Technische Universität München
Chair for Biological Imaging

Bachelor Thesis

Real Time Processing in Photoacoustic Microscopy

Author:

Erdem Başeğmez

Advisor:

Prof. Dr. Daniel Razansky

Co-Advisors:

Dr. Héctor Estrada Beltrán

Jake Turner

Organization:

Helmholtz Zentrum München

Munich, May 2013

Contents

1

Introduction

Optoacoustic imaging is an emerging technique to image biological tissues non-invasively and safely. The technique combines superiorities of acoustical and optical methods: low scattering of ultrasound with high absorption of light in depth of the biological tissues. Its hybrid modality makes it possible to obtain high resolution and sensitive contrast images.

The optoacoustic effect, also called the photoacoustic effect or the thermoacoustic effect was discovered by Alexander Graham Bell in 1880, who realized that acoustic waves are generated from objects illuminated by sunlight [1]. However the research made little progress until the 1970s, mostly due to the lack of appropriate light sources and sensitive ultrasound detectors.

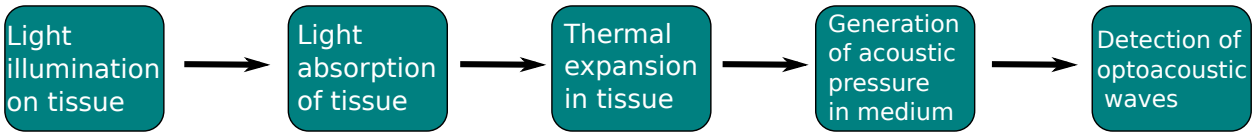

Figure 1.1: Generation of acoustic waves via the optoacoustic effect

The fundamental principle of generation of signals through optoacoustic effect is shown in Figure 1.1. An object is illuminated with nano-second pulses of electromagnetic radiation energy. The absorbed energy converts into heat and the temperature of the object increases, which then causes thermoelastic expansion. The thermoelastic expansion generates acoustic pressure in the medium, which contains information about the spatial distribution of the light absorption in the object. As shown in Figure 3.1, the acoustic waves are detected by the transducers and the image is represented with the position and the data from the detector with or without image reconstruction depending on the device setup.

Optoacoustic imaging is expected to offer applications in various fields in biology and medicine, from imaging of centimeter-large breast tumors to several micrometer-large single red blood cells [2]. Initial clinical applications include the imaging of melanoma

cancer, early chemotherapeutic response and tissue metabolism as well as the detection of breast and prostate cancer. Major preclinical applications include the visualization of angiogenesis, microcirculation, tumor microenvironments, drug response, brain function and gene activities [3].

Optoacoustic imaging devices have the potential to revolutionize the visualization of deep brain activities [4] and the diagnosis and treatment of metabolic diseases and cancers [3]. The devices that use optoacoustic effect for imaging tissues can be categorized into two modalities: Photoacoustic Computed Tomography (PACT) and Photoacoustic Microscopy (PAM).

PACT devices mostly have an array of transducers which receive the optoacoustic waves at different positions simultaneously. The use of multiple transducers thus enables real-time imaging of the region of interest and neglects the need of stages for scanning. However, such systems are inherently complex and expensive due to the bespoke nature and cost of the hardware. Additionally, the visualization relies on the reconstruction of the raw data [5].

On the other hand, PAM detects the optoacoustic waves with only one transducer. As such, it requires mechanical parts to scan the region of interest. Compared to PACT, PAM has a simple structure and less costs whilst raw sinograms are readily interpretable. PAM can provide high resolution images and the imaging of the acquired data from the transducer can take place in real-time while scanning the region of interest.

2

Objectives

The main objective of this thesis was to enable real time processing and visualization of a fast scanning coaxial optoacoustic microscope in MATLAB. The main problem faced controlling two data acquisition devices, Spectrum and National Instruments cards, simultaneous to the production of visual feedback on the data which was being acquired.

Problem Description Even though the mechanical scanning stages could run in the background using a macro, a strategy had to be developed to overcome the issues related to the sequential structure of MATLAB. Pure MATLAB code will not allow one to acquire and visualize data simultaneously due to hardware and driver software restrictions. Following the MATLAB-only approach, calibration has to run before the experiment so that the positions of the stages can be predicted. During an experiment, the motion of the stages and the data acquisition is done one after another. In other words, to acquire data of a certain position, stages have to stop first, when the data of the position is acquired, stages move to the next position. Due to the pausing and moving of the stages, scanning, and thus the imaging, is very slow.

Specific Objectives To resolve these issues, a backend coded in C++ programming language needs to be developed, which could be called directly from MATLAB using MATLAB Executable (MEX) files. This strategy should allow MATLAB functions and scripts to be called from inside the MEX instance to manipulate and visualize the data in real time. Running the macro in MATLAB to move the stages continously and acquiring data in MEX should enable to achieve fast scanning and real time visualization.

3

Theoretical Background

3.1 Light absorption

All physical substances consist of charged particles, such as electrons and protons. Electromagnetic waves interact with the substances due to their charged particles in the form of elastic scattering, absorption, etc. When an absorption occurs, the absorbed energy transforms into heat (thermal energy),causes a chemical reaction or is re-emitted. The portion of the absorbed energy which transforms into heat causes the photoacoustic effect to occur. The absorption depends on the properties of the illuminated tissue, such as molecular structure and ion density, the properties of the light, such as wavelength and polarization, and also the environmental conditions.

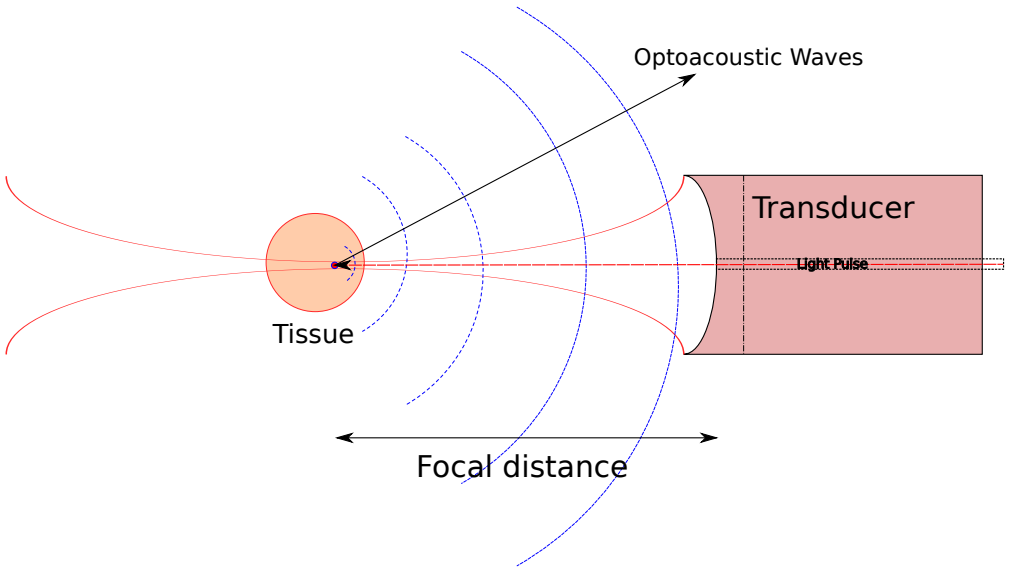

Figure 3.1: Optoacoustic wave generation and detection

The absorption coefficient of the molecule is highly sensitive to the wavelength of the

light, thus the optoacoustic detection of a specific absorber can be optimised by using a specific wavelength of light. Typical endogenous absorbers in tissues are hemoglobin, melanin and water, thus the imaging of vessels and skin is applicable for the photoacoustic detection devices, as shown in Figure 3.2. Besides this, contrast agents can be injected into the tissues for optoacoustic imaging.

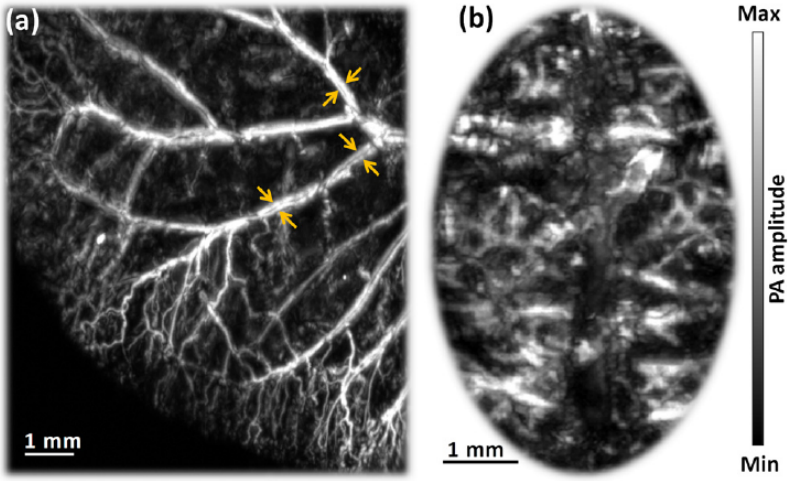

Figure 3.2: a. In vivo image of a mouse ear vasculature, b. in vivo image of a brain vasculature acquired through intact scalp and skull [5]

Optical and radio frequency waves (RF) are commonly used to generate the photoacoustic effect. The propagation the radiation in the tissue depends on its spectral region, which is modeled by radiative transfer equation (RTE). According to RTE, the fluence rate of radiation has parameters, such as the scattering coefficient μ_s($\approx$100 cm^{-1} in scattering tissues), the absorption coefficient μ_a (1 cm^{-1}) and reduced scattering coefficient $\mu\prime_s$ ($\approx$10 cm^{-1} in scattering tissues). In the diffusive regime, the attenuation of light depends on $\mu\prime_s$ and μ_a. Since the deviation of $\mu\prime_s$ due to the change in wavelength of radiation is notably higher than μ_a, the absorption by the tissue is significant for the illumination depth, thus the detection depth. The propagation of the RF waves depends on the RF polarization state and antenna design. It is also restricted due to the electromagnetic diffraction, which occurs when the wavelength is comparable to the size of the tissue. In addition, RF diffraction influences the global distribution of the electromagnetic field inside the tissue. Thus, the suitable spectrum to detect deeper than the ballistic light regime ($\approx$1 mm in the turbid tissue) is from $\approx$500 nm to NIR (near infrared), which can be called optical illumination. The radiation intensity used in photoacoustic measurements needs to be belower than the MPE (maximum permissible exposure) for skin, which guarentees safe illumination [6].

3.2 Wave Equation

When the radiation source illuminates the tissue, a pressure field is induced within the medium. The most common way for illumination is to use intense pulsed radiation sources [7]. The pulse width of the radiation is selected such that it meets the thermal confinement and acoustic stress confinement conditions so that thermal diffusion and volume expansion of the absorber during the illumination is neglectable. Ensuring these conditions via choosing a radiation source with pulse width of ≈10 ns simplifies to express the wave equation of the pressure field [2]. The heating function $H(r,t)$ indicates the thermal energy converted at spatial position r and time t. For the optical illumination $H(r,t)$ can be expressed as

$$H(r,t) = \mu_a(r)\Phi(r,t) \tag{3.1}$$

and under both stress and thermal confinement conditions heating time of the absorber can be treated as a Dirac delta fcuntion $\delta(t)$,

$$H(r',t') \approx A(r')\delta(t') \tag{3.2}$$

where $A(r')$ is the absorbed energy density (specific optical absorption).

The distance from the detector to the optoacoustic source, the absorber, is directly indicated with the time that the pressure wave requires to arrive at the detector. The resulting pressure waves from a spherical source in a homogenous, non-viscous medium can be expressed as [2]

$$p(r,t) = \frac{\beta}{4\pi C_p}\frac{\partial}{\partial t}\left[\frac{1}{v_s t}\int dr' A(r')\delta(t - \frac{|r-r'|}{v_s})\right], \tag{3.3}$$

where r is the spatial location of the detected pressure, t is time, β is the isobaric volume expansion coefficient, C_p is the isobaric specific heat, v_s is the acoustic speed.

The acoustic waves from a spherical source generally have a typical N-Shape(Figure 3.3) profile, having both positive and negative amplitudes [2].

3.3 Optoacoustic Waves

The magnitude of the generated acoustic wave is proportional to the local light intensity, optical absorption coefficient, thermoelastic properties and the size of the absorber and inversely proportional to the distance from the optoacoustic source, the absorber. The shape of the wave depends on the absorber's size and optical absorption properties [7] [2]. The spectrum of the induced acoustic wave depends on the duration of radiation pulse, the optical absorption variations and absorber size. The pulse width of the optoacoustic wave is proportional to the size of the optoacoustic source, thus the expected frequency of the wave is higher for smaller absorbers [2]. The optoacoustic waves, which

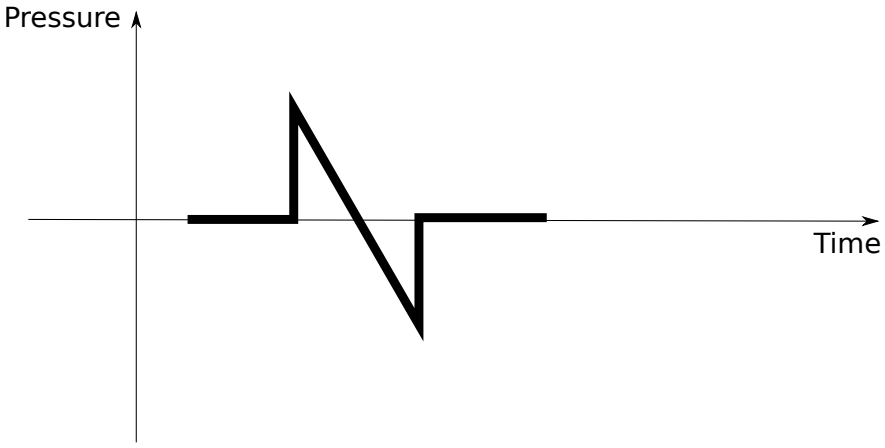

Figure 3.3: Optoacoustic waves generated by a spherical source

contain useful information, induced by biological tissues are in the ultrasonic spectrum between hundreds of kilohertz and tens of megahertz, when the radiation pulse width is in nanoseconds range [7].

