys are (as string):
                    -``Device_Number``:
                        Serial number of the DMD (can be used later to connect
                        to specific DMD  by handling it to the initialization routine).
                    -``ALP_Version_Number``:
                        The version number of the ALP device.
                    -``Temperature_PBC``:
                        The internal temperature of the DMD.
                        *If ``converted`` is True, the temperature will be stated
                        in degree celsius.*
                    -``Trigger_Edge``:
                        Whether the DMD reacts to rising or falling triggers.
                        *If ``converted`` is True, the entry will be a string
                        'ALP_EDGE_FALLING' or 'ALP_EDGE_RISING'.*
                    -``USB_Connection``:
                        Whether the connection is ok or removed.
                        *If ``converted`` is True, the entry will be a string
                        'ALP_OK' or 'ALP_DEVICE_REMOVED'.*
                    -``Display_Height``
                        Height of the DMD (type python [sic] int).
                    -``Display_Width``
                        Width of the DMD  (type python [sic] int).

        Raises
        ------------------------
        Exception:
            - If one of the calls of ``AlpDevInquire`` returns an error.
        """

        ditc = {'Display_Height':self.disp_height,'Display_Width':self.disp_width}
        ditc['Device_Number'] = ctypes.c_int(0)
        ret = dmd_dll.AlpDevInquire(self.DevID,ALP_DEVICE_NUMBER,ctypes.byref(ditc['Device_Number']))
        ditc['ALP_Version_Number'] = ctypes.c_int(0)
        ret += dmd_dll.AlpDevInquire(self.DevID,ALP_VERSION,ctypes.byref(ditc['ALP_Version_Number']))
        ditc['Trigger_Edge'] = ctypes.c_int(0)
        ret = dmd_dll.AlpDevInquire(self.DevID,ALP_TRIGGER_EDGE,ctypes.byref(ditc['Trigger_Edge']))
        if conv:   # See ALP documentation for the numbers
            ditc['Trigger_Edge']= ALP_INQ_ARGS[ditc['Trigger_Edge'].value]
        ditc['USB_Connection'] = ctypes.c_int(0)
        ret += dmd_dll.AlpDevInquire(self.DevID,ALP_USB_CONNECTION,ctypes.byref(ditc['USB_Connection']))
        if conv:
            ditc['USB_Connection'] = ALP_ERR[ditc['USB_Connection'].value]
        ditc['Temperature_PBC'] = ctypes.c_int(0)
        ret += dmd_dll.AlpDevInquire(self.DevID,ALP_PBC_TEMPERATURE,ctypes.byref(ditc['Temperature_PBC']))
        if conv:
```

```
           ditc['Temperature_PBC'] = ditc['Temperature_PBC'].value/256.
        if ret != 0:
            raise Exception('Error occured while inspecting DMD')
        return ditc

    def allocSeq(self,name,picNum,data_format = 0):
        """ Allocates memory to store later a sequence of pictures

        ** Implementation of ``AlpSeqAlloc`` and party of ``AlpSeqControl`` from the DLL.

        Parameters
        -----------------
        name : *any type that can be key for a dict*
             Name for the sequence. It can be accessed by DMD.seq_ids[name]
        picNum :  *int*
             The number of XGA pictures belonging to the sequence.
             Could be limited by memory (but unlikely).
        data_format : *opt, int from {0,1,2,3}*
            Specifies the data format for the pictures of the sequence.
            Other modules are designed for the default (Bitplanes, row 0 first).
            See the documentation of the ALP library for more details (default is
            ALP_DATA_BINARY_TOPDOWN). Integers will be converted to ctypes.c_int

        Raises
        -------------
        Exception:
            - If either ``AlpSecAlloc`` or ``AlpSecControl`` returns an error.
        """

        seqID = ctypes.c_int()
        c_picNum = ctypes.c_long(picNum)
        ret = dmd_dll.AlpSeqAlloc(self.DevID,BIT_PLANES,c_picNum,ctypes.byref(seqID))
        if ret != 0:
            raise Exception('Allocation of memory failed. Error %s'%ret)
        else:
            print ('Successfully allocated memory for sequence %s'%name)
        self.seq_ids[name] = seqID              ### All sequence IDs are stored in this dict.
        self.last_added_seq = name
        c_data_format = ctypes.c_int(data_format)       # See ALP Documentation for other formate
        #ret = dmd_dll.AlpSeqControl(self.DevID,seqID,ALP_DATA_FORMAT,c_data_format)        ------------------------<
        if ret != 0:
            raise Exception('Changing data format to allocate sequence %s failed. Error %s'%(name,ret))

    def freeSeq(self,name):
        """ Releases a sequence and releases therby the memory allocated by the sequence.

