SDImaging Manual

Ryan D Muir, Shane Z Sullivan

V1.1 (Last Updated July 21, 2013)

Introduction

This manual is intended both for the user and the developer. Developers will be interested in all sections of the manual which describe the full workings of the program, while users will simply be interested in the ‘InitAlazar.m’ and ‘SDImaging.m’ front end sections. Advanced users may find inspiration for new ways to set up the microscope hardware or quickly perform a novel analytical technique by learning about the ConfigureBoard section. For these users, it is advised that they copy and paste the main code into their working folder of the day and alter the copied version of the code; this way, the configuration of the code is always available for consideration at a later date, and the original code & configuration remains intact for other users. Additional tips for the advanced user is outlined in the section titled, ‘Making the program suit your needs.’

The code is, and likely always will be, in a dynamic state of development, and the manual will likely be maintained serviced or maintained only upon request by an outside source. Nevertheless, the developer may take solace that the original code was lain down in an extremely general form for photon detection, which allows new modes of analysis and control to be added. The modes of analysis may change, as may the methods in which the microscope scans, but the bulk of the underlying framework is a strong template from which this program will expand.

Known Bugs, Performance Issues, Suggested New Features

July 5, 2013 Slow Performance Issue: CalculateSamplesPerPixel takes ~1 second to complete for a typical galvo/resonant scan, but for longer scans, it can take 5 seconds or more. The user may be annoyed that it takes such a long time to begin image capture.

July 5, 2013 Slow Performance Issue: Setting AcqParameters.BufferSize to a large value is necessary in captures where the computer cannot handle the data load in realtime. It takes ~1 second per gig for the program to acquire data. I do not know any way around this.

July 5, 2013 Capture Efficiency Performance Issue: The program continually captures data and stores it in the RAM buffer until AlazarAbortAsyncRead is called, or until the buffer is filled. In either of these 2 events, the data remaining in the buffer is discarded. For cases where a large buffersize was set the buffer overflowed, the user is likely trying to capture and process as much data as possible before the buffer fills. It would be better to detect if the buffer is nearly filled, stop further data acquisition, and then process what is available in the buffer. I do not know if this is possible.

July 5, 2013 Sensitive clock on the ATS9462: The ATS9462 seems to lock up if given an invalid, messy, or interrupted clock. It’s particularly sensitive while in 10MHz PLL mode. While locked up, the card will often refuse to trigger (if using the AlazarDSO vendor software), or otherwise behave strangely. Restarting the entire computer is the only way to restore it to a normal state.

July 5, 2013 Trigger enable doesn’t work on ATS9462: The ATS9462 doesn’t have a working trigger enable. The ATS9350’s trigger enable works fine. Use the ATS9350 if you need a trigger enable for your experiment.

July 5, 2013 First few samples of capture read ‘0’ on ATS9462: The ATS9462 sometimes reads ‘0’ for its first few samples. This seems to be a random glitch. For the user, this makes some pictures appear ‘washed out’ while viewing in matlab. On close inspection, one of the pixels in the first column will appear to be noticeably darker than the others. One failed pixel out of 512x512 pixels should be a minor issue, and should not affect the quantitation of the data anywhere else.

July 5, 2013 Clock cannot be set below 150MHz in 10MHz PLL mode on ATS9462: This is not a glitch, but rather a performance hindrance. It is well documented by Alazar that the valid clock values in 10MHz PLL mode are any integer MHz between 150MHz and 180MHz. If the user needs to use 10MHz PLL mode with the 9462 cards, set it to an integer multiple of the laser firing frequency. Otherwise, use direct clock mode.

June 21, 2013 Cannot Mask Data Acquisition Channels: This is an Alazar MATLAB issue; it is intended that the data collection channels be selectable. Collecting data from only the selected channels would be the most resource efficient way of programming this. Unfortunately, this is a broken feature on Alazar’s end, and can cause MATLAB to crash. This has been remedied by setting all channels to always collect data to ensure a robust data collection session. To reclaim some amount of efficiency, only the data from a particular board is actually read out from DMA protected RAM, and of that, only the channels selected to be processed are considered when compiling images.

Making the program suit your needs

Due to the nature of analytical chemistry, you will likely find yourself needing to perform some analytical method that hasn’t *quite* been performed before. Before despairing that science is hard, a modestly detailed understanding of how this program works, as well as some knowledge of the features available on the Alazar card, may prove that this program can be adapted to your needs in a pinch.

1) SaveRawData

The most reliable and go-to method for performing a novel analysis of the imaging data is to use the ‘SaveRawData’ option. This option will save the raw data from every laser shot as it comes off of the ADC. Afterwards, a custom program can be scripted by the user to perform whatever analysis is required. This is a far more attractive option than attempting to edit the original code! The user may want to refer to the script CalculateSamplesPerPixel.m for inspiration on how to analyze the captured data. For example, should the user want to descramble the data into an image, a relevant portion of that program is shown below for galvo/resonantmirror scanning, which assigns each of the datapoints in a frame into the X and Y coordinates of the pixel in the final image which it belongs:

%The number of mirror periods per frame is equal to the number of galvo steps (Number of Horizontal Lines)

NumResMirrorPeriods = AcqParameters.HPixels;

ShotsPerFrame = AcqParameters.ShotsPerMirrorPeriodGalvoRes\*NumResMirrorPeriods;

%Determine the starting phase of each mirror. The user specified the number of laser shots

%until 0 degrees crossing of each mirror from start of capture.

ResMirrorPhase = 2\*pi\*AcqParameters.MonitorPhaseDelay/AcqParameters.ShotsPerMirrorPeriodGalvoRes;

%Places laser shots in bins from 1 to VPixels

ShotBinsY = 0.5\*sin(linspace(-1\*ResMirrorPhase,(2\*pi\*NumResMirrorPeriods)-ResMirrorPhase,ShotsPerFrame+1))+0.5;

ShotBinsY = round((AcqParameters.VPixels-1)\*ShotBinsY(1:(end-1)))+1;

%Places each laser shot in the correct bin. The galvo steps once after every VPixel sweep.