Acoustic waves scatter significantly less (two to three orders) in the medium during propagation compared to the radiation, namely light [8]. The scattering of optoacoustic waves in soft tissues are commonly ignored. However, the attenuation coefficient of the acoustic waves increases significantly at high frequencies or in highly acoustic absorptive media. Furthermore, the tissue acts as a low-pass filter for high frequency acoustic waves, which reduces the spatial resolution. As a result, ultrasound attenuation influences the amplitude and the profile of the optoacoustic waves. It can be concluded that there is a compromise between resolution and the detection in depth, since higher resolution requires detection of higher frequency ultrasound waves, which leads to narrower detection depth due to higher acoustic absorption.

The optoacoustic detection depth relies both on the light absorption and the ultrasound attenuation. For appropriate imaging, finding the right combination of the radiation wavelength and the bandwidth of the optoacoustic detection is important. Furthermore, acoustic speed is also significant, which varies with the tissue. However, the tolerance of optoacoustic imaging to acoustic speed variation is higher than pure ultrasound detection, since optoacoustic waves propagate from the absorber to the detector only, whereas pure ultrasound detection requires a round trip.

4

Photoacoustic Microscopy

PAM detects optoacoustic waves using a focused transducer. The minimum axial resolution is limited by the wavelength of the optoacoustic waves($\frac{\lambda}{2}$) and the lateral resolution is determined according to the type of focusing: acoustic or optical focusing.

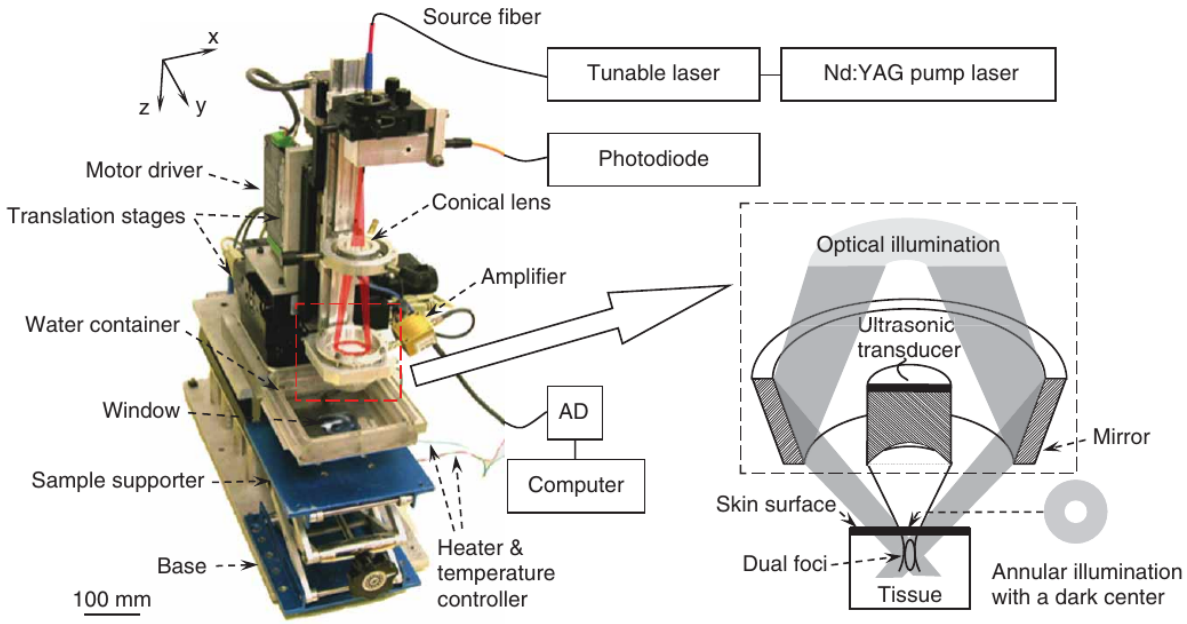

Figure 4.1: An example of PAM setup [8]

4.1 AR-PAM

Pure optical high resolution imaging methods are limited to detecting absorbers deeper than ≈1 mm in the skin, since high scattering dominates the propagation of light in tissue [8]. Compared to optical scattering, acoustic scattering is very low, thus acoustic focusing

is more applicable for the tissues, in which optical scattering is very high. AR-PAM suppresses the acoustic waves generated outside the focal zone and it can image deep in the tissue with high lateral resolution as per the acoustic focal width, where the axial resolution can be determined by the detector's bandwidth. Depending on the setup, the imaging depth can reach several centimeters.

4.2 OR-PAM

Achieving higher resolution is possible by detecting acoustic waves of higher frequency over shorter distances. However, at higher frequency the acoustic waves attenuate [3], which limits imaging depth, as discussed in Section 3.3. Thus, optical focusing is applicable for imaging within $\approx$1 mm, which also provides lateral resolution, where axial resolution is still determined by the detector's bandwidth. The imaging depth of OR-PAM is similar to high resolution pure optical imaging methods. However, higher image contrast can be achieved with OR-PAM. A setup which combines both OR-PAM and confocal optical microscopy can achieve both light scattering and absorption characteristics of the target at high resolution.

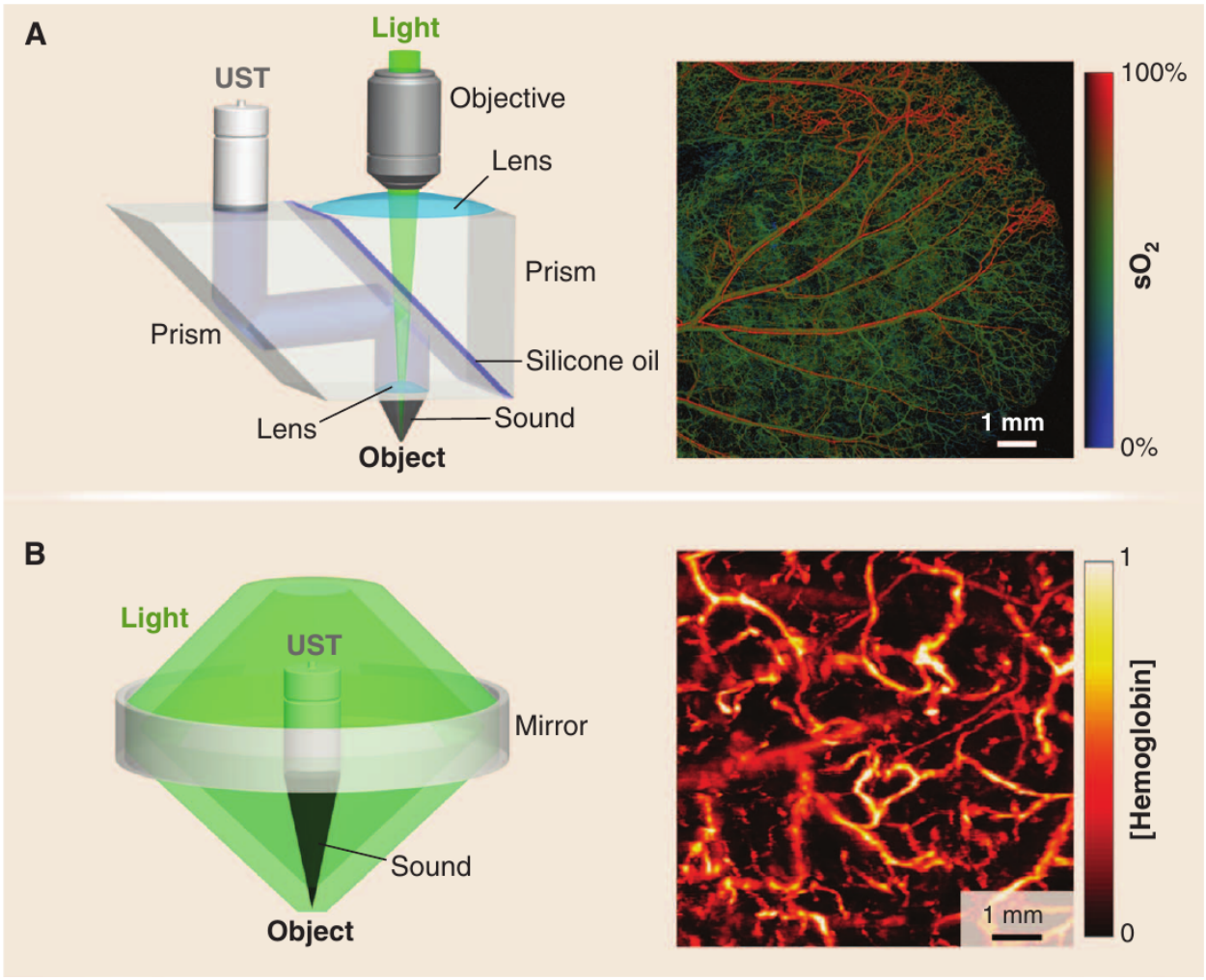

Figure 4.2: A. OR-PAM of $s0_2$ in a mouse ear, B. AR-PAM of normalized total hemoglobin concentration, in a human palm [3]

5

Device

The main parts of our PAM setup consisted of the laser (light) source, photodiode, mechanical stages, laser distance sensor, mount (Figure 5.1), ultrasound transducer and data acquisition cards.

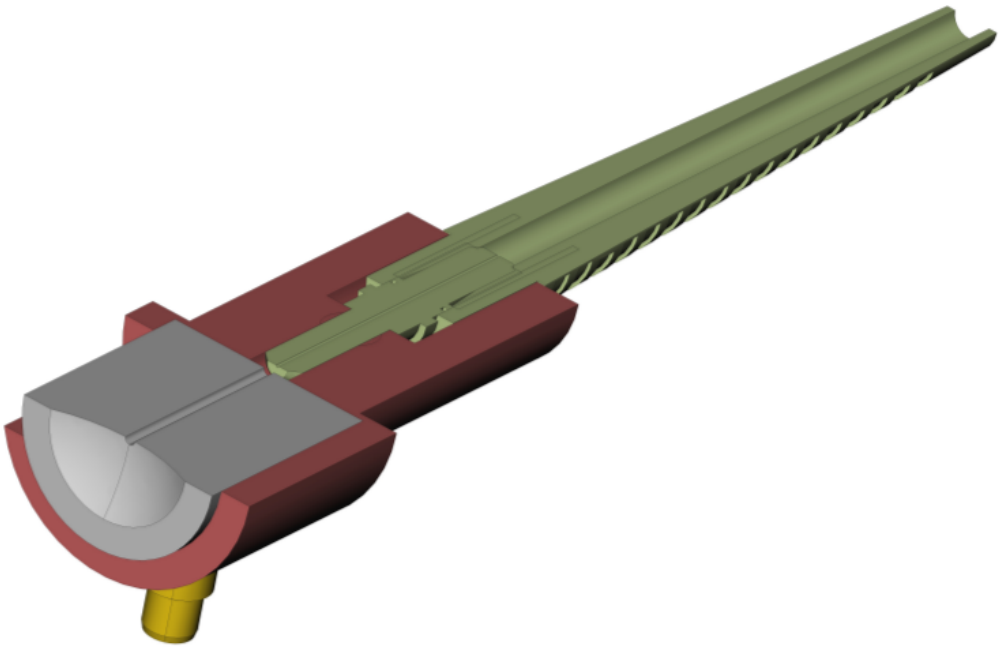

Figure 5.1: Mount [9]

The stages of the PAM scan the region of interest following a saw-tooth path. The motorized linear stage (LTM 60F, OWIS GmbH, Staufen, Germany) moves in only one direction (slow axis) on the y-axis, the high speed linear stage (M-683.2U4) controlled by Piezomotor Controller (C-867.160, Physik Instrumente GmbH & Co., Karlsruhe, Germany) continously scans on the perpendicular x-axis (fast axis), moving in both directions.

The laser distance sensor (M11L/10-10B, MEL Mikroelektronik GmbH., Eching, Germany) stands on the stages to measure x-position of the transducer in real time. The distance sensor is connected to a data acquisition card (NI PCIe-6321, National Instruments Germany GmbH., München, Germany). The stages hold the mount, which coaxially aligns the optical fiber output and the ultrasound transducer. A Gradient-Index (GRIN) lens is attached to the optical fiber output in order to provide focused illumination. The custom piezoelectric ultrasonic focused detector (Insensor, Denmark) has a central frequency of 25 MHz, an active aperture size of 11 mm and a focal length of 12 mm. The main characteristic of this transducer is the aperture in the middle of its spherical surface which allows the coaxial illumination of the sample. The GRIN lens simplifies the optical focusing on the surface without necessity of other mirrors, as its refractive index changes across its diameter, such that the light is focused directly out of the fiber tip. The lens is connected to the laser source via single mode PCF (Photonic-crystal Fiber) fiber. The laser source operates at 593 nm wavelength, as a radiation source to generate optoacoustic waves. It creates light pulses whose duration lies in the range of 10 ns, with an average per pulse energy of tens of μJ up to a maximum repetition rate of 10000 pulses per second. Small portion of the beam is delivered to a photodiode to measure the power of the pulse for further use in power correction during imaging. The optoacoustic waves are converted to analog signals in the transducer and then amplified via an IC (integrated circuit) to overcome the card's quantization noise. The amplified ultrasonic signal and the photodiode signal are then connected to another data acquisition card (M3i.4142 PCI-X, Spectrum GmbH, Grosshansdorf, Germany), which has 14-bit resolution and 250 MHz sampling rate, when two channels are active.