        ** Implementation of ``AlpSeqFree`` from the DLL.

        Raises
        ---------------------
        Exception:
            - If ``AlpSeqFree`` returns an error.
        """

        ret = dmd_dll.AlpSeqFree(self.DevID,self.seq_ids[name])
        if ret != 0:
            raise Exception('Releasing sequence %s failed. Error %s' %(str(name), ALP_ERR[ret]))
        else:
            del self.seq_ids[name]
            print ('Released sequence %s :)'%name)
            self.last_added_seq = None

    def inquireSeq(self,name,conv=True):
        """ Allows to inquire a sequence and returns a dict with the most important properties.

        ** Implementation of parts of ``AlpSeqInquire`` from the DLL.

        By default, the values are returned a converted form.
        If ``converted`` is False, the values will be returned as they come
        from the DMD, i.e. as ctypes.c_int.
```

```

        Returns
        ----------------
        propts : *dict*
                Dictionary containing the properties. Keys are (as string):
                    -``Seq_Bitplanes``:
                        Bit depth of the pictures in the sequence. Should be 1.
                    -``Seq_Bitnum``:
                        The bit depth for displaying  (could reduce bitdepth for showing).
                        Should also be 1.
                    -``Seq_Bin_Mode``:
                        If bitplanes or bitnum = 1 (binary mode), it is possible
                        to use a mode without  dark phase. Shows, whether this
                        mode is active.
                    -``Seq_Picnum``:
                        Number of pictures in the sequence.
                    -``Seq_Pic_Time``:
                        Time beween start of two consecutive pictures (in micro s).
                        The illumination time might be smaller but is chosen so that
                        it is maximal.
                    -``Seq_Illum_Time``:
                        Time, one picture is displayed on the DMD. Is <= ``Seq_Pic_Time`` - 44 microseconds.
                        If the DMD is in ``ALP_BIN_UNINTERRUPTED`` mode, it will be set to
                        0 and ignored.
                    -``Seq_Min_Illuminate_Time``:
                        Minimal possible value for ``Seq_Illuminate_Time``. (in mirco s)
                    -``Seq_Data_Format``
                        Data format of the sequence
                    -``Seq_ON_Time`` :
                        Total active projection time.
                    -``Seq_OFF_Time`` :
                        Total inactive projection time.
        """
        ditcsq = {}     ### To be read in blocks of 4 lines; init, query, test if conv, convert
        ditcsq['Seq_Bitplanes']=ctypes.c_int()
        ret = dmd_dll.AlpSeqInquire(self.DevID,self.seq_ids[name],ALP_BITPLANES,ctypes.byref( ditcsq['Seq_Bitplanes']))
        if conv:
            ditcsq['Seq_Bitplanes'] = ditcsq['Seq_Bitplanes'].value
        ditcsq['Seq_Bitnum']=ctypes.c_int()
        ret += dmd_dll.AlpSeqInquire(self.DevID,self.seq_ids[name],ALP_BITNUM,ctypes.byref( ditcsq['Seq_Bitnum']))
        if conv:
            ditcsq['Seq_Bitnum']=ditcsq['Seq_Bitnum'].value
        ditcsq['Seq_Bin_Mode']=ctypes.c_int()
        ret += dmd_dll.AlpSeqInquire(self.DevID,self.seq_ids[name],ALP_BIN_MODE,ctypes.byref( ditcsq['Seq_Bin_Mode']))
        if conv:
            ditcsq['Seq_Bin_Mode'] = ALP_INQ_ARGS[ditcsq['Seq_Bin_Mode'].value]
        ditcsq['Seq_Picnum']=ctypes.c_int()
        ret += dmd_dll.AlpSeqInquire(self.DevID,self.seq_ids[name],ALP_PICNUM,ctypes.byref( ditcsq['Seq_Picnum']))
        if conv:
            ditcsq['Seq_Picnum']=ditcsq['Seq_Picnum'].value
        ditcsq['Seq_Pic_Time']=ctypes.c_int()
        ret += dmd_dll.AlpSeqInquire(self.DevID,self.seq_ids[name],ALP_PICTURE_TIME,ctypes.byref( ditcsq['Seq_Pic_Time']))
        if conv:
            ditcsq['Seq_Pic_Time'] = str(ditcsq['Seq_Pic_Time'].value/1000.) + ' ms'
        ditcsq['Seq_Illuminate_Time']=ctypes.c_int()
        ret += dmd_dll.AlpSeqInquire(self.DevID,self.seq_ids[name],ALP_ILLUMINATE_TIME,ctypes.byref( ditcsq['Seq_Illuminate_Time']))
        if conv:
            ditcsq['Seq_Illuminate_Time'] = str(ditcsq['Seq_Illuminate_Time'].value/1000.) + ' ms'
        ditcsq['Seq_Min_Illum_Time']=ctypes.c_int()
        ret += dmd_dll.AlpSeqInquire(self.DevID,self.seq_ids[name],ALP_MIN_ILLUMINATE_TIME,ctypes.byref( ditcsq['Seq_Min_Illum_Time']))
        if conv:
            ditcsq['Seq_Min_Illum_Time'] = str(ditcsq['Seq_Min_Illum_Time'].value/1000.) + ' ms'
        ditcsq['Seq_ON_Time']=ctypes.c_int()
        ret += dmd_dll.AlpSeqInquire(self.DevID,self.seq_ids[name],ALP_ON_TIME,ctypes.byref( ditcsq['Seq_ON_Time
```