ShotBinsX = repmat(1:AcqParameters.HPixels,[AcqParameters.ShotsPerMirrorPeriodGalvoRes,1]);

ShotBinsX = ShotBinsX(1:end);

While saving the raw data will definitely work, it should be considered a method of last resort. Scripting a custom analysis can be a complicated task. Further, every experienced experimenter understands the high importance of verifying the quality of the data immediately as it is taken, and it is inconvenient to load every saved file and check that the experiment is going well. If the existing functions of SDImaging can be adapted to your needs, this is highly desirable. Consider having your script called at the end of ‘AcquireData.m’ to automatically view your results.

2) Lock-in amplification & TotalCycledPolarizations

A common application of this program is to replace lock-in amplification. For recovering a very small signal from a very large background, encoding your signal into a narrow band of frequency space is hard to beat! In such a case, the recorded data is likely a signal which has been narrowband amplified and filtered. If the modulation frequency can be an integer division of the laser frequency, then a method of analyzing the data which some may find preferable to working with the raw data directly is to use the ‘TotalCycledPolarizations’ to separate the timeseries of data into the integer divided number of polarizations. The program will naively separate and bin the datapoints by modulation phase, and then a simple script can multiply the resulting few images by one period of a sine wave and summed (to be accomplished with a custom script or in imageJ). This way, the user can get some understanding of the quality of their data from the images after capture. To prevent ‘smearing’ of the image along the fast axis, an unfortunate side effect of lock-in amplification, the user may wish to reduce the number of pixels along the fast axis of detection.

For users who are a bit more familiar with the code, more power is available to quickly achieve an all-around excellent solution. A couple lines of modification can achieve lock-in amplification in near-realtime. Two-Phase lock-in amplification is simply the multiplication of the realtime data trace by a sine wave and a cosine wave, and then the magnitude (the square root of the sum of the squares) of every datapoint gives the amplitude of the frequency component. If these operations could be performed in realtime on the rawdata, the SDImaging program would natively group the data appropriately and generate an image. Grouping together locked-in datapoints which were sampled at different points in time has been coined ‘discontinuous lock-in’ by the Simpson group. To perform these operations on the raw data, find the below piece of code:

%Transfer the data from DMA protected RAM into free RAM. For whatever reason, reading from DMA protected RAM is slow. Reading from it only once is fastest.

RawDataTemp(:,ChannelOrder(Board(boardId).ActiveChannels)) = reshape(bufferOut.Value,[samplesPerRecord,numel(ChannelOrder(Board(boardId).ActiveChannels))]);

Here is where the rawdata is pulled off of the DMA buffer. If each channel of RawDataTemp is multiplied by a sine wave and cosine wave at the modulation frequency, with the wave spanning the number of shots per frame, then re-storing the magnitude of each of those datapoints in RawDataTemp would allow SDImaging to group the data by pixel appropriately and generate an image. A little bit of understanding of the code can go a long way! If you do modify the code, be sure to copy it to your local working directory of the day before you do so. Don’t modify the script that everyone uses!

3) Alazar hardware features & trigger enable

Another facet of Alazar SDImaging suite which is helpful to learn about are all of the functions of the digital I/O. The manual provides a complete list, but particularly useful digital I/O are: busy, serial data, pacer, and trigger enable. For capturing polarization dependent data, the trigger enable can be a particularly useful mechanism. The trigger enable acts as a second trigger; the primary trigger is ignored until the secondary trigger is triggered. Once the second trigger is tripped, the image will start immediately when the primary trigger is triggered. For capturing polarization dependent data, there will likely be a function generator which sources the modulation voltage to the polarization modulating device. The image is normally triggered by a signal indicating the position of the resonant mirror(s); if the resonant mirror indicator signal is put over to the trigger enable, and the sync output of the function generator is connected to the regular trigger, the image will start when 1) the resonant mirror(s) is (are) in place, and 2) the first polarization comes up, as indicated by the sync output.

In some cases, direct clocking is advantageous over the typical 10MHz phase-locked-loop clock. Some of the alazar cards (the ATS9462 cards in particular) do not support clocking at 80MHz when fed a 10MHz clock; they must instead be clocked at 2x the laser frequency, or 160MHz. If the extra data is undesired, then direct clocking using a phototodiode and a pickoff works well; using the laser’s oscillator photodiode works even better, since its amplitude doesn’t vary as you change the power. Lasers that produce clocks which are more than ~200kHz away from 10MHz should be directly clocked; Significant timing jitter has been observed outside this range of tolerance, which adds substantial noise to the signal. Familiarizing with the clocking options (including the internal clock in some cases) can be useful!

4) SubFrames

An issue that will commonly arise in polarization modulation imaging is that the number of cycled polarizations must evenly divide into the number of samples per frame. Perhaps you are here reading this because you got such an error! One way of dealing with this is to slightly change the number of shots per frame. This will likely deviate your resonant mirror from its peak resonant frequency, which will likely be OK for most applications. The image resolution can then be reduced so that each pixel experiences every polarization. For Lissajous, this may be an unacceptable solution, because it is likely that most pixels will not experience all polarizations, even at a reduced resolution. If it is necessary to capture a very long exposure with many channels, saving the raw data may not be an option. An alternative method of dealing with this issue for long exposures is to multiply the number of samples per record (number of samples per frame) to a value which is evenly divisible by the number of cycled polarizations, and then use subframes to ensure that the different polarizations stay separate. For example, suppose that there are 8 total cycled polarizations; multiplying the number of subframes by 8 takes that many times as much data as originally specified, and then setting subframes to 8 separates the data into that many subframes. The images can then be averaged offline (either in imageJ or with a matlab script). The samplesPerRecord can be manipulated at the following line of code in acquireData:

if AcqParameters.LissaJous

samplesPerRecord = lcm(AcqParameters.ShotsPerSlowMirrorPeriod,AcqParameters.ShotsPerFastMirrorPeriod); %One LissaJous Period

else %GalvoRes

samplesPerRecord = AcqParameters.ShotsPerMirrorPeriodGalvoRes\*AcqParameters.HPixels; %I.E. Number of Galvo Steps/Lines

end

5) Oversampling

In some applications yet to be developed, a person may wish to sample faster than the laser fires. Collecting data between pixels can potentially be useful for filtering out noise or capturing data from continuous wave optical sources. In this case, initAlazar or configureboard may be edited to capture at the desired rate, and ShotsPerMirrorPeriod can be readjusted to compensate for this faster sampling. In order to separate samples which are on & off the laser pulses, TotalCycledPolarizations can be set to separate an arbitrary number of temporally adjacent pulses into separate images.