5.1 Signal chain

The trigger signal set by a waveform generator is connected to Spectrum card, National Instruments card and the light source. The positive edge of the pulse signal triggers the laser source, which releases a pulse of light (electromagnetic radiation). A portion of the radiation hits the photodiode and the rest travels inside a optical fiber to the mount, which consists of the detector and the beam. The radiation then exits the fiber and hits the tissue. The tissue absorbs the light and the optoacoustic phenomenon occurs. The optoacoustic wave generated due to the thermal expansion is detected by the transducer in the mount and the digitized values are visualized by a computer.

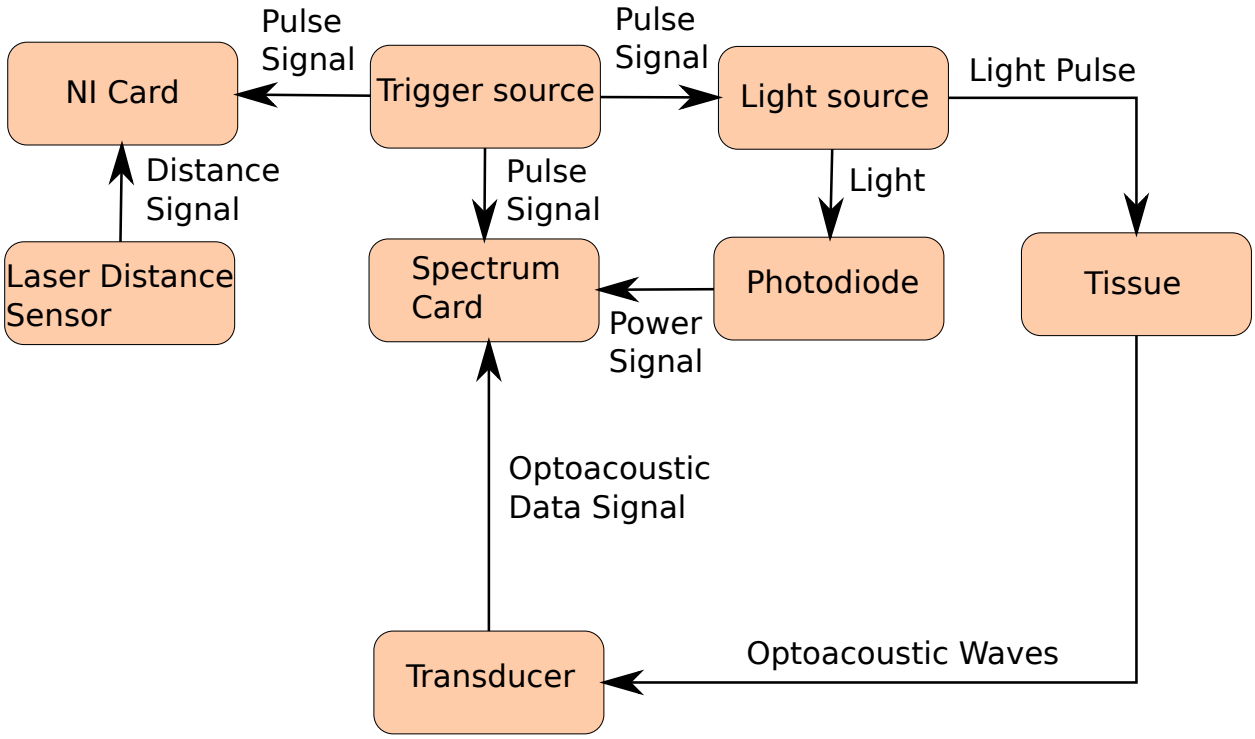

Figure 5.2: Signal chain

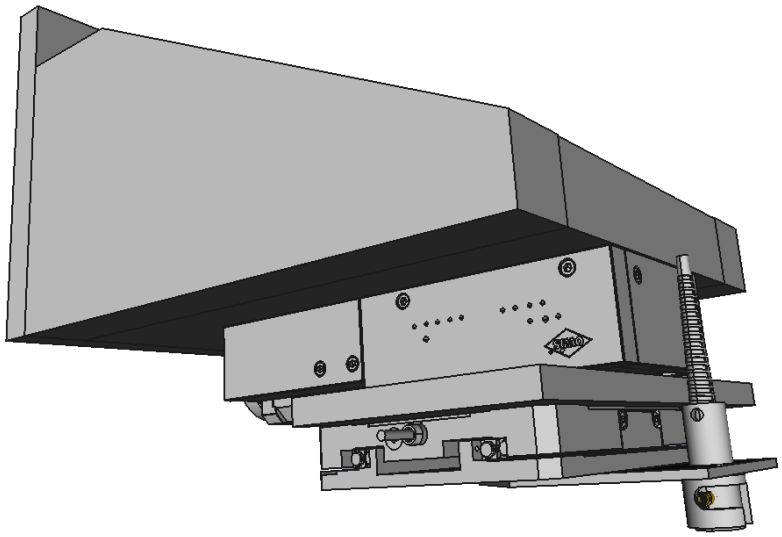

Figure 5.3: Perspective look into the stages, mount and laser distance sensor [9]

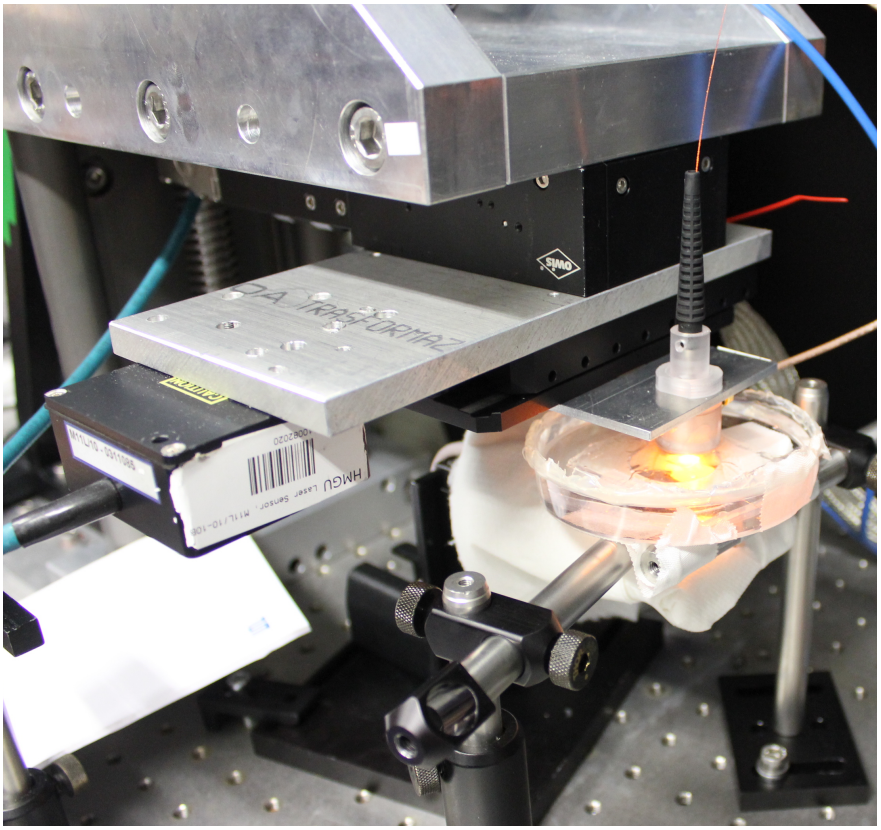

Figure 5.4: PAM Device scanning a mouse ear

6

Background for Objectives

6.1 Data Acquisition

Data Acquisition (DAQ) is a process of measuring data, such as current, voltage, resistance etc. A DAQ system consists of sensors, measurement hardware and a computer (PC) with programmable software. PC-based DAQ systems provide high processing power, productivity, display and connectivity capabilities. Sensors, also called transducers, convert physical phenomenon into electrical signal. DAQ devices make an interface between the signals and computer. They digitalize the analog signals so that computer can interprete the measured data. These devices can do signal conditioning (amplification, attenuation, filtering, isolation), analog-to-digital converting (ADC), generate pulses and automate measurement processes. The DAQ devices are connected to the PC over the bus, such as a slot or port. The bus lets the user to pass the instructions and settings to the DAQ device and measured data. Using driver software of the DAQ device simplifies connection with DAQ device by running low-level (register-level) hardware commands [10]. In our system the ultrasound detector, laser distance sensor and photodiode are the transducers; the Spectrum and National Instruments (NI) cards are DAQ devices.

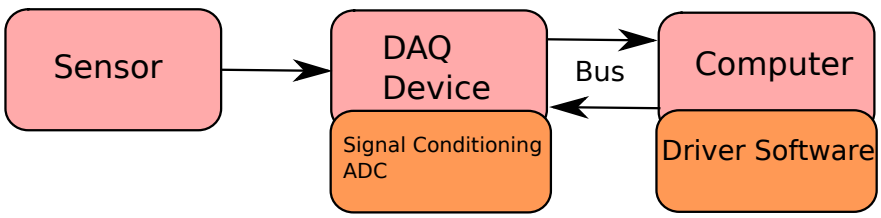

Figure 6.1: Parts of DAQ System

6.2 Digital Signal Processing

Digital Signal Processing (DSP) is the manipulation of the signal for modification, improvement or inspection. Its goal is to measure, filter and/or compress the analog signals. First steps to process the signal are sampling, then digitizing the signal by DAQ device. For sampling, Nyquist sampling theorem should be ensured to avoid aliasing. If the analog signal to be sampled is x(t)(Fourier transformed: X(f)), which contains no frequency higher than B, then it can be sampled at a series of spaced $\frac{1}{2B}$ seconds apart. In short, sampling frequency f_s can be determined using the formula $f_s \geq 2B$. The sampled signal is then quantized by ADC in DAQ device. After quantization, the data can be post-processed using a computer.

In Figure 6.2, where $X(f)$ is marked with blue strokes, it can be seen that the sampling frequency f_s is equal to $2B$, thus aliasing is avoided.

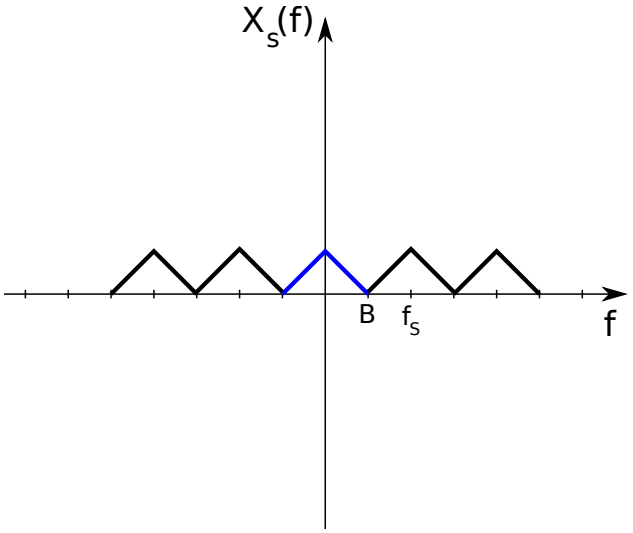

Figure 6.2: Sampling of x(t)

6.3 Real Time Computing

The relation of Real Time Computing (RTC) to our device is that the system has to acquire and visualize data in real time, which corresponds to an operation without perceivable delay. For our system, the response time is in order of microseconds to milliseconds.

7

Implementation

7.1 Physical constraints and settings of DAQ devices

7.1.1 Spectrum Card

Some values induced by the physical contraints must be determined to run the DAQ devices with proper settings. The photodiode and transducer are connected to the Spectrum card, which can achieve higher sampling rates up to 250 MHz, when both channels are active. Sampling rate of the DAQ device is determined, so that Nyquist sampling theorem is assured for the acquisition. Since the maximum frequency of the optoacoustic waves is approximated at 70 MHz, sampling rate f_s should be more than the double of the input frequency(70 MHz). In this case 250 MHz sampling rate for Spectrum is applicable.

As discussed, illuminated absorber generates ultrasound waves, which travel the acoustic focal distance and arrive at the transducer. The duration they need to arrive at the transducer can be found with $t = f_d/c$. For our setup focal distance is 12 mm and the speed of sound is approximated at 1530 ms^{-1}, thus the duration for arrival is 8 µs. To detect the optoacoustic waves generated by absorbers in deep tissue, the acquisition length is prolonged by around 4 µs. Estimating the total duration of acquisition per illumination at 12 µs, number of samples to be acquired per illumination can be found with $t_d \times f_s = 3000$ samples, where t_d is the duration. An example to the acoustic waves arriving at the transducer is shown in Figure 7.1.