```python
']))
        if conv:
            ditcsq['Seq_ON_Time'] = str(ditcsq['Seq_ON_Time'].value/1000.) + ' ms'
        ditcsq['Seq_OFF_Time']=ctypes.c_int()
        ret += dmd_dll.AlpSeqInquire(self.DevID,self.seq_ids[name],ALP_OFF_TIME,ctypes.byref( ditcsq['Seq_OFF_Time']))
        if conv:
            ditcsq['Seq_OFF_Time'] = str(ditcsq['Seq_OFF_Time'].value/1000.) + ' ms'
        ditcsq['Seq_Data_Format']=ctypes.c_int()
        ret += dmd_dll.AlpSeqInquire(self.DevID,self.seq_ids[name],ALP_DATA_FORMAT,ctypes.byref( ditcsq['Seq_Data_Format']))
        if conv:
            ditcsq['Seq_Data_Format']=ALP_INQ_ARGS[ditcsq['Seq_Data_Format'].value]
        if ret != 0:
            raise Exception('Error while inquirering sequence %s.'%name)
        return ditcsq

    def controlSeq(self,name,arg,num):
        """ Allows to control properties of the sequence.

        Parameters
        --------------------
        name :
            Name the sequence was allocated with.
        arg : *string*
            Property to be changed. One of the following:
             -``ALP_BIN_MODE`` :
                    Allows to control, wheter the sequence should be displayed
                    normally (0) or in uninterrupted mode (2106).
                    Can also pass ``ALP_BIN_NORMAL`` or ``ALP_BIN_UNINTERRUPTED``
                    as string.
                    **Requires a following call of ``DMD.timingSeq()`` to become active
             -``ALP_DATA_FORMAT`` :
                     Allows to change the data format.  See ALP-Documentation for further details.
             -``ALP_FIRSTFRAME``  :
                     Allows to restrict the pictures to be shown.
                     Selects the first picture of the sequence to be shown.
             -``ALP_LASTFRAME``   :
                     Allows to restrict the pictures to be shown.
                     Selects the last picture of the sequence to be shown.
             -``ALP_SEQ_REPEAT``  :
                     Sets how often the sequence should be shown when DMD.startProj(seq) is called.
                     Default is 1.
        num : *int*
            A parameter to specify the changement.
        """
        if isinstance(num,str):
           c_num = ALP_CONTRL_ARGS[num]
        else:
           c_num = ctypes.c_int(num)
        ret = dmd_dll.AlpSeqControl(self.DevID,self.seq_ids[name],ALP_CONTRL_ARGS[arg],c_num)
        if ret != 0:
            raise Exception('Changing argument %s of sequence %s failed. Error %s' %(ALP_CONTRL_ARGS[arg],name,ALP_ERR[ret]))

    def putSeq(self,name,data_array):
        """ Passes a numpy array of length l = (pic_num*display_height*display_width/8) to the DMD

        **Implementation of ``AlpSeqPut`` from the DLL.
        See ALP documentation for further details.
        """
        print("Uploading images to the DMD...")
        #array_pointer = data_array.ctypes.data_as(POINTER(c_ubyte))
        array_pointer = data_array.ctypes.data_as(POINTER(c_char))
        #array_pointer = data_array.ctypes.data         #creates the pointer, a np routine
        ret = dmd_dll.AlpSeqPut(self.DevID,self.seq_ids[name],ALP_DEFAULT,ALP_DEFAULT,array_pointer)
        if ret != 0:
            raise Exception('Putting pictures into sequence failed. Error %s' %ALP_ERR[ret])
        else:
            print ('loaded data for sequence %s on dmd :)'%name)

```

```

    def timingSeq(self,name,illuminate_time=None):
        """ Allows to set the picture time.