6) The vendor software & stream to disk

Some radical pilot stage experiments require data capture, but are not necessarily scanning or imaging applications. Examples include building and testing an interferometer, checking the rotation angle of quartz, or calibrating a waveplate. Otherwise, the Alazar is sometimes simply used as an oscilloscope that captures extremely long data sequences, which is useful for diagnosing problems with laser stability. Certainly, the main SDImaging software can be made to work in a pinch after cleverly editing enough values. However, just a much simpler program to stream data to disk, potentially after a trigger event, would be desirable. In this case, using the AlazarDSO vendor software is the best option available. The ‘stream to disk’ option in the dropdown menu saves all of the raw data in the format specified. Further, the main ‘oscilloscope’ program can be used to preview your data capture in realtime before saving to disk. The downside of this program is that the external I/O ports cannot be configured, except as a trigger enable.

7) PreTrigger Samples

In this program, acquisition is considered as starting the moment when triggered. However, the Alazar cards actually capture data continuously, even when the computer is idle. Only when the computer is shut down do the cards stop acquiring data! Therefore, samples are available from before the trigger event. In your experimental setup, if the trigger regularly comes some number of samples late, the preTriggerSamples and postTriggerSamples can be edited to shift the acquisition back in time a bit. Note that the total record size must stay the same. Look for this line of code in acquireData.m:

%Set some variables to make Alazar Functions happy

preTriggerSamples = 0;

postTriggerSamples = samplesPerRecord;

8) New scanning methods & the unknown

There are many analytical techniques that will likely be developed with the aid of this software, most of which I could not predict or begin to outline here. However, the general tools were presented in all of the previous points. If you’re stuck, try familiarizing yourself with the code a bit more. The current architecture provides a great platform for manipulating data through the TotalCycledPolarizations and SubFrames commands. Familiarization with the Alazar I/O port can really help a lot with interfacing with new hardware, or with multiple pieces of current hardware.

Perhaps you have (or will) envisioned some completely new scanning method, and wish to know how easy it will be to implement in the current version of the software. With luck, it may fit into one of the existing methods in a pinch, but in the worst case scenario, you’re on track to being a developer! (*yeaaah!*)*.* Provided that you have at least ~40 hours of matlab experience, it will take another ~6 hours to read the rest of this manual, the code, and glance through the alazar SDK manual. Have a glance down at the last section, ‘Implementing Your Own New Scanning Routine.’ This code was written in a highly general way which makes adding new scanning methods as painless as possible. Perhaps this won’t be a month long project after all!

Hardware Setup

There are many ways to accomplish the hardware setup, which is partly what makes this platform versatile and adaptable to new experiments. Because of the variety of experiments to be performed, as well as the continued development of better or more stable ways of controlling the peripheral devices, there can be no concrete guide for how to setup the hardware for your experiment. This section is meant to serve as a general guide for configuring the hardware platform, and having it timed/controlled by alazar.

One of the major features (and even the namesake) of the SDImaging suite is the ability to synchronously digitize with the laser firing events. The best way to clock the alazar card to the laser is to engage the card’s 10MHz PLL. Using a binary counter/divider circuit, divide the laser frequency down by some integer to 10MHz (ideally sourcing from the laser’s oscillator photodiode), and feed the divided signal into the clock input of the master alazar card (the top card). The Alazar then defines the 10MHz input as 10MHz, even if that 10MHz deviates significantly from a true 10MHz. Supposing that the laser frequency was divided by 8 to generate the ~10MHz signal, setting the Alazar clock to 80MHz will synchronize the Alazar to the laser frequency (even if that frequency is significantly different from 80MHz). With the laser clocked by a 10MHz PLL, the sampling frequency can be set in software (see initalazar.m), which increases the adaptability of the system to new experiments.

In some cases, direct clocking is advantageous over the typical 10MHz phase-locked-loop clock. Some of the alazar cards (the ATS9462 cards in particular) do not support clocking at 80MHz when fed a 10MHz clock; they must instead be clocked at 2x the laser frequency, or 160MHz. If the extra data is undesired, then direct clocking using a phototodiode and a pickoff works well; using the laser’s oscillator photodiode works even better, since its amplitude doesn’t vary as you change the power. Lasers that produce clocks which are more than ~200kHz away from 10MHz should be directly clocked; Significant timing jitter has been observed outside this range of tolerance, which adds substantial noise to the signal. Familiarizing with the clocking options (including the internal clock in some cases) can be useful!

After the clock is setup, the Alazar will need to somehow synchronize or time the peripheral optics of the microscope. There are currently two major types of optics we control with the Alazar & supporting electronics: 1) Resonant optics (mirrors, tuning forks), and 2) galvo mirrors (mirrors on servo motors). One way of controlling galvo mirrors is to have the Alazar card put out a TTL pulse every time the galvo should step to the next line in the scan. The pacer out of the Alazar I/O port generates these pulses nicely. For this strategy, some custom external circuit will need to take that TTL pulse and condition it to step the galvo to a new line, with the number of steps before resetting back to the first line understood by the software as HPixels or VPixels (see SDImaging.m). This circuit will also need to have a clear input, which prevents to the galvo from stepping until the image acquisition sequence has initiated (and the Aux out Busy provides this signal nicely), or alternatively, the circuit can send out a trigger signal which triggers the readiness for a new acquisition to start at the reset point. An alternative strategy is to have a function generator synchronize with the laser with its own 10MHz PLL, and generate a ramp output which recycles back to its starting point every ShotsPerFrame number of pulses. The sync output of the function generator will provide a reference trigger point for the acquisition to start.