Interpretation of this calculation is that 3000 samples on one channel on the Spectrum card are acquired per light pulse illuminated onto the tissue. The number of samples per illumination is defined as segment size in the code. One other aspect to calculate segment size is the input from the photodiode. To make the power correction calculations, the signal from the photodiode is required. The signal per illumination oscillates in the beginning, then starts a steady decay. The point, where it starts decaying can be referred

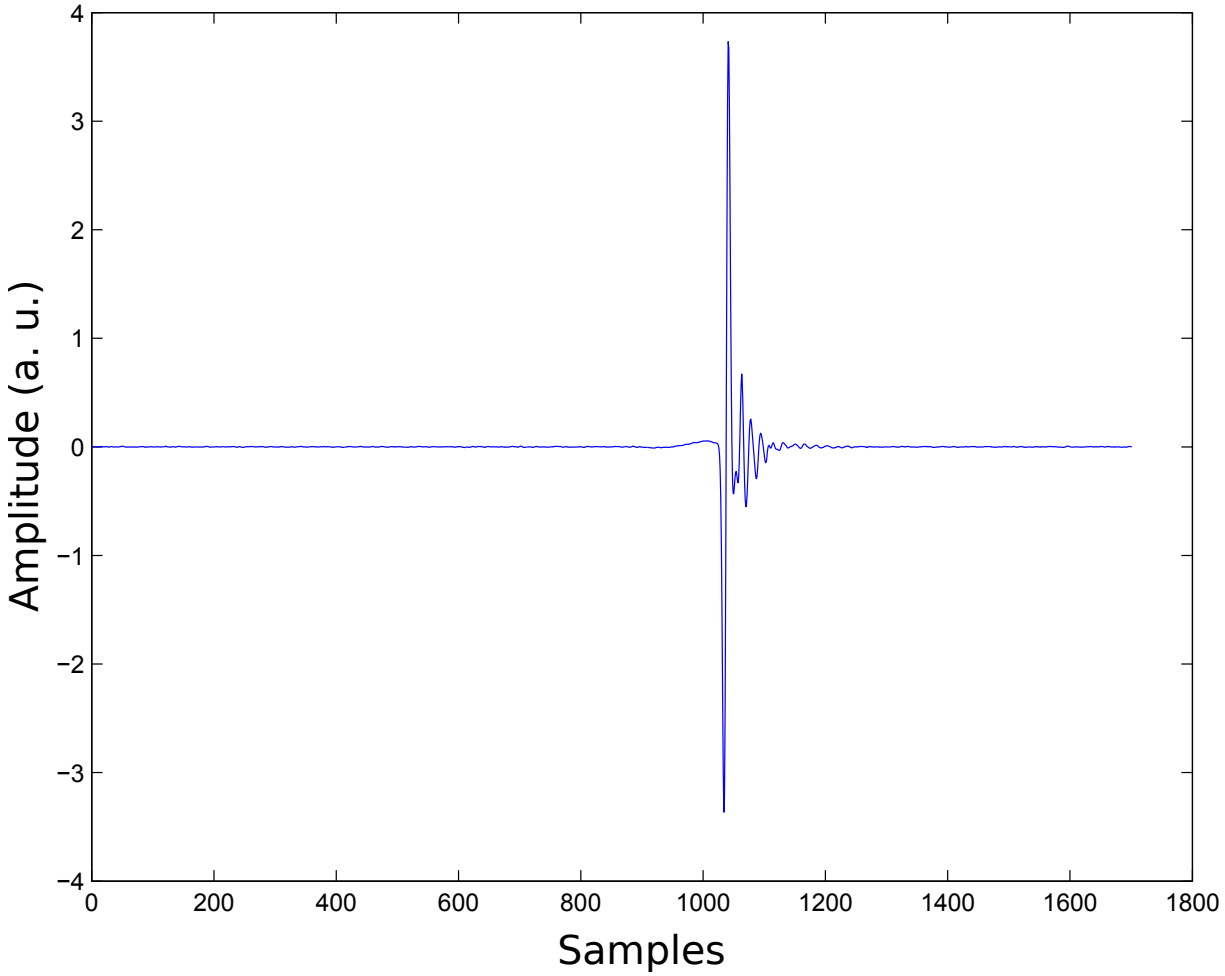

Figure 7.1: Optoacoustic Waves Detected by Transducer, segment size 1800 Samples

as the minimum number of samples to be acquired, namely the minimum segment size as shown in Figure 7.2 [11].

Both the light source and the DAQ devices are triggered by an oscillator. When the trigger signal is at the positive edge, the source creates the light pulse and releases at the negative edge of the trigger, where the width of the pulse signal is determined by the propagation delay in laser source. Also at the negative edge of trigger signal, Spectrum card acquires data of segment size. Scanning a region with PAM as the stages move requires illuminating more positions, thus acquiring more data. Pulse repetition frequency of the trigger source states how many positions are illuminated per second. The Spectrum card, thus has to be set to acquire data of segment size per each negative edge of the trigger signal, which is defined as multi-triggering [11]. The pulse repetition frequency is determined by the speed of the stages so that the resolution is assured. The amount of samples to be acquired per scan is determined by the pulse repetition frequency, segment size and region of interest.

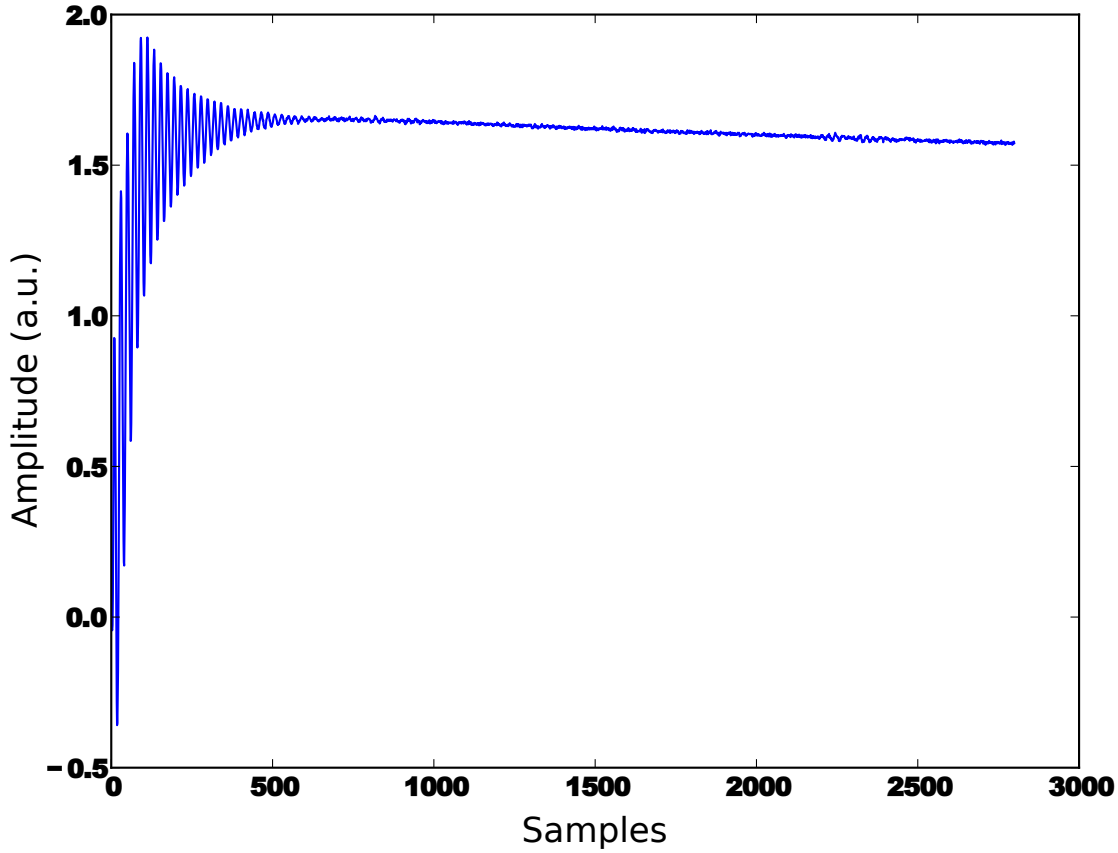

Figure 7.2: Light Power Signal on Photodiode

To avoid signal loss both ends of the signal should possess comparable impedances. The transducer has a high impedance, thus Spectrum card is set to high impedance (1 MΩ) for transducer channel. For the photodiode channel, the card is set to low impedance.

The voltage range of the signal should be set so to minimize data loss. The signal from the transducer is expected to have an amplitude of 200 mV after amplification and the photodiode 10 V.

7.1.2 NI card

NI card is synchronized with Spectrum card using the same oscillation source. The laser distance sensor is connected to the NI card, which implies the position of the stage, thus the illuminated position. Since the light source is constrained with the maximum pulse repetition frequency of 10 KHz, 25 KHz sampling rate for NI card is applicable.

The distance sensor channel is set to DC coupling and the voltage range is ±10 V.

7.2 MATLAB Executable

The objectives of the thesis should be achieved in a way that the system is compatible with MATLAB, meaning the data is manipulable and visualizable using MATLAB commands. Since the control of cards has to be in C++ programming language, an interface between MATLAB and executable files is used, called MEX(MATLAB Executable). MEX allows the user to compile the C++ code, which can contain MATLAB structures, such as array or matrix. A change in the content of conventional C++ is required to compile the MEX. The definition of `int main(void)` function has to be changed with

`void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[]).`

The inputs to MEX can be accessed with `prhs[i]` (right hand side) and outputs with `plhs[i]` (left hand side), where `i` is the index of the input. The input or output data has to be the type `mxArray`, which acts as a MATLAB structure. In order to access and manipulate the data inside `mxArray` with conventional C++ techniques, the pointer to the `mxArray` structure is obtained with `mxGetPr()` command. For example, to change the tenth element of the third input to -1:

```
mxArray* input;
input = (mxArray*) prhs[2];
double* inp;
inp = mxGetPr(input);
inp[9] = -1;
```

Matrix structures can be created with `mxCreateDoubleMatrix(M,N,mxREAL)` where `M` and `N` are the dimensions of the matrix. An example would be:

```
double* gs;
mxArray* galatasaray;
galatasaray = mxCreateDoubleMatrix(10,1,mxREAL);
gs = mxGetPr(galatasaray);
```

The content of `galatasaray` matrix can be changed with `gs[i]` convention. To create an output matrix, `galatasaray` should be replaced with `plhs[i]`, where `i` is the index of output.

The prepared code is compiled to a MEX file in MATLAB command-line using:

```
mex -Ifiles -Lfiles -lNIDAQmx -Ispectrum -Lspectrum -lspectrum_comp -lspectrum
-lSpcStdNT -lspcm_win64_msvcpp -lspcm_win32_msvcpp -lspcm_win32_cvi file.cpp
```

where `file.cpp` is the file, which contains the code, `-I` the header folder and `-L` the library folder . `-l` parameter adds the proper library files to compilation.

7.3 Calling Visualization Scripts

The structures inside MEX cannot be used directly by MATLAB scripts until execution of MEX ends. A strategy has to be developed, which will enable to use MATLAB visualization commands, such as `plot()` or `imagesc()` for inner data in MEX. In order to overcome this issue, `mexCallMATLAB()` function is used, which allows to call MATLAB scripts inside MEX. Prior to execution, when a MATLAB script is indicated as an input to MEX, the script can be used to manipulate and visualize the data in an other new instance. The input script and inputs to that script are executed simply calling the `feval()` function of MATLAB from inside MEX. `feval()` function evaluates the inputs, which are strings, and executes them as MATLAB commands. The figure of visualization has to be updated each time with MATLAB's `drawnow` command. An example to a function for calling MATLAB Scripts would be:

```
void CallImageSc(mxArray* plotfcn,mxArray *input0){
    mxArray *ppFevalRhs[2] = {plotfcn, input0};
    mexCallMATLAB(0, NULL, 2, ppFevalRhs, "feval");
    mexCallMATLAB(0, NULL, 0, NULL, "drawnow");
}
```

The created `CallImageSc()` function takes as input the name of the script, namely `plotfcn`, which is an input to the MEX and the input to that MATLAB script, namely `input0`, which can be any of the matrix structures created inside MEX.