        **Implementation of parts of ``AlpSeqTiming`` from the DLL.
        Picture time should be in microseconds. Maximum is 10s.
        The picture time is the time between the start of two consecutive pictures.
        Can optionally also change the illumination time for ``ALP_BIN_NORMAL`` mode.
        The Illumination time is the time, the picture is actually viewed.

        Parameters
        ------------------------
        pic_time : *int*
            The time between the start of two consecutive pictures.
            If None, it will be set to the smallest possible time compatible with
            illumination time. If both are None, it will be set to
            1/30 second.
        illuminate_time : *int*
            The time a picture will be illuminated.  If None, it will be the
            maximal possible time; approxemately pic_time - 44 miroseconds.
        """
        pic_time = illuminate_time+45
        if not illuminate_time:
            illuminate_time = 0
        if not pic_time:
            pic_time = 0
        print("illu time: ", illuminate_time)
        print("pic time: ", pic_time)
        ret = dmd_dll.AlpSeqTiming(self.DevID,self.seq_ids[name],ctypes.c_long(int(illuminate_time)),ctypes.c_long(int(pic_time)),ALP_DEFAULT,ALP_DEFAULT,ALP_DEFAULT)
        if ret != 0:
            raise Exception('Changing time failed. Error %s' %ALP_ERR[ret])

    def controlProj(self,cont_type, cont_value):
        """ Allows to control the project.

        **Implementation of ``AlpProjControl`` from the DLL.

        The control parameters can also be passed as integers, accoding to the documentation.

        Parameters
        -------------
        cont_type : *str*
            One can change the following properties :
                -``ALP_PROJ_MODE`` :
                    Changes the projection mode. Possible cont_value are:
                    - ``ALP_MASTER`` : The pictures are refreshed by the DMD accoding to the settings by DMD.timingSeq.
                    - ``ALP_SLAVE`` : The transition of a picture follows an external trigger.
                -``ALP_PROJ_INVERSION``:
                    Inverts the image pixels. Possible cont_value are
                    - ``ALP_DEFAULT``
                    - ``NOT_ALP_DEFAULT``
                -``ALP_PROJ_UPSIDE_DOWN``:
                    Flipps the image. Possible cont_value are
                    - ``ALP_DEFAULT``
                    - ``NOT_ALP_DEFAULT``
        """
        if isinstance(cont_type,str):
            c_cont_type = ALP_CONTRL_ARGS[cont_type]
        else:
            c_cont_type = ctypes.c_int(cont_type)
        if isinstance(cont_value,str):
            c_cont_value = ALP_CONTRL_ARGS[cont_value]
        else:
            c_cont_value = ctypes.c_int(cont_value)
        ret = dmd_dll.AlpProjControl(self.DevID,c_cont_type,c_cont_value)
        if ret != 0:
            raise Exception('Changing properties of Project failed. Error %s' %ALP_ERR[ret])
        else:
            print("Successfully change the ", cont_type, "to", cont_value)
```

```

    def startProj(self,seq_name=None):
        """ Starts projecting the sequence ``seq_name``.

        **Implementation of ``AlpProjStart`` from the DLL.

        If no argument is passed, the last_added_seq will be played.
        """
        if not seq_name:
            seq_name = self.last_added_seq
        ret = dmd_dll.AlpProjStart(self.DevID,self.seq_ids[seq_name])
        if ret != 0:
            raise Exception('Playing sequnce %s failed. Error %s.' %(seq_name,ALP_ERR[ret]))
        else:
            print ('playing sequence %s :)'%seq_name)

    def startContProj(self,seq_name=None):
        """ Starts continuously playing the sequence ``seq_name``.

        If None is given, starts the last inquired sequence.
        **Implementation of ``AlpProjStart`` from the DLL.
        """
        if not seq_name:
            seq_name = self.last_added_seq
        ret = dmd_dll.AlpProjStartCont(self.DevID,self.seq_ids[seq_name])
        if ret != 0:
            raise Exception('Playing continuously sequnce %s failed. Error %s.' %(seq_name,ALP_ERR[ret]))
        else:
            print ('playing continuously sequence %s :)'%seq_name)

    def waitProj(self):
        """Pauses the script until the acutally plaing sequence has finished. """
        ret = dmd_dll.AlpProjWait(self.DevID)
        if ret != 0:
            raise Exception('Waiting for sequnce failed. Error %s.' %ALP_ERR[ret])

    def haltProj(self):
        """ Stops the sequence currently running on the DMD.