Controlling the resonant mirrors can also be done a variety of ways. For labs rich with function generators, a synchronized function generator can be used to control the mirror. A cheaper way of controlling the mirror is to simply drive it directly with the Pacer out; the TTL square wave that the pacer out generates can directly drive the mirror, though applying a high pass filter to remove the DC component may be nice to the mirror. Unless a custom external voltage divide is constructed, there is no way of controlling the amplitude of this driving signal. Currently, the favorite way to control the resonant mirror is to use the JAFCI box, which provides an exceptionally clean driving signal, and also maintains a constant mirror phase.

The Alazar needs to be triggered to start its acquisition somehow. This trigger should signal that the user wishes to initiate an image capture, that all of the mirrors are in their starting positions, and that we are ready to start collecting data right at this very laser shot in time. Any signal that repeatably signals the positions of both mirrors can be adapted for use in software. For lissajous imaging in which multiple resonant mirrors are used, the phase of each resonant optic at the start of acquisition should be specified in SDImaging.m.

Lastly, SD sampling needs to be timed to sample at the peak of every signal event. A way of accomplishing this is to change the cable length from the clock photodiode to the Alazar clock input. Trying different cables of different length can tediously accomplish this task. A more convenient way is to use a ‘trombone’ phase adjuster, or a digital timing delay circuit. While observing a realtime image, adjust the phase of the clock until the signal is maximized. Note that there is some delay internal to the Alazar card, which is dependent upon the amount of amplification the signal receives by the Alazar. Changing your ADC voltage range in the InitAlazar function will change the amount of delay that your signal receives, requiring a new timing optimization!

When creating your own hardware setup, many users will wish to use the AuxIO ‘busy’ signal. The user should note that the busy signal activates at the point when the board is armed and waiting for a trigger, and NOT when the board is triggered! This can still be useful if, for example, you expect to observe a trigger signal once per every linescan. If you’re wanting to use this signal as a ‘allow stepping’ signal to your galvo controller circuit, having a trigger that is expected to come within the first linescan is essential.

InitAlazar.m

This function, along with the SDImaging function, are the two functions which will be ‘run’ in this software suite. In particular, this function should be called at the start of every matlab session, after the laser has been turned on to provide a clock to the Alazar card. The InitAlazar function communicates with the Alazar hardware and sets up all of the ADC ranges, I/O ports, trigger conditions, clock parameters, input impedances, and similar parameters for the card. This will need to be called only once per matlab session, or if the clock signal was accidentally interrupted at any point by a user.

The user configurable parameters available in initAlazar are dependent upon the current state of the code development. Commonly, the configurable parameters are:

Clocktype: 1 (10MHz PLL) or 2(Direct Fast External Clock).

PacerValue: Divides the clock by PacerValue and outputs it on one of the I/O ports through the PacerOut routine. Different cards are engineered differently, so the pacervalue may need to be multiplied or divided by some integer to achieve the correct pacerout frequency.

PulsePair: 0 or 1. A common method in the lab was to use a beamsplitter & delay stage to double the firing rate of the laser. This indicates if this hardware was setup. If it is setup, then double the clock frequency and the PacerValue.

CardType = 1 or 2. 1 for the 9350 cards 2 for the 9462 cards. The different cards have different ways that the pacervalue needs to be adjusted. Also, the different cards have different voltage ranges.

ChanAInputRange = 0 to 4;

ChanBInputRange = 0 to 4;

ChanCInputRange = 0 to 4;

ChanDInputRange = 0 to 4;

🡪Specifies the ADC voltage range input, coded as an integer from 0 to 4. 0 for +- 200mV, 1 for +- 400mV, 2 for +- 800mV/1V (9350 +-1 / 9462 +-800), 3 for +- 2V, 4 for +- 4V. It is important that the most sensitive range be used whenever possible! Setting a very large range to measure a very small voltage results in noisy images!

After processing the pulsepair information to adjust the samplerate and pacervalue, the following variables are stored in the ‘ConfigureVariables’ class: PulsePair, ClockType, SampleRate, PacerValue, CardType, ChanInputRange(all 4 values in a vector).

addpath('C:\DIRAC\Matlab Include') % Add path to AlazarTech mfiles

🡪A proprietary set of matlab include files are loaded into matlab’s path, which provide callable functions to interact with the alazar hardware. These functions are called by matlab through the calllib function, which calls a function in a shared library (help calllib in matlab). For Alazar functions, a typical function call looks like:

<retCode> = calllib(‘ATSAPi’,’<function handle>’,<other parameters>)

AlazarDefs % Call mfile with library definitions

🡪A set of definitions are loaded into memory from AlazarDefs.m (This file can be viewed in C:\DIRAC\Matlab Include\AlazarDefs.m). This script defines several text strings as different binary values. This allows the developer to call alazar functions with intuitive textual parameters, instead binary values.

for Chan = 1:4

switch ConfigureVariables.ChanInputRange(Chan)

🡪The chaninputrange is re-coded into the values expected by alazar

The rest of the code is copied & pasted from the Alazar example matlab files, and the comments placed by Alazar provide indication as to what each section does. The last portion of the code calls configureBoard.

ConfigureBoard.m

ConfigureBoard is called by InitAlazar. Nearly the entire function has been copied & pasted directly from the Alazar code template, with relevant parameters replaced as necessary. For many new experiments, advanced users will adjust the values here will be adjusted to fit the particular needs of the experiment. Alazar functions called in matlab will follow the form:

<retCode> = calllib(‘ATSAPi’,’<function handle>’,<other parameters>)

Additionally, matlab supports continuation of code on the next line by using an ellipses (…). The original alazar developers cleverly used this to document what each of the parameters is doing. Along with the AlazarDefs called at the beginning of this file, which defines several text strings as different binary values, many of the parameters sent are evidently understandable. Due to the extremely high level of documentation in this script, casual inspection of the script and the routines called provides more intuition than I can provide in this document.