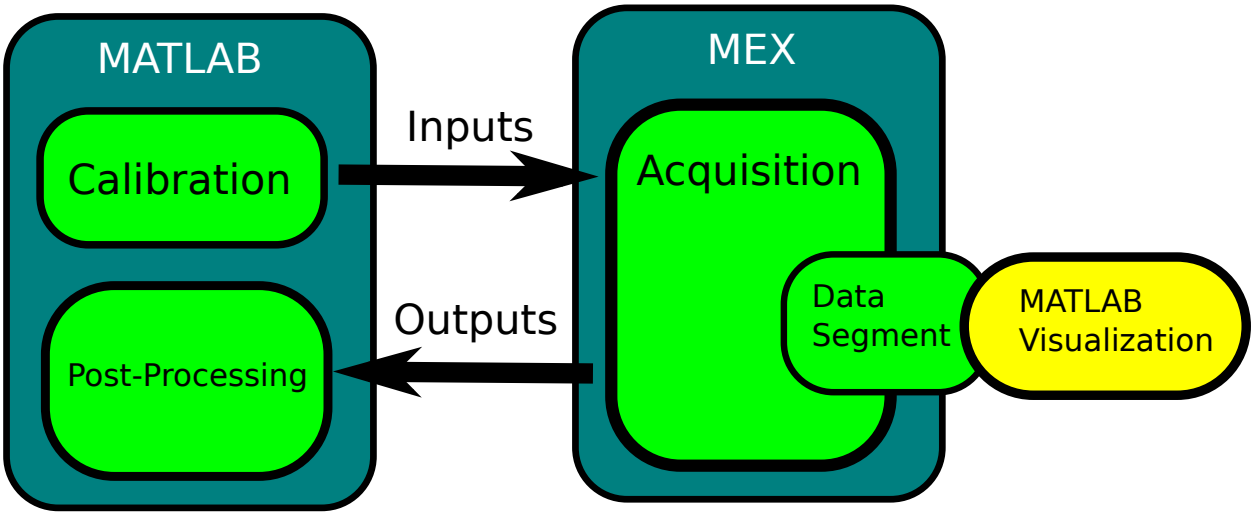

Figure 7.3: MATLAB, MEX and MATLAB Visualization

7.4 Creating Threads

The other issue relies on the driver software of the Spectrum card, which does not support to start the acquisition as a task, which would enable to acquire data and run callbacks in the background so that the visualization of the data is in real time. The driver software is designed so that the user will not be able start another command until the acquisition ends. In order to resolve this, the acquisition of the Spectrum card is initiated as another instance, namely a thread, handled by the operating system of the computer. The functions executed by the threads run in the background so that the thread will allow to continue in the code and run the command in the next line. An example to this situation would be:

```
hThread = (HANDLE)_beginthreadex( NULL, 0, &ThreadFunc, NULL, 0, &threadID );
nicard.acquire();
```

In this example, `hThread` is a handle, which starts a new thread and runs `ThreadFunc`, which contains the command to start acquisition for Spectrum card. Using this handle, `nicard.acquire()` command will not wait till the acquisition of the Spectrum card ends.

When running threads, critical sections should be paid attention to avoid segmentation faults, which may arise due to access of different processes to the same section of the memory at the same time. In order to avoid segmentation faults, convenient functions should be used to stop, pause or wait for running threads.

In the example of Figure 7.4, `funcA()`, `funcB()`, `funcC()` and `funcD()` are defined to run one interval long, where `funcThread()` one and a half as a thread. `funcThread()` does not avoid `funcC()` or `funcD()` from starting and is free of them, as long as it does not access the same section of memory.

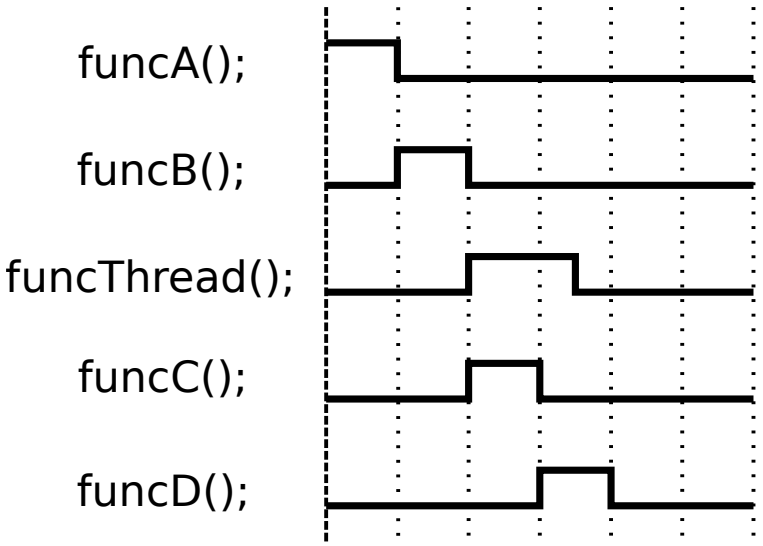

Figure 7.4: Example for running a thread

7.5 Data Handling

The data acquired on the card's memory should be transferred to the computer memory as soon as possible for real time visualization, thus both of them operate with First In First Out (FIFO) method. As shown in Figure 7.5, Spectrum card and NI card handle the acquired data in a similar way.

The Spectrum card stores the data in its own buffer and creates chunk in the computer memory the same size as the card buffer. The chunk in computer memory is updated from the card buffer,when the data is acquired, which is of a notify size. When the chunk fills up its size, it starts updating from the beginning of the chunk as if in a loop. Thus the data should be collected from the chunk at each notification event and stored in a different structure, such as an array or matrix. NI card stores data in its buffer, which is then copied to a chunk in the computer memory and stored in an array or matrix.

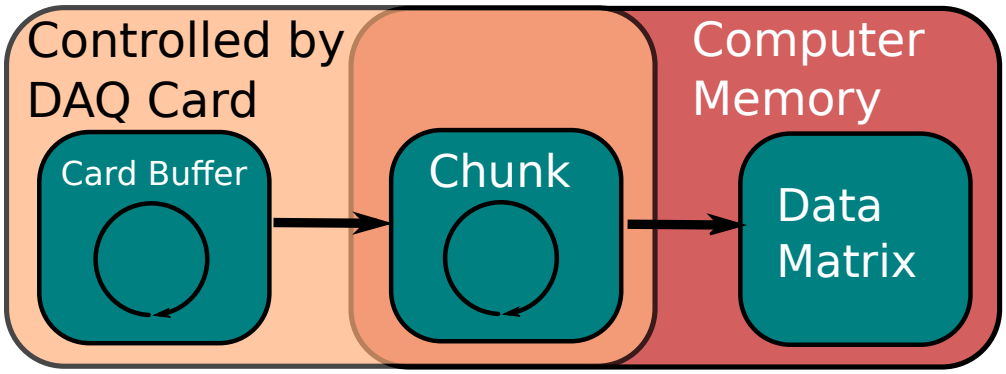

Figure 7.5: Card Buffer, Chunk and Memory Relationship

7.6 Scheduling

One other issue is to plan the scheduling so that the transfer operation of chunks will not be started till the particular acquisition is done. Otherwise the power corrected data, which is instantaneously created during acquisition, will have unexpected zero values, since MATLAB initiates its structures with zero. This will cause loss of data and distortions in the visualization.

Since the duration of the acquisition is determined by the size of the acquisition data, the acquisition thread should run until the conditions are fulfilled, which indicate the end of the acquisition. In order to ensure this, the end of the process in the thread is waited for infitinely long so that the acquisition of whole data is guarenteed, which is controlled by the `WaitForSingleObject(hThread, INFINITE)` command. The wait command avoids the closing of the DAQ cards, which will double ensure the end of acquisition.

A brief look into the scheduling of the application is shown in Figure 7.6.

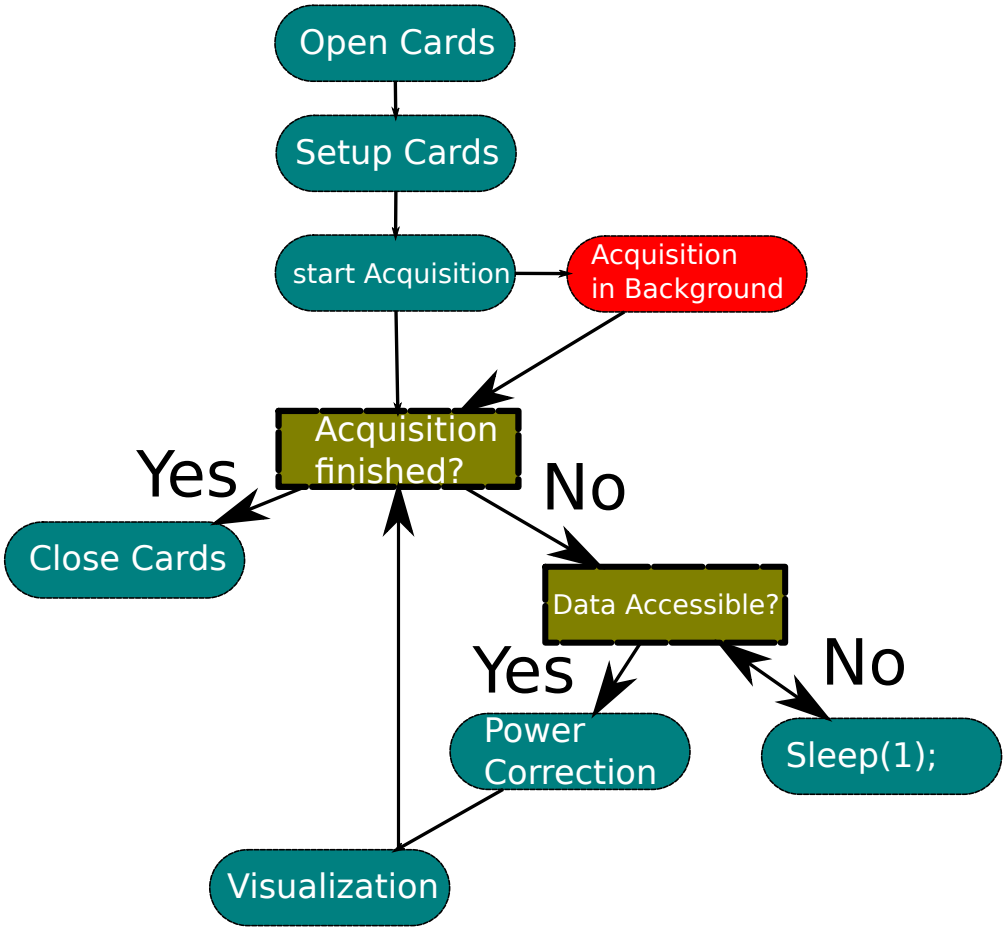

Figure 7.6: Scheduling of the application

7.7 Explanation of Code

The code is so designed that the acquisition cards are represented as objects, which own their settings, acquired data and functions(attributes), such as `open()`, `close()`, `acquire()`, `clean()` or `setup()`. The idea behind the design is to be able to control settings easier and differentiate the data from each channel to avoid confusions.

NI card has the object name NICard, where Spectrum card AQCard. The objects are called instances, when the structure is initialized. For example, in the code the Spectrum card is opened with `card.open()` command and the NI card with `nicard.open()`.

Both of the DAQ devices have to be opened first to be available for usage and then settings should be done with setup attribute. NICard has an easy structure to set and acquire. A task with relevant settings is created where channels, voltage range, trigger settings and callback are defined in `setup()`. To start acquisition with NICard, the task defined in `setup()` is started with `acquire()` attribute. However, AQCard requires more detailed setup using the driver software's function `spcm_dwSetParam_i32(card,`

`register, value)` to set required registers, instead of having seperate functions as in NICard.

NICard handles acquisition in the background on its own without the necessity of threads, while it keeps the acquired data in `nicard.chni[]` array. Memory allocation for the chunk of AQCard is done using `pvAllocMemPageAligned(size)` and release with `vFreeMemPageAligned(data, size)`. Acquisition with AQCard is controlled with `acquire()` attribute. In short, `acquire()`, which is run inside a thread, keeps track of acquired data so far and acquires data until a certain amount is fulfilled. During acquisition, it keeps track of a few counters, such as `notifyCounter`, `flagAcquire` etc. to transfer the data from the chunk to the proper index in the `card.channel[]` array, which is supposed to contain all of the acquired data of channel.

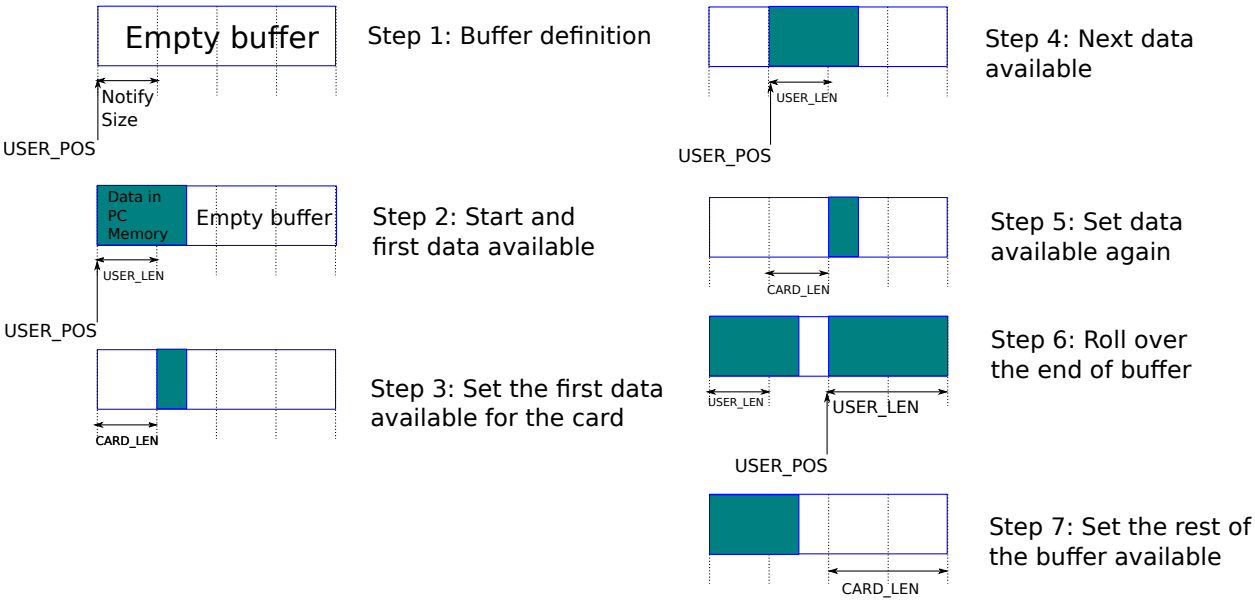

Figure 7.7: Buffer Handling in Spectrum Card [11]

An example of card buffer handling of Spectrum card is shown in Figure 7.7, where the buffer consists of four notify sizes. `USER_LEN` register contains the currently available number of bytes that are free to write new data to the card. The user can fill this amount of bytes with new data to be transferred. After filling an amount of the buffer with new data to transfer to card, the user tells the driver with `CARD_LEN` register that the amount of data is ready to transfer.