        In fact it finishes the actually playing sequence and stops then.
        See semester thesis of Matthias Mueller-Schrader for details.
        """
        ret = dmd_dll.AlpProjHalt(self.DevID)
        if ret:
            raise Exception('Halting project failed. Error %s .' %ALP_ERR[ret])

    def inquireProj(self,conv=True):
        """ Returns some information about the project on the DMD.

        ** Implementation of ``AlpProjInquire`` from the DLL.

        By default, the arguments are passed in a converted way. Set conv=False to
        get them as c_int.

        Returns
        ------------------------
        tmp : *dict*
                Dictionary containing the properties. Keys are:

                ``ALP_PROJ_MODE`` :
                    The projection mode (master or slave).
                ``ALP_PROJ_STATE`` :
                    The actual state of the projection (active or idle).
        """
        tmp = {}
        tmp['ALP_PROJ_MODE'] = ctypes.c_int()
        ret = dmd_dll.AlpProjInquire(self.DevID,ALP_CONTRL_ARGS['ALP_PROJ_MODE'],ctypes.byref(tmp['ALP_PROJ_MODE']))
        if ret:
            raise Exception('Inquireing project failed. Error %s.' %ALP_ERR[ret])
        tmp['ALP_PROJ_STATE'] = ctypes.c_int()
```

```python
        ret = dmd_dll.AlpProjInquire(self.DevID,ctypes.c_int(2400),ctypes.byref(tmp['ALP_PROJ_STATE']))
        if ret:
            raise Exception('Inquireing project failed. Error %s.' %ALP_ERR[ret])
        if conv:
            for key in tmp.keys():
                tmp[key] = ALP_INQ_ARGS[tmp[key].value]    ### Convert to right format.
        return tmp


    def compilePicturev2(self, img):
        print("Compiling images...")
        if isinstance(img, np.ndarray) and img.ndim == 2:
            img = [img]
        tArray = np.zeros(0, dtype='B')
        image_nbr = 1
        for arr in img:
            arr.resize(1, (1024 * 768))
            tArray = np.append(tArray, arr)
            if image_nbr % 500 == 0:
                print("Progress", int(100 * image_nbr / len(img)), "%")
            image_nbr = image_nbr + 1
        print(int(len(tArray)/786432), "images compiled in this package")
        return tArray

    def compilePicturev3(self, img):
        print("Compiling images...")
        img.resize(1, (1024 * 768))
        #img = np.unpackbits(img)
        image_print = np.packbits(img)
        image_print.resize(768, 1024)
        imgagee = Image.fromarray(image_print)
        imgagee.save('table.png')
        return img

    def loadArrToDMD(self,name,img,timing = None,unint=True):
        """ Takes an (collection of) arrays, converts it into the right format
        and transforms it to the DMD.

        Parameters
        --------------------------
        name : *int,str... must be hashable*
            The name for the sequence. Is needed to be able to control the sequence later
            and to start it.
        img : *2dim numpy array or collections of it*
            The image(s). Each image should be a 2dim boolean numpy array with shapes
            (disp_height,disp_width). Several images can be handeld as list or tuple
            of arrays or  a 3-dim array with the pictures aligned along the axis 0.
        timing : *opt, float*
            The time each picture shuold be shown [microsecond].
        unint : *opt, bool*
            Whether the uninterrupted mode should be implemented or not.
            (See also documentation of API)
        """
        pckd = self.compilePicture(img)
        picnum = len(pckd) * 8 /(self.disp_height*self.disp_width)
        self.allocSeq(name,picnum)
        if unint and not timing:
            timing = self.inquireSeq(name)['Seq_Pic_Time']
        if unint:
            self.controlSeq(name,'ALP_BIN_MODE',2106)
        if timing or unint:
            self.timingSeq(name,timing)
        self.putSeq(name,pckd)

    def inspect(self,conv=True):
        """ Inspects the DMD and returns a dict with the most important values.

        Combines DMD.inquireDev(), DMD.inquireSeq(lastSeq), DMD.inquireProj() and
        returns a dictionarry containing all the keys from the methods.
        If the last allocated sequence was removed or no sequence was allocated,
        this information will not be added to the dict.
```

```
        """
        tmp = self.inquireDev(conv)         ### Infos from the device.
        if self.last_added_seq:                 ### Infos from the last seq (if existing).
            tmp.update(self.inquireSeq(self.last_added_seq,conv))
            tmp['Name of last alloc Seq'] = self.last_added_seq
        tmp.update(self.inquireProj(conv))  ### Infos from the proj.
        return tmp

```