Note that not all of the available Alazar functions are called in this script. For an exhaustive list of all alazar function available, the Alazar SDK manual provides a detailed description of every available function. In particular, the ‘Configuring a Board’ section is highly relevant.

Some notable functions which are frequently edited for many experiments (see ‘Making the program suit your needs’) are:

AlazarSetCaptureClock: For some experiments, such as oversampling experiments, users may want to adjust the clock to an arbitrary value. For simply checking the stability of the laser, the user may wish to use the internal clock instead of synchronizing to the laser.

AlazarInputControl: Additional parameters are available, such as AC or DC coupling, and for some cards, input impedance to the capture channels.

AlazarSetTriggerOperation: Can define triggering off of a data channel instead of the external trigger. Can also define the trigger level.

AlazarSetExternalTrigger: Can select trigger input range for some cards, and AC or DC coupling.

AlazarSetTriggerDelay: Tells the card to wait X many samples after trigger event before collecting data.

AlazarConfigureAuxIO: Configures the I/O port on the cards. Commonly used configurations are ‘busy’ and ‘pacer.’ Other options are Aux\_Out\_Trigger, Aux\_In\_Trigger\_Enable, Aux\_Out\_Trigger\_Enable, and Aux\_Out\_Serial\_Data. See the SDK Manual on the AlazarConfigureAuxIO for more information on these commands.

SDImaging.m

This function, along with the InitAlazar function, are the two functions which will be ‘run’ in this software suite. In particular, this function should be called when the user wishes to take an image, and after the InitAlazar function has been run at least once in this matlab session. The SDImaging script currently functions as the front end for the program; it compiles all of the user defined options before calling AcquireData.m to take an image. Note that at the beginning of the script, I import all of the variables from InitAlazar saved in the ConfigureVariables class, and use these values to set some of the capture options. The program is excessively documented within the script, and thus I refer the reader to the script itself to see what most of the options do. I list here some options which deserve some additional explanation.

AcqParameters.SaveRawData: 0 or 1. Saves the raw data as 16bit unsigned integers. The zero value of the ADC is ~32,768 (2^15). Even for 12 or 14 bit digitizers, the numbers are shoehorned into 16bit values. The exact zero point for every channel of every digitizer is unique, and is further unique for every ADC input range. If knowing the exact zero point is important for your analysis, make sure that you have some reference of 0 in your image (such as a square aperture obscuring some of the image), or take an image with the sample removed! Also, there is simply too much data to save to a regular hard disk. A RAID system may be able to handle the high load. A recommended method of saving data is to create a RAM disk, which partitions a portion of the system RAM as a virtual physical drive in the system. ImDisk is an excellent program for creating RAM disks. Indeed, the host computer should have a lot of RAM available!

AcqParameters.BufferSize: The Alazar creates a sector of reserved RAM for direct memory access (DMA) purposes. The cards continually stream data to the reserved RAM through a DMA process. The amount of RAM available for streaming to is set by this value. The program continually captures data and stores it in the RAM buffer until AlazarAbortAsyncRead is called, or until the buffer is filled and the program errors out with the message, ‘Error: AlazarPostAsyncBuffer failed.’ If this error message was encountered during a long acquisition, try increasing the amount of RAM available through this parameter, but open up the task manager and ensure that you’re not oversaturating the available RAM!

AcqParameters.TotalCycledPolarizations: This parameter was created with the intention that the user will have a fast polarization modulating device in the optical circuit, with a modulation period an integer divisor of the sampling rate. For example, if the user has a modulation frequency that is 1/10th the sampling frequency, then setting this parameter to 10 will create separate images for all 10 points of the modulation. Simply, it separates out every 10th sample in this scenario and creates a new image. If no modulation is occurring, or the user does not wish to separately image every N many laser pulses, simply set this parameter to 1. This can be used simultaneously with SubFrames.

AcqParameters.SubFrames: This parameter is most beneficial in lissajous scanning methods, but can be sometimes be adapted for use in other situations (see ‘Making the program suit your needs’). Unlike TotalCycledPolarizations, which separates the data by every Nth sample, this separates the data into N different images by grouping each consecutive set of SamplesPerFrame/N samples. In lissajous, this can be used to create a series of high framerate, sparsely sampled images. This can be used simultaneously with TotalCycledPolarizations.

Channel\*\_Threshold: These parameters are used for photon counting; PMTs put out a negative voltage, and zero voltage is ~32,768. Any voltage sampled below this threshold is considered a count. Note that the values representing zero fluctuate as the ADC range is changed in InitAlazar, as will the absolute voltage that any value represents. The user will need to explore multiple values until one is found which qualitatively maximizes signal/noise.

CountSaturationAlert: If at any pixel a count is observed at every sampling, then the photon counting method has been saturated at that pixel. This qualitatively means that the signal is very bright at that pixel, but we can’t quantitatively say how bright. The user could reduce the laser power, but unless the PMTs are at the end of their linear dynamic range (requiring over 100 photons per laser pulse), your signal/noise is always better from having more signal, and going to signal averaging. See the publication by Muir, Kissick, and Simpson 2010. This will check for saturation of any pixels, and warn the user in the form of a console text display. Detecting saturated pixels does not require a significant amount of additional computing resources.

Lambda: The amount of photons hitting the detector conforms to a Poisson distribution (See the publication by Kissick, Muir, Simpson 2010). When we seek to quantitatively understand how ‘bright’ our signal was at any particular pixel, we are in fact seeking to understand the mean amount of photons that is observed from every pulse (and thus, we are seeking the mean of the Poisson distribution). Lambda is a commonly used notation to describe the mean of the Poisson distribution. Enabling this parameter converts the raw count data into Poisson means. The advantage of converting the data to Lambda is that the observed number of counts scales nonlinearly with the intensity, and converting to lambda linearizes the data. If saturation occurs, this normally returns a NaN value, though this has been adjusted so that it returns a semi-arbitrarily selected value of 15 instead; 15 photons per pulse is a nearly immeasurable value by photon counting, which marks the upper end of this detection scheme. Further, NaN values may not be tolerated in all programs, which ensures later compatability. Converting to lambda does not require a significant amount of additional computing resources.