Before finishing the execution of the application, DAQ devices have to be closed with `close()` attributes and the allocated memory has to be released with `cleanMem()` and `Cleanup()`.

A calibration has to be done before starting the real measurement, which delivers the inputs to the application, that contain parameters for the card and power correction, such as total samples to acquire per channel, postsamples and segment size, sampling

rate, sensitivity of channels, region of interest, position of y-axis stage, whose position is predicted by the calibration beforehand, and the script's name which will be called for visualization.

The refresh rate of power correction, which is updated once per segment size and visualization, which is updated once per window size, are easy to setup, since the total samples to be acquired per channel and acquired so far are known via `flagAcquire` and `totalSamples`. `Sleep()` function and `flagAcquire` are used accordingly to avoid processing and visualization before acquisition of the data.

The data of the channels are defined as outputs of the application, which are to be used in further operations in MATLAB.

8

Results

In the previous optoacoustic microscope setup, the measurement had to be done choosing one of two options: either with fast scan without power correction and visualization or slow scan with power correction and position prediction. Since power correction is significant for visualization, slow scan was usually preferred. The accomplished objectives of this thesis overcome issues of both options, using two DAQ devices and enabling real time processing.

Using the slow scan method of the previous setup, the duration of the measurement takes roughly 45 minutes, which assumes that the region of interest and the focal distance are set properly and precisely. The duration of the measurement is crucial, since the tissue, namely the mouse, has to stay anaesthetised during the measurement. Prolonged measurements will affect the results of the measurement and may even kill the mouse. In order to obtain better results, duration of the measurement must be shortened, which is accomplished with real time acquisition. Furthermore, real time acquisition and fast scan provides the opportunity to monitor temporal changes in the tissue.

In order to obtain good results from the measurement, the power of the laser and the focal distance of the transducer should be determined precisely, which are quite laborious operations without real time feedback. The divergence of the focus is so high that a very little change in focal distance may significantly influence the results of the measurement. Without real time feedback, it may take tens of minutes to find the focal distance for the region of interest, whereas it takes only a few seconds to find the focal distance with real time feedback. The same issue also applies for power measurement and setting of the light source.

In Figure 8.1, mouse ear was imaged using new setup of PAM. The acquisition of the region of interest would have taken around 3 hours with the slow scan, whereas it took only 37 seconds with the current one using laser power correction. Measurement duration is solely down to the scanning duration, since the acquisition is in real time.

The real time feedback has also been proved to be crucial in steps previous to the actual

measurement such as the fine alignment of the optical fiber. Using moving average, the signal of the photodiode is displayed on the screen, while the alignment to optimize the coupling between the free laser beam and the optical fiber is performed.

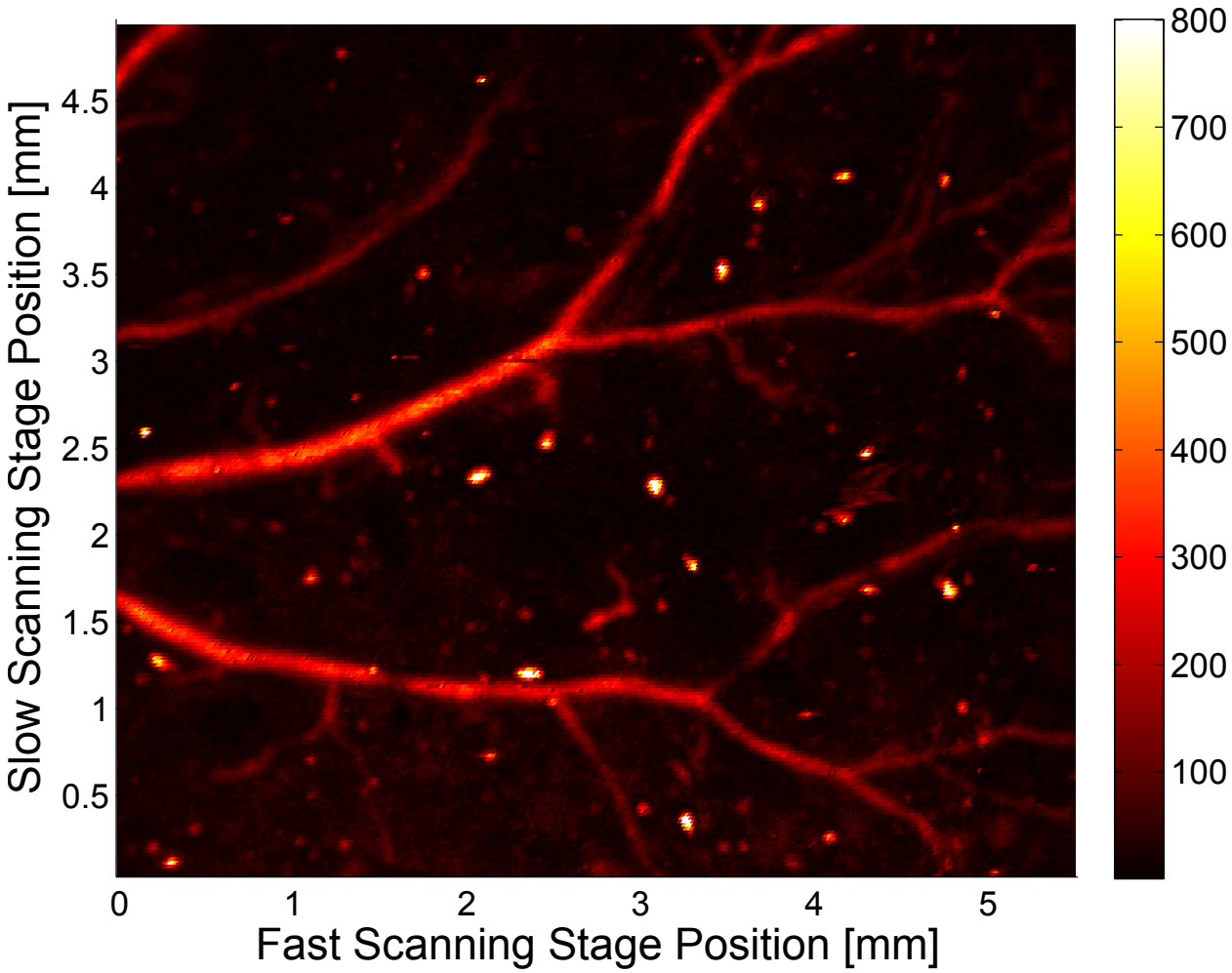

Figure 8.1: In vivo image of a mouse ear vasculature using maximum amplitude projection (color scale in arbitrary units)

9

Discussion

Signal propagation delay occurs as the signals flow, which is expected. Acquisition delay is neglectable and acquisition can be considered a real time operation. However, the delay in visualization depends on the used function. For example `plot()` causes a delay in visualization, since this function runs slow with many samples. However, `imagesc()`, which creates a color map of data, can run many times faster and avoid delays in visualization. In order to visualize in real time, it should be paid attention to how many samples per unit time the visualization function can process, which will also determine the refresh rate of the visualization. Even though the visualization may cause delays, the acquisition in real time will not be influenced at all.

The amount of samples to be acquired affects the type of variables in C++ code. It is significant to know whether the amount is representable in a certain type and ensure that the maximum number is not exceeded. In order to overcome this issue, `unsigned long long int` type, whose maximum is 18446744073709551615, is used to contain big values, such as totalSamples, flagAcquire etc. Furthermore, the matrix created in MATLAB cannot exceed a certain size. The maximum size of matrix in MATLAB, which increases with newer versions, should be known in order to avoid crashes. The maximum size depends also on the operating system and configuration of computer. The maximum number of elements in largest real double array is $2^{48} - 1$ for a computer with the configuration 64-bit Windows XP, MATLAB 7.4 and later.

Since the data in matrix is updated at each notification, notify size of Spectrum card, which has to be a multiple of page size: 4 KB, should be so determined that the duration of data transfer from the chunk to the matrix will not exceed the sampling duration, otherwise data loss will occur.

The outputs of the application have the type `mxArray`, which occupies five to six times more space than the size of data to be acquired. Thus, empty space in the computer memory should be ensured to avoid buffer overrun.

The Spectrum card's driver software misses the features for starting a task in the

background and copying the acquired data to an array. The card did not allow to acquire more than the size of the buffer, and thus, technical support was requested via calling the Spectrum Systementwicklung Microelectronic GmbH. As a conclusion, the card does not support starting a task in the background and the issue was resolved using strategies developed on my own, such as starting threads, decreasing the size of the buffer and using chunks in the computer memory to copy the notification data, which is then transfered to an array in the computer memory.

10

Conclusion

In this work, real time processing was accomplished in order to shorten the measurement duration and visualize the data acquired by two data acquisition devices. Scheduling and data handling strategies were implemented in C++ in order to process data in computer memory instead of storing the data in harddisk, which provided fast scanning.

As C++ is a lower level language compared to MATLAB, it enables to initialize threads, which run in the background so that each procedure in the code does not have to wait until the prior initiated one ends. Threads and synchronization signals would keep the two data acquisition cards synchronized in the background.

This strategy allowed MATLAB functions and scripts to be called from inside the MEX instance to manipulate and visualize the data in real time. Thus, operations which requires real time feedback, such as focusing and power measuring were implemented and could be easily extended. Making use of the MEX environment for visualization of the data segment in an independent MATLAB instance also keeps the acquired data in the computer memory to be post-processed in MATLAB after the end of the acquisition.

In addition, a seperation of the calibration and stage control and an integration of the stage control solely into MEX avoided synchronization issues of the data acquisition cards.

This application not only provides real time acquisition for fast scanning, but also makes it possible to execute operations prior to measurement, where real time feedback is a necessity, such as focusing and power meter.

10.1 Possible Improvements

- Benefiting from external visualization applications which can be called directly from operating system instead of calling MATLAB scripts.
- Controlling the oscillator using software or using NI card as trigger source would help to automate the measurement.

- An emergency exit button should be designed and integrated into the system, since moving parts exist.
- The optoacoustic waves generated by a certain position may influence the positions nearby during scan. The algorithm to detect these errors can be developed to be executed in GPU (Graphical Processing Unit) instead of CPU(Central Processing Unit), which can run multiple independent calculations simultaneously, since calculation for each position is independent of the other one.