VoltageClippingAlert: This detects if any individual sampled voltage has exceeded the range of the ADC, causing problems with quantitatively analyzing the data. Unlike CountSaturationAlert and Lambda, this unfortunately does require a significant amount of additional computing resources; it is recommended that this be disabled this for realtime imaging applications.

First/SecondChannelToDisplay: Displays every image in the realtime image viewer window after averaging NumberOfFramesPerImage many frames together. Up to two different images may be simultaneously viewed in realtime. Viewing a realtime display is useful for situations such as realtime microscope alignment or finding an ideal field of view or searching the sample for signal. As of July 5, 2013, modern computers are not quite fast enough to compile every frame into an image in realtime. Integrating multiple frames together is fast, since this is a simple summation of two rawdata arrays, but compiling the rawdata arrays into an image matrix is slow. Therefore, in order to view realtime data from a single channel, NumberOfFramesPerImage must be set to a value around ~6. This of course drops the framerate down by a factor of 6, but is still a very usable realtime image. As computers become faster, this value can hopefully be substantially reduced. Setting NumberOfImages to a large value and SaveProcessedData to false then generates the realtime display for field of view searching and microscope alignment. Note that the system can still save and process data with multiple channels at NumberOfFramesPerImage = 1, but only as long as the buffer set by BufferSize can handle it, and the display will severely lag behind in time.

AcquireData.m

This function is called by SDImaging.m, and is responsible for setting up the user specified acquisition and taking data. Alazar provides some control over how the data is transferred from the card to the RAM buffer via DMA. One buffer is one RAM buffer which the Alazar DMAs to, of which we define several buffers worth of space as defined by the user in BufferSize. The Alazar schedules a DMA transfer to the buffer when a specified number of records has been captured, and deposits all new records into that buffer. In this program, recordsPerBuffer = 1. One record is defined as one frame’s worth of samples, which is specified by samplesPerRecord. Each board dumps both of its channels into one buffer, the first channels record coming sequentially before the second. In this program, each buffer is therefore sized as: 2 Bytes/Sample \* samplesPerRecord\*NumActiveChannels. The total number of frames to be captured is NumberOfFramesPerImage\*NumberOfImages, and is specified by buffersPerAcquisition.

%The number of samples per record (samples per frame) is dependent upon the imaging method

if AcqParameters.LissaJous

samplesPerRecord = lcm(AcqParameters.ShotsPerSlowMirrorPeriod,AcqParameters.ShotsPerFastMirrorPeriod); %One LissaJous Period

else %GalvoRes

samplesPerRecord = AcqParameters.ShotsPerMirrorPeriodGalvoRes\*AcqParameters.HPixels; %I.E. Number of Galvo Steps/Lines

end

🡪For lissajous, the number of samples per record is the number of samples until the mirrors repeat their trajectory pattern. This is a least common multiple problem, and the least common multiple of the Shots per slow and fast mirror periods determines the number of shots per frame. For Galvo/Res scanning, it is simply the number of shots per mirror period times the number of galvo steps.

if mod(samplesPerRecord,AcqParameters.TotalCycledPolarizations ~= 0)

disp('Warning: The number of cycled polarizations does not evenly divide into the number of samples per frame. Polarization data will not be properly separated!');

disp(['The number of samples per frame is: ', num2str(samplesPerRecord)]);

end

🡪 An issue that will commonly arise in polarization modulation imaging is that the number of cycled polarizations must evenly divide into the number of samples per frame. The core assumption of the data processing is that every frame is identical, including by polarization. This assumption was made out of necessity of speed & computing resources (see the data processing later in this section).

%Set some variables to make Alazar Functions happy

preTriggerSamples = 0;

postTriggerSamples = samplesPerRecord;

🡪 In this program, acquisition is considered as starting the moment when triggered. However, the Alazar cards actually capture data continuously, even when the computer is idle. Only when the computer is shut down do the cards stop acquiring data! Therefore, samples are available from before the trigger event. In your experimental setup, if the trigger regularly comes some number of samples late, the preTriggerSamples and postTriggerSamples can be edited to shift the acquisition back in time a bit. Note that the total record size must stay the same.

%In an attempt to speed up the process as much as possible, I'm only going to pull data from the boards with enabled channels.

%Further, to reduce the number of elements that I need to address, I will size the variables to match the number of enabled channels.

%To specify which column the data goes in, which is ideally specified by channel number, I will specify by ChannelOrder(Channel)

ActiveBoards = [any(AcqParameters.Channel(1:2))\*1 , any(AcqParameters.Channel(3:4))\*2];

ActiveBoards = ActiveBoards(ActiveBoards > 0);

ChannelOrder = cumsum(AcqParameters.Channel);

NumActiveChannels = sum(AcqParameters.Channel);

ActiveChannels = find(AcqParameters.Channel);

Board(1).ActiveChannels = ActiveChannels(ActiveChannels < 3);

Board(2).ActiveChannels = ActiveChannels(ActiveChannels >= 3);

🡪I am trying to record data from only the channels which the user specified as active. This requires some adjustment to the AlazarHardwareSetup, as well as some adjustment to my data analysis protocol. All of these variables represent preparation for both places.

The Alazar Hardware Setup section is almost entirely copy & pasted from the Alazar example code. One section which has been modified is this section for selecting which channels to capture from, which is in a state of disrepair as of June 5th, 2013 (see BUG section):

%Select which channels in each board to acquire data from. This was specified by the user in SDImaging.

for boardId = 1:boardCount

% TODO: Select which channels in this board (A, B, or A + B)

enabledChannelMask = CHANNEL\_A + CHANNEL\_B;

enabledChannelMaskArray(1, boardId) = enabledChannelMask;

end

for boardId = ActiveBoards

% TODO: Select which channels in this board (A, B, or A + B)

enabledChannelMask = CHANNEL\_A\*AcqParameters.Channel(boardId\*2-1) + CHANNEL\_B\*AcqParameters.Channel(boardId\*2);

enabledChannelMaskArray(1, boardId) = enabledChannelMask;

end

In the initialize variables, figures, and waitbar before startcapture, everything is rather well documented and requires no further explanation. All variables were instantiated with single point precision to save memory and computing resources. It is worth noting that CalculateSamplesPerPixel is called in this section.

The start capture section is also rather well documented. Some of the code in here is slightly less than elegant, because it is all highly speed optimized. Everything within the main loop is time sensitive, and needs to run as fast as possible. Many permutations of matlab code were generated at each minor section to find the fastest method available. I will specifically highlight some noteworthy sections:

if bufferFull

% Get data off Alazar card buffers into reserved DMA RAM

setdatatype(bufferOut, 'uint16Ptr', 1, samplesPerBuffer);

%Transfer the data from DMA protected RAM into free RAM. For whatever reason, reading from DMA protected RAM is slow. Reading from it only once is fastest.

RawDataTemp(:,ChannelOrder(Board(boardId).ActiveChannels)) = reshape(bufferOut.Value,[samplesPerRecord,numel(ChannelOrder(Board(boardId).ActiveChannels))]);

%Make the buffer available to be re-filled by the board

retCode = calllib('ATSApi', 'AlazarPostAsyncBuffer', boardHandle, pbuffer, bytesPerBuffer);

if retCode ~= ApiSuccess

fprintf('Error: AlazarPostAsyncBuffer failed -- %s\n', errorToText(retCode));

captureDone = true;

end

end % if bufferFull

🡪When a buffer is filled, the data is pulled from the RAM buffer into another RAM location. For whatever reason, reading from the DMA protected RAM is slow, so reading from that section only once is fastest. Afterwards, the buffer is reposted as a free buffer ready to be filled with more data. If a buffer fails to post, then the RAM buffer has previously overflown; try increasing the BufferSize.

%It would be more elegant to write a program without creating separate names for RawDataCounting and RawDataAveraging,

%such as creating a class, but matlab executes faster if these are kept as separate low-level variables.

if AcqParameters.Counting

RawDataCounting = RawDataCounting + single(RawDataTemp < AcqParameters.Threshold\_Channel);

end

if AcqParameters.Averaging

RawDataAveraging = RawDataAveraging + single(RawDataTemp);

end

🡪I take the data just previously stored in RawDataTemp and accumulate its values into another vector. Accumulating the rawdata in this way is extremely efficient; the plus operation is parallelized in matlab. Compiling an image first and accumulating compiled image data is much more computationally expensive. However, because I group my rawdata in this way, the implicit assumption is that every frame is identical. If frames start with different polarizations (samplesPerRecord does not evenly divide by TotalCycledPolarizations), then polarization data will not be correctly separated. AcqParameters.Threshold\_Channel was trimmed and reshaped during the initialize variables section to be the same size as RawDataTemp.

%Transfer the data into the three matrices. I played with separating the rawdata by channel into a new variable rawdatachan, but found that the below implementation was slightly faster.

HighSampleImageArray(AcqParameters.HighSamplePixelGroupMatrixIndex) = RawDataCounting(AcqParameters.HighSamplePixelGroupDataIndex,ChannelOrder(Channel));

MidSampleImageArray(AcqParameters.MidSamplePixelGroupMatrixIndex) = RawDataCounting(AcqParameters.MidSamplePixelGroupDataIndex,ChannelOrder(Channel));

LowSampleImageArray(AcqParameters.LowSamplePixelGroupMatrixIndex) = RawDataCounting(AcqParameters.LowSamplePixelGroupDataIndex,ChannelOrder(Channel));

%Sum the data and transfer it to the FinalImageMatrix

ImageCounting(AcqParameters.HighSampleFinalPixelIndex) = sum(HighSampleImageArray);

ImageCounting(AcqParameters.MidSampleFinalPixelIndex) = sum(MidSampleImageArray);

ImageCounting(AcqParameters.LowSampleFinalPixelIndex) = sum(LowSampleImageArray);

🡪 This is essentially my own fast implementation of matlab’s native accumarray(). From some small simulations, I have found that my own implementation is ~2x faster than matlab’s hardcoded accumarray() function! See CalculateSamplesPerPixel for some background on how \*MatrixIndex, \*DataIndex, and \*PixelIndex were formed. Essentially, the data is reorganized into 3 separate matrices, and binned per column in each matrix. The point of reorganizing in 3 separate matrices instead of 1 large matrix is that the pixels are highly unevenly sampled; creating one large matrix would require a large amount of 0’s to pad the extra elements. Each matrix pulls from pixels that were Highly sampled, sampled a mid-sampled, and lowly-sampled. ~80% of the pixels are considered to be ‘lowly’ or normally sampled. One large sparse matrix would have worked for this purpose, but sparse matrices are actually extremely slow to create and load. After the data is organized into the three matrices, the data is integrated by columns and loaded into the appropriate pixels of final image.

CalculateSamplesPerPixel.m

This function determines the trajectory of the scanning pattern. Once the trajectory is determined, every laser shot is assigned to a pixel, based upon the user's description of image size in vertical and horizontal pixels. A large part of the effort here is to come up with a more efficient version of the accumarray() function, to compile all of the samples into a final image. I have found that the sum() function is incredibly efficient and highly parallelized in matlab. However, it is most efficient if it is called once to perform one large operation, instead of hundreds or thousands of times to perform smaller operations. However, to sum across a matrix padded with 0's is not efficient, which is what would happen if I attempted to create just one large matrix and sum across one dimension. From experimentation, sparse matrices are simply not efficiently implemented in matlab. I have decided that exactly 3 matrices meets the balance of not calling the sum function repeatedly, as well as preventing from summing across vast amounts of 0's. I create these three matrices by grouping pixels based upon the number of times that they were sampled, with a way of automatically assigning cutoffs for which pixel belongs in which group. For images with multiple polarizations and/or subframes, I create all images in parallel. I do not parallelize across channels of acquisition, as I believe the benefit would be insubstantial compared to the increased programming challenge.