Appendix A

syncCountdown.cpp

```
#include <stdio.h>
#include <stdlib.h>
#include <mex.h> // for MATLAB mex file
#include <matrix.h>

#include <conio.h>
#include <process.h> // threading
#include <windows.h> // threading

// -----card driver includes -----
#include "dlltyp.h" // 1st include
#include "regs.h" // 2nd include
#include "spcerr.h" // 3rd include
#include "spcm_drv.h" // 4th include

#include "NIDAQmx.h"
//#include <NIDAQmx.h>
//#define DAQmxErrChk(functionCall) if( DAQmxFailed(error=(functionCall)) ) goto
    Error; else

// classes example
#include <iostream>
#include <fstream>
using namespace std;
int32 CVICALLBACK EveryNSamplesCallback(TaskHandle taskHandle, int32
    everyNsamplesEventType, uInt32 nSamples, void *callbackData);

void CallImageSc(mxArray* plotfcn,mxArray *input0){
    mxArray *ppFevalRhs[2] = {plotfcn, input0};
```

```
    mexCallMATLAB(0, NULL, 2, ppFevalRhs, "feval");
    mexCallMATLAB(0, NULL, 0, NULL, "drawnow");
}

void CallPlot(mxArray* plotfcn,mxArray *input0, mxArray *input1, mxArray
    *input2){
    mxArray *ppFevalRhs[4] = {plotfcn, input0, input1,input2};
    mexCallMATLAB(0, NULL, 4, ppFevalRhs, "feval");
    mexCallMATLAB(0, NULL, 0, NULL, "drawnow");
}

void CallManipulate(mxArray* manipulatefcn, mxArray *maniParams, mxArray *A,
    mxArray *ROI, mxArray *x, mxArray *y, mxArray *p){
    mxArray *ppFevalRhs[7] = {manipulatefcn, maniParams, A, ROI, x,y,p};
    mexCallMATLAB(0, NULL, 7, ppFevalRhs, "feval");
    mexCallMATLAB(0, NULL, 0, NULL, "drawnow");
}

void CallPlotCorrected(mxArray* plotfcn,mxArray *maniParams, mxArray *x, mxArray
    *y, mxArray *p){
    mxArray *ppFevalRhs[5] = {plotfcn, maniParams, x, y, p};
    mexCallMATLAB(0, NULL, 5, ppFevalRhs, "feval");
    mexCallMATLAB(0, NULL, 0, NULL, "drawnow");
}

void StageMove(mxArray* stagefcn, mxArray* ROI, mxArray* stageParams){
    mxArray *ppFevalRhs[3] = {stagefcn, ROI, stageParams};
    mexCallMATLAB(0, NULL, 3, ppFevalRhs, "feval");
}


void* pvAllocMemPageAligned (uint64 qwBytes)
    {
    // for unknown reasons VirtualAlloc/VirtualFree leaks memory if qwBytes <
        4096 (page size)
    // therefore use _aligned_malloc () to get small amounts of page aligned
        memory
    if (qwBytes >= 4096)
        return VirtualAlloc (NULL, (size_t) qwBytes, MEM_RESERVE | MEM_COMMIT,
            PAGE_READWRITE);
    else
        {
```

```
        void* pvMem = _aligned_malloc ((size_t) qwBytes, 4096);
        memset (pvMem, 0, (size_t) qwBytes);
        return pvMem;
        }
    }

// ***************************************************************************

void vFreeMemPageAligned (void* pvMemory, uint64 qwBytes)
    {
    // for unknown reasons VirtualAlloc/VirtualFree leaks memory if qwBytes <
        4096 (page size)
    // therefore use _aligned_malloc () to get small amounts of page aligned
        memory
    if (qwBytes >= 4096)
        VirtualFree (pvMemory, 0, MEM_RELEASE);
    else
        _aligned_free (pvMemory);
    }

int bKbhit(void)
    {
    return _kbhit();
    }

// ***************************************************************************

int cGetch()
    {
    return _getch();
    }

char* szTypeToName (int32 lCardType)
    {
    static char szName[50];
    switch (lCardType & TYP_SERIESMASK)
        {
        case TYP_M2ISERIES:  sprintf (szName, "M2i.%04x", lCardType &
            TYP_VERSIONMASK); break;
        case TYP_M2IEXPSERIES: sprintf (szName, "M2i.%04x-Exp", lCardType &
            TYP_VERSIONMASK); break;
```

```
        case TYP_M3ISERIES:  sprintf (szName, "M3i.%04x", lCardType &
            TYP_VERSIONMASK); break;
        case TYP_M3IEXPSERIES: sprintf (szName, "M3i.%04x-Exp", lCardType &
            TYP_VERSIONMASK); break;
        default:             sprintf (szName, "unknown type");
                                 break;
        }
    return szName;
    }


class NICard {
public:
        int32     error;
    TaskHandle taskHandle;
    int32     read;
    int32     rate;
    // int32     samples;
    uint64_li  samples;
    double*    chni;
    float64    data[1];
        char      errBuff[2048];

    int quantity;
    int counter;
    void setup();
    void acquire();
    void Cleanup(void);
    void stop();

} nicard;

void NICard::setup(){
    /*********************************************/
    // DAQmx Configure Code
    /*********************************************/
    DAQmxCreateTask("",&taskHandle);
    DAQmxCreateAIVoltageChan(taskHandle,"Dev1/ai5","",DAQmx_Val_Cfg_Default,-10.0,10.0,DAQmx_Val
    DAQmxSetAITermCfg(taskHandle,"Dev1/ai5", DAQmx_Val_RSE);
    DAQmxCfgSampClkTiming(taskHandle,"/Dev1/PFI0",rate,DAQmx_Val_Falling,DAQmx_Val_FiniteSamps,s
        // first rate, then samples.
    DAQmxRegisterEveryNSamplesEvent(taskHandle,DAQmx_Val_Acquired_Into_Buffer,1,0,EveryNSamplesC
```

```
}

void NICard::acquire(){
    DAQmxStartTask(taskHandle);
    return;
}

void NICard::stop(){
    DAQmxStopTask(taskHandle);
    Cleanup();

    printf("\nAcquired %d total samples.\n",quantity);
    return;
}

int32 CVICALLBACK EveryNSamplesCallback(TaskHandle taskHandle, int32
    everyNsamplesEventType, uInt32 nSamples, void *callbackData)
{
    int32      error=0;
    int32      read;
    char       errBuff[2048]={'\0'};

    /*********************************************/
    // DAQmx Read Code
    /*********************************************/
    DAQmxReadAnalogF64(taskHandle,1,10.0,DAQmx_Val_GroupByScanNumber,nicard.data,1,&read,NU

    for(int i=0; i<1;i++){
        // Sleep(0);
        nicard.chni[nicard.counter++] = nicard.data[i];
        //printf("%f\n",data[i]);
    }
    nicard.quantity+=read;

    // if( read>0 ) {
    //  printf("Acquired %d samples. Total %d\r",read,totalRead+=read);
    //  fflush(stdout);
    // }

    return 0;
}
```

```
void NICard::Cleanup (void)
{
    if( taskHandle!=0 )
    {
        DAQmxStopTask(taskHandle);
        DAQmxClearTask(taskHandle);
        taskHandle = 0;
    }
}

class AQCard {
public:
        drv_handle hCard;
    int32     lCardType, lSerialNumber, lFncType;
    int16*    pnData;
    char      szErrorTextBuffer[ERRORTEXTLEN];
    uint32    dwError;
    int32     lStatus, lAvailUser, lPCPos;
    // int32     totalSamples;
    int32     postSamples,segmentSize,samplingRate, channel1Sens;
        // int32             flagAcquire;
    double* ch0;
    double* ch1;
    double* ch1Corrected;
    // uint64    qwTotalMem = 0;
    // uint64    qwToTransfer = MEGA_B(2);

    // // settings for the FIFO mode buffer handling
    // int32     lBufferSize = MEGA_B(4);
    // int32     lNotifySize = KILO_B(16);

    // uint64    qwTotalMem;
    // uint64    qwToTransfer;
    uint64_li qwTotalMem;
    uint64_li qwToTransfer;
    uint64_li totalSamples;
    uint64_li flagAcquire;

    // settings for the FIFO mode buffer handling
    int32     lBufferSize;
    int32     lNotifySize;
```

```
        void open();
        void setup();
        void acquire();
        void close();
    void cleanMem();
        void transfer();

} card;

void AQCard::open(){
        hCard = spcm_hOpen ("0");
    if (!hCard){
        printf ("no card found...\n");
        return;
    }
    spcm_dwGetParam_i32 (hCard, SPC_PCITYP,  &lCardType);
    spcm_dwGetParam_i32 (hCard, SPC_PCISERIALNO, &lSerialNumber);
    spcm_dwGetParam_i32 (hCard, SPC_FNCTYPE, &lFncType);

    switch (lFncType){
        case SPCM_TYPE_AI:
            printf ("Found: %s sn %05d\n", szTypeToName (lCardType),
                lSerialNumber);
            break;

        default:
            printf ("Card: %s sn %05d not supported by examplee\n", szTypeToName
                (lCardType), lSerialNumber);
            return;
    }
}

void AQCard::setup(){
    // int32 reg;
    // spcm_dwGetParam_i32(hCard, SPC_TRIG_CH_ORMASK0, &reg);
    // printf("register is:%d\n", reg);

    // trigger settings
    spcm_dwSetParam_i32 (hCard, SPC_CHENABLE, 3);                 // just 1
        channel enabled, 3 for both channels.
```

```
// spcm_dwSetParam_i32 (hCard, SPC_POSTTRIGGER, 3000);           // 1k of
   pretrigger data at start of FIFO mode
// spcm_dwSetParam_i32 (hCard, SPC_SEGMENTSIZE, 3008);           // 1k of
   pretrigger data at start of FIFO mode
spcm_dwSetParam_i32 (hCard, SPC_POSTTRIGGER, postSamples);           // 1k of
   pretrigger data at start of FIFO mode
spcm_dwSetParam_i32 (hCard, SPC_SEGMENTSIZE, segmentSize);           // 1k of
   pretrigger data at start of FIFO mode
spcm_dwSetParam_i32 (hCard, SPC_CARDMODE, SPC_REC_FIFO_MULTI); // FIFO mode
spcm_dwSetParam_i32 (hCard, SPC_TIMEOUT, 10000);               // timeout 10 s
// spcm_dwGetParam_i32(hCard, SPC_PRETRIGGER, &reg);
// printf("register is:%d\n", reg);


    ///
spcm_dwSetParam_i32(hCard, SPCM_X0_MODE, SPCM_XMODE_TRIGIN);
// spcm_dwSetParam_i32(hCard, SPC_TRIG_EXT0_PULSEWIDTH, 0);
// spcm_dwSetParam_i32(hCard, SPC_TRIG_OUTPUT, 0);

spcm_dwSetParam_i32(hCard, SPC_TRIG_ORMASK, SPC_TMASK_EXT1);
spcm_dwSetParam_i32(hCard, SPC_TRIG_EXT1_MODE, SPC_TM_NEG);
///

// spcm_dwSetParam_i32(hCard, SPC_TRIG_EXT0_LEVEL0, 800);
// spcm_dwSetParam_i32(hCard, SPC_TRIG_EXT0_LEVEL1, 1500);

spcm_dwSetParam_i32 (hCard, SPC_SAMPLERATE, samplingRate);

spcm_dwSetParam_i32 (hCard, SPC_CLOCKOUT, 0);                 // no clock
   output

// channel settings
spcm_dwSetParam_i32(hCard, SPC_PATH0, 0);
spcm_dwSetParam_i32(hCard, SPC_AMP0, 10000);
spcm_dwSetParam_i32(hCard, SPC_PATH1, 0);
// spcm_dwSetParam_i32(hCard, SPC_AMP1, 200);
spcm_dwSetParam_i32(hCard, SPC_AMP1, channel1Sens);

spcm_dwSetParam_i32(hCard, SPC_50OHM0, 0);
spcm_dwSetParam_i32(hCard, SPC_ACDC0, 0);
spcm_dwSetParam_i32(hCard, SPC_FILTER0, 1);
spcm_dwSetParam_i32(hCard, SPC_DIFF0, 0);
```

```
    spcm_dwSetParam_i32(hCard, SPC_50OHM1, 1);
    spcm_dwSetParam_i32(hCard, SPC_ACDC1, 0);
    spcm_dwSetParam_i32(hCard, SPC_FILTER1, 1);
    spcm_dwSetParam_i32(hCard, SPC_DIFF1, 0);
    //

    // define the data buffer
    pnData = (int16*) pvAllocMemPageAligned ((uint64) lBufferSize);
    if (!pnData){
        printf ("memory allocation failed\n");
        spcm_vClose (hCard);
        return;
    }
    spcm_dwDefTransfer_i64 (hCard, SPCM_BUF_DATA, SPCM_DIR_CARDTOPC, lNotifySize,
        pnData, 0, lBufferSize);
    // spcm_dwSetParam_i32(hCard, SPC_M2CMD, M2CMD_CARD_WRITESETUP);
}

void AQCard::acquire(){
    // start everything
    dwError = spcm_dwSetParam_i32 (hCard, SPC_M2CMD, M2CMD_CARD_START |
        M2CMD_CARD_ENABLETRIGGER | M2CMD_DATA_STARTDMA);
    // dwError = spcm_dwSetParam_i32 (hCard, SPC_M2CMD, M2CMD_CARD_START |
        M2CMD_CARD_ENABLETRIGGER);
    // dwError = spcm_dwSetParam_i32 (hCard, SPC_M2CMD, M2CMD_DATA_WAITDMA);
    // check for error
    if (dwError != ERR_OK){
        spcm_dwGetErrorInfo_i32 (hCard, NULL, NULL, szErrorTextBuffer);
        printf ("%s\n", szErrorTextBuffer);
        vFreeMemPageAligned (pnData, (uint64) lBufferSize);
        spcm_vClose (hCard);
        return;
    }

    // run the FIFO mode and loop through the data
    else {
        int32 samplesPerBuffer = lBufferSize/2;
               int32 samplesPerNotify = lNotifySize/2;
        int32 notifyPerBuffer = samplesPerBuffer / samplesPerNotify;
        int32 notifyCounter = 0;
        int32 bufferCounter = 0;
```

```
while (qwTotalMem < qwToTransfer){
    if (dwError = spcm_dwSetParam_i32 (hCard, SPC_M2CMD,
        M2CMD_DATA_WAITDMA) != ERR_OK){
        if (dwError == ERR_TIMEOUT)
            printf ("... Timeout\n");
        else
            printf ("... Error: %d\n", dwError);
            spcm_dwGetErrorInfo_i32 (hCard, NULL, NULL, szErrorTextBuffer);
            printf ("%s\n", szErrorTextBuffer);
        break;
    }

    else {
        spcm_dwGetParam_i32 (hCard, SPC_M2STATUS,      &lStatus);
        spcm_dwGetParam_i32 (hCard, SPC_DATA_AVAIL_USER_LEN, &lAvailUser);
        spcm_dwGetParam_i32 (hCard, SPC_DATA_AVAIL_USER_POS, &lPCPos);

        if (lAvailUser >= lNotifySize){
            qwTotalMem += lNotifySize;

            uint64_li beginning = flagAcquire/2;
            int current = notifyCounter*samplesPerNotify;
            for(int i=0; i<samplesPerNotify/2;i++){
                //printf("%d\n",card.ch0[beginning+i]);
                //printf("%d\n",card.pnData[notifyCounter*samplesPerNotify+i*2]);
                card.ch0[beginning+i] = card.pnData[current+i*2];
                card.ch1[beginning+i] = card.pnData[current+i*2+1];
            }

            notifyCounter++;
            if(notifyCounter == notifyPerBuffer){
                notifyCounter = 0;
                bufferCounter++;
            }

                                flagAcquire += samplesPerNotify;
                                //printf("%d\n", flagAcquire);

            // this is the point to do anything with the data

            spcm_dwSetParam_i32 (hCard, SPC_DATA_AVAIL_CARD_LEN,
                lNotifySize);
```

```
            }

            // check for escape = abort
            // if (bKbhit ())
            //    if (cGetch () == 27)
            //       break;
         }
      }
   }

   // send the stop command
   dwError = spcm_dwSetParam_i32 (hCard, SPC_M2CMD, M2CMD_CARD_STOP |
      M2CMD_DATA_STOPDMA);
}

void AQCard::close(){
   // close card
   printf ("Closing card...\n");
   // vFreeMemPageAligned (pnData, (uint64) lBufferSize);
   spcm_vClose (hCard);
   return;
}

void AQCard::cleanMem(){
   // clean up
   printf ("Cleaning memory...\n");
   vFreeMemPageAligned (pnData, (uint64) lBufferSize);
   printf("Finished.\n");
   return;
}

unsigned __stdcall ThreadFunc(void* arg){
   printf( "In thread...\n" );
   // PUT FUNCTIONS HERE THAT ARE SUPPOSED TO RUN IN THE BACKGROUND
   card.acquire();

   _endthreadex( 0 );
   return 0;
}

//// WARNING THREAD ^^^^^^
```

```
void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[]){
    // settings for the FIFO mode buffer handling
    // card.lBufferSize = MEGA_B(25);
        nicard.counter = nicard.quantity = 0;
    card.flagAcquire = 0;
    card.lBufferSize = MEGA_B(4); //was 256
    // card.lBufferSize = MEGA_B(200);
    card.lNotifySize = KILO_B(16);
    card.qwTotalMem = 0;

    //card Params as array
    double* i0;
    mxArray *input0;
    input0 = (mxArray *) prhs[0];
    i0 = mxGetPr(input0);
    // card.qwToTransfer = MEGA_B((uint64) i0[0]);
    // card.totalSamples = (int32) i0[0];
    card.totalSamples = (uint64_li) i0[0];
    card.totalSamples = (card.totalSamples/card.lNotifySize+1) * card.lNotifySize;
    //card.totalSamples = 1998848;
    printf("totalsamples is: %llu\n", card.totalSamples);
    // card.qwToTransfer = 2*2 * (uint64) card.totalSamples; // 2 channels * 2
        bytesPerSample * totalSamplePerChannel
    card.qwToTransfer = 2*2 * (uint64_li) card.totalSamples; // 2 channels * 2
        bytesPerSample * totalSamplePerChannel
        //card.qwToTransfer = 268435456;
    printf("qwToTransfer is :%llu\n", card.qwToTransfer);
    card.postSamples = (int32) i0[1];
    card.segmentSize = (int32) i0[2];
    card.samplingRate = MEGA((int32) i0[3]);
    card.channel1Sens = (int32) i0[4];

    nicard.samples = card.totalSamples/card.segmentSize;
    nicard.rate = (long) 25000.0;

    double Ax = i0[5]; // piezoVelocity
    double Ay = i0[6];
    double PulseRepFreq = i0[7];
    double OwisVelocity = i0[8];

    double* i1;
    mxArray *input1;
```

```
input1 = (mxArray *) prhs[1];
i1 = mxGetPr(input1);
double* A = i1;

double* i2;
mxArray *input2;
input2 = (mxArray *) prhs[2];
i2 = mxGetPr(input2);
double* ROI = i2;

double* i3;
mxArray *input3; //y
input3 = (mxArray *) prhs[3];

mxArray *plotfcn;
plotfcn = (mxArray *) prhs[5];

mxArray *stagefcn;
stagefcn = (mxArray *) prhs[6];
double piezoVelocity = Ax;
int piezoCycle = (int32) i0[9];

mxArray* stageParams;
stageParams = mxCreateDoubleMatrix(3,1,mxREAL);
double* stagePar;
stagePar = mxGetPr(stageParams);

stagePar[0] = piezoCycle;
stagePar[1] = piezoVelocity;
stagePar[2] = OwisVelocity;

mxArray *channel0;
mxArray *channel1;
channel0 = mxCreateDoubleMatrix(card.totalSamples,1,mxREAL);
channel1 = mxCreateDoubleMatrix(card.totalSamples,1,mxREAL);

card.ch0 = mxGetPr(channel0);
card.ch1 = mxGetPr(channel1);

mxArray *channel1Corrected;
channel1Corrected = mxCreateDoubleMatrix(card.totalSamples,1,mxREAL);
card.ch1Corrected = mxGetPr(channel1Corrected);
```

```
    mxArray *maniParams;
    maniParams = mxCreateDoubleMatrix(7,1,mxREAL);
    double* mani;
    mani = mxGetPr(maniParams);
    mani[0] = Ax;
    mani[1] = Ay;
    mani[2] = card.postSamples;
    mani[3] = PulseRepFreq;
    mani[4] = OwisVelocity;
    int frameQuantity = (int32) i0[9]*2;
    // int frameQuantity = (int32) i0[9];
    int windowSize = card.totalSamples/frameQuantity;
        mani[5] = windowSize;

    mxArray *plotParams;
    plotParams = mxCreateDoubleMatrix(3,1,mxREAL);
    double* plotP;
    plotP = mxGetPr(plotParams);
    plotP[0] = windowSize;
    plotP[1] = card.postSamples;

    mxArray *channelni;
    channelni = mxCreateDoubleMatrix(nicard.samples,1,mxREAL);

    nicard.chni = mxGetPr(channelni);

/// POWER CORRECTION RELATED
    double* i4;
    mxArray *input4;// photodiode calibration array
    input4 = (mxArray *) prhs[4];
    i4 = mxGetPr(input4);
    double* lpcal = i4;

    double lpmin=10e-3;// minimum laser pulse energy (uJ)
    double* photoD;
    mxArray* photoDiode;
    photoDiode = mxCreateDoubleMatrix(nicard.samples,1,mxREAL);
    photoD = mxGetPr(photoDiode);

    mxArray *xposition;
    xposition = mxCreateDoubleMatrix(nicard.samples,1,mxREAL);
```

```
    double* xpos;
    xpos = mxGetPr(xposition);

    int shotsPerFrame=100;
    mxArray* currentFrame;
    currentFrame = mxCreateDoubleMatrix(shotsPerFrame,card.segmentSize,mxREAL);
    double* curFrame;
    curFrame = mxGetPr(currentFrame);
///
    nicard.setup();

    card.open();
    card.setup();
        // card.acquire();
    // THREADING
    HANDLE hThread;
    unsigned threadID;

    // Create the thread.
    nicard.acquire();
    hThread = (HANDLE)_beginthreadex( NULL, 0, &ThreadFunc, NULL, 0, &threadID );

// WARNING STAGES WILL MOVE
    mxArray *countdownfcn;
    countdownfcn = (mxArray *) prhs[7];


    mxArray* countdownParams;
    countdownParams = mxCreateDoubleMatrix(1,1,mxREAL);
    double* countdownPar;
    countdownPar = mxGetPr(countdownParams);

    Sleep(1000);
    int totalWait=3;
    for(int i=0;i<=totalWait;i++){
        countdownPar[0] = totalWait-i;
        CallImageSc(countdownfcn, countdownParams); // calls the countdown script
            with 1 input
        if(totalWait-i!=0)
            Sleep(1000);
    }
    StageMove(stagefcn, input2, stageParams);
```

```
// WARNING
    // nicard.acquire();

    double aux=0;
    int shotNumber=0;
    double Dotohp;

    // for(int i=0;i<card.totalSamples;i++){
    for(uint64_li i=0;i<card.totalSamples;i++){
        while((i*2) > card.flagAcquire){ //int acquiredSamplesPerCh = flagAcquire
            / 2;
            Sleep(1);
        }

/// POWER CORRECTION RELATED MODIFIED
        // aux+=card.ch0[i]*card.ch0[i]; // mean of the square of the photo diode
            signal
        aux+=card.ch0[i];
        if( (i+1)%card.segmentSize == 0){
            aux=aux/card.segmentSize;
            photoD[shotNumber]=lpcal[0]*aux*aux*aux+lpcal[1]*aux*aux+lpcal[2]*aux+lpcal[3];
                // cubic polynomial fit
            xpos[shotNumber]= nicard.chni[shotNumber]*A[0]+A[1];// linear fit
            //Corrected transducer signal
            if (photoD[shotNumber]>lpmin)
                Dotohp = 1.0/ photoD[shotNumber];
            else
                Dotohp=1.0;

            for (int n=0; n<card.segmentSize;n++)
                card.ch1Corrected[n+shotNumber*card.segmentSize]=card.ch1[n+shotNumber*card.segmer
                    Dotohp;
            aux=0;
            shotNumber+=1;
        }
///

/// VISUALIZATION RELATED
        if((i+1)%(windowSize) == 0){
            plotP[2] = (double) (i+1)/windowSize;
            // CallPlotCorrected(plotfcn, plotParams, xposition, input3, channel1);
            for(int m=0; m<shotsPerFrame; m++){
```

```
                for(int n=0; n<card.segmentSize;n++){
                    // curFrame[m+shotsPerFrame*n] =
                        card.ch1[i-shotsPerFrame*card.segmentSize+1+
                        m+shotsPerFrame*n]; // raw data
                    curFrame[m+shotsPerFrame*n] =
                        card.ch1Corrected[i-shotsPerFrame*card.segmentSize+1+
                        m+shotsPerFrame*n]; // power corrected data

                }
            }
            CallImageSc(plotfcn, currentFrame);
            // printf("%llu\n", card.qwTotalMem);
        }
///
    }

    WaitForSingleObject(hThread, INFINITE);
        CloseHandle( hThread );

    // CallPlot(plotfcn, channel0, channel1, channelni);
    nicard.stop();
    card.close();
        card.cleanMem();

    printf("total acquired:%llu\nthe goal was:%llu\n", card.qwTotalMem,
        card.qwToTransfer);
    printf("acquisition finished, now copying data to MATLAB\n");

    double* o0;
    plhs[0] = mxCreateDoubleMatrix(card.totalSamples,1,mxREAL);
    o0 = mxGetPr(plhs[0]);
    for(int i=0;i<card.totalSamples;i++){
        o0[i] = card.ch0[i];
    }

    double* o1;
    plhs[1] = mxCreateDoubleMatrix(card.totalSamples,1,mxREAL);
    o1 = mxGetPr(plhs[1]);
    for(int i=0;i<card.totalSamples;i++){
        o1[i] = card.ch1[i];
    }
```

```
    double* o2;
    plhs[2] = mxCreateDoubleMatrix(nicard.samples,1,mxREAL);
    o2 = mxGetPr(plhs[2]);
    for (int i=0;i<nicard.samples;i++){
        o2[i] = nicard.chni[i];
    }

    double* o3;
    plhs[3] = mxCreateDoubleMatrix(shotsPerFrame,card.segmentSize,mxREAL);
    o3 = mxGetPr(plhs[3]);
    for(int i=0; i<shotsPerFrame; i++){
        for(int j=0; j<card.segmentSize;j++){
            // curFrame[i+shotsPerFrame*j] = 1.0;
            o3[i+shotsPerFrame*j] = curFrame[i+shotsPerFrame*j];
        }
    }

    double* o4;
    plhs[4] = mxCreateDoubleMatrix(card.totalSamples,1,mxREAL);
    o4 = mxGetPr(plhs[4]);
    for(int i=0;i<card.totalSamples;i++){
        o4[i] = card.ch1Corrected[i];
    }

// #  define uint64_li unsigned long long int

    // ofstream a_file("C:\example.txt");
    // for(int i=0;i<10;i++){
    //    a_file<< i;
    //    a_file<<"\n";
    // }
        return;
}
```