This function has been moderately speed optimized and can usually be run in around a second for a typical galvo/resonant setup, though large lissajous trajectories (around 1/2 a second per frame) can require 5 seconds or more.

I have decided to split this function by a giant outer if statement, where the if statement is the current scanning method being used. For LissaJous and Galvo/Res, the code is nearly copy/pasted, due to the high similarity in their techniques. Different scanning methods, such as 3D scanning methods, may require additional modification. Nonetheless, it is hopefully easy to see how a new scanning method could be implemented in this code as a copy/paste job, with minor edits to a couple of the loops to accommodate the new scanning trajectory.

This function is also well commented, but I will highlight relevant portions:

%Places laser shots in bins from 1 to HPixels, also adjusts for starting phase of each mirror

ShotBinsX = 0.5\*sin(linspace(-1\*SlowMirrorPhase,(2\*pi\*NumSlowMirrorPeriods)-SlowMirrorPhase,ShotsPerFrame+1))+0.5;

ShotBinsX = round((AcqParameters.HPixels-1)\*ShotBinsX(1:(end-1)))+1;

🡪This line of code determines the pixel column in the finalimage that each sample came from. A sine wave of amplitude from 0 to 1 is generated in the first line (the resonant mirror scans in a sine wave trajectory), and the second line places each of the samples along the sine wave trajectory into one of HPixels number of bins. The same is repeated for the YPixels. For galvo/res scanning, each line is scanned sequentially, so ShotBinsX is a repmat and reording of 1:HPixels.

%Determine the total number of times each pixel within each subframe was sampled. This will be used later on for determining P and for Signal averaging

SamplesPerPixel = zeros(AcqParameters.VPixels,AcqParameters.HPixels,AcqParameters.TotalCycledPolarizations,AcqParameters.LissaJousSubFrames);

for SF = 1:AcqParameters.LissaJousSubFrames

for Pol = 1:AcqParameters.TotalCycledPolarizations

for i = ((SF-1)\*ShotsPerFrame/AcqParameters.LissaJousSubFrames+Pol):AcqParameters.TotalCycledPolarizations:(SF\*ShotsPerFrame/AcqParameters.LissaJousSubFrames)

SamplesPerPixel(ShotBinsY(i),ShotBinsX(i),AcqParameters.PolOrder(Pol),SF) = SamplesPerPixel(ShotBinsY(i),ShotBinsX(i),AcqParameters.PolOrder(Pol),SF) + 1;

end

end

end

🡪This determines the number of times each pixel within each polarization image and subframe image was sampled. This will be used later on for determining P and/or for signal averaging. It is also used extensively through the rest of the script.

\*LastFrame.m

These functions are all rather short and straightforward. They perform the last step in signal averaging, display the data if requested, and save the data if requested. There is not much to discuss on the matter. There are two functions for each of the methods, counting and averaging. The functions are all nearly identical to each other; the executive decision for having separated display routines was made to allow for the scenario where a truly novel imaging technique. Perhaps some imaging techniques require displaying quite differently (for example, 3D imaging). Having modularized code makes new techniques easy to plug & play.

Implementing Your Own New Scanning Routine

Perhaps you have (or will) envisioned some completely new scanning method, and wish to know how easy it will be to implement in the current version of the software. With luck, it may fit into one of the existing methods in a pinch, but in the worst case scenario, you will need to implement an entirely new imaging trajectory into the code. This is indeed a significant undertaking and requires a full understanding of the code in all its parts. Once this understanding is achieved, this guide is intended to highlight the parts most relevant to implementing a completely new scanning technique. In particular, I am writing with the expectation that 3D scanning will be the next item to be implemented. Despite that an entirely new spatial dimension will need to be accounted for, the amount of code that needs to be altered is surprisingly little:

1) Create new parameters in SDImaging: You will likely have some new optical device which has some parameter that needs to be set. If so, create a parameter(s) and description(s) for it in the SDImaging file, and store it in AcqParameters.<NewParameter>. Storing the parameter in this way will automatically pass it to all of the relevant functions that you will need to update.

2) Update samplesPerRecord (samples per frame) at the beginning of AcqParameters: The number of samples per record needs to be set. Create a new if branch for samplesPerRecord for your new scanning technique, and using all of the relevant user defined parameters, determine the number of samplesPerRecord.

3) Update the variable initialization in AcqParameters: In the ‘Initialize Variables, figures, and waitbar’ section, create a new if branch for your technique. Initialize new variables for ImageCounting, RawDataCounting, Threshold\_Channel, ImageAveraging, and RawDataAveraging.

4) Create a new method in CalculateSamplesPerPixel: Create an entirely new if branch for your technique in CalculateSamplesPerPixel. Copy and paste one of the other methods into it as a template for the next steps.

5) Update ShotBinsX, ShotBinsY, ShotBinsZ in CalculateSamplesPerPixel: Following the previous form, create and fit the X, Y, and Z trajectory into the user selected amount of HPixels, VPixels, and DPixels.

6) Update SamplesPerPixel calculation in CalculateSamplesPerPixel: If you are accessing a new spatial dimension, this new spatial dimension must be included in SamplesPerPixel.

7) Update Index calculation in CalculateSamplesPerPixel: ‘CurrentSampleNum’ will need to be updated. The CurrentIndex calculation will need to be updated.

8) Create two new \*lastframe.m files: Copy and paste the \*lastframe.m files for counting & averaging from one of the previous methods as a template. Decide how you will display the images in matlab if at all. Make sure that the save data routine is correct for your method.

9) Update AcqParameters to call your \*lastframe.m files: Create a new if branch for both the counting and averaging calls to \*lastframe and call your new files.

10) Update the SaveRawData and SaveProcessedData scripts to include your new method.