Bibliography

[1] A. G. Bell. On the production of sound by light. *Am. J. Sci.*, 20:305, 1880. (Cited on page 1.)

[2] Changhui Li and Lihong V Wang. Photoacoustic tomography and sensing in biomedicine. *Physics in Medicine and Biology*, 54(19):R59, 2009. (Cited on pages 1 and 7.)

[3] Lihong V. Wang and Song Hu. Photoacoustic tomography: In vivo imaging from organelles to organs. *Science*, 335(6075):1458–1462, 2012. (Cited on pages 2, 10, and 11.)

[4] Rui Ma, Sebastian Söntges, Shy Shoham, Vasilis Ntziachristos, and Daniel Razansky. Fast scanning coaxial optoacoustic microscopy. *Biomed. Opt. Express*, 3(7):1724–1731, Jul 2012. (Cited on page 2.)

[5] Lidai Wang, Konstantin Maslov, Wenxin Xing, Alejandro Garcia-Uribe, and Lihong V. Wang. Video-rate functional photoacoustic microscopy at depths. *Journal of Biomedical Optics*, 17(10):106007–1–106007–5, 2012. (Cited on pages 2 and 6.)

[6] Jan Laufer, Edward Zhang, Gennadij Raivich, and Paul Beard. Three-dimensional noninvasive imaging of the vasculature in the mouse brain using a high resolution photoacoustic scanner. *Appl. Opt.*, 48(10):D299–D306, Apr 2009. (Cited on page 6.)

[7] Vasilis Ntziachristos and Daniel Razansky. Molecular imaging by means of multispectral optoacoustic tomography (msot). *Chemical Reviews*, 110(5):2783–2794, 2010. (Cited on pages 7 and 8.)

[8] Hao F Zhang, Konstantin Maslov, George Stoica, and Lihong V Wang. Functional photoacoustic microscopy for high-resolution and noninvasive in vivo imaging. *Nat Biotech*, 24(7):848–851, July 2006. (Cited on pages 8 and 9.)

[9] Figures provided by Dr. Héctor Estrada Bertrán. (Cited on pages 13 and 15.)

[10] National instruments http://www.ni.com/data-acquisition/what-is/. (Cited on page 17.)

[11] Spectrum Systementwicklung Microelectronic GmbH. M3i.41xx manual. January 2013. (Cited on pages 20 and 27.)