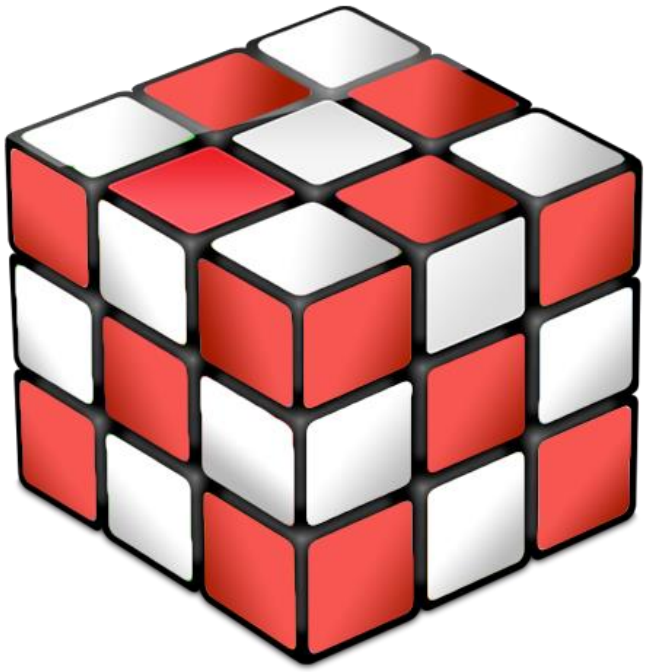
ADVACAM

API C++ Pixel processing

- Clustering
- Spectral Imaging



PIXET

Developer Documentation

1. Table of contents

2. Introduction

The **PIXet** is a multi-platform software developed in ADVACAM company. It is a basic software that allows measurement control and saving of measured data with Medipix detectors. It supports Medipix2, Medipix3, Timepix and Timepix3 detectors and all the readout-devices sold by ADVACAM company such as FitPIX, AdvaPIX, WidePIX, etc. It is written in C++ language and uses multi-platform Qt libraries.

This document describes the pixel processing C++ API. It is using the common library **pxproc**, the **clustering** library providing work with single clusters and the **spectraimg** library for cluster-based imaging. The data can be processed online, while measuring, or offline by data replay functions.

3. Requirements

3.1. Hardware

This API requires computer with x86 compatible architecture (no ARM, but can be on request), 64bit Windows or Linux and connected some Advacam hardware with imaging chip. Medipix3, Timepix, Timepix3, etc.

Some functions are universal for all hardwares (pypixet.start(), dev.doSimpleAcquisition(…), etc).
Some functions are not working with all (dev.setOperationMode(…) is not working with devices that has configurable individual pixels).
Some functions are specialized for only one chip type (dev.setColorMode(…) is Mpx3 only).

3.2. Software

The Pixet Python API can be used from the Python 2.7 interpreter integrated in the Pixet program or from commandline with external Python up to 3.x without the Pixet.

For starting from the Pixet Python scripting plugin are not need any special files.

If you want to run scripts without the Pixet, need additional files:
API functions using of **pypixet.pyd** and **pypxproc.pyd** and need **Python versions 2.7 to 3.x**, for Windows the Pixet core dlls: **pxcore.dll**, **pxproc.dll**, or linux **.so** ekvivalents. The library is **64bit only**.

The Pixet core need the **pixet.ini** file with proper hwlibs list inside, necessary hardware dll files (eq **minipix.dll**), subdirectory "**configs**" with config files for all present imaging chips (eq MiniPIX-I08-W0060.xml).

Pixet core on Windows need more Microsoft Visual Studio .NET standard dlls (vccorlib140.dll etc).

3.2.1. Required files – in more detail

3.2.1.1. The Pixet core, python and additional libraries:

pxcore.dll (allways), **pxproc.dll** (clustering+spectral imaging) or equivalent SO files on Linux
clustering.dll or **spectraimg.dll** (clustering or spectral imaging) or equivalent SO files on Linux

3.2.1.2. The **pixet.ini** file and the hwlibs

In the active directory must be the pixet.ini file. It must contains the [hwlibs] section with list of hwlib DLLs (or SOs) for devices that your project may supports. The hwlib files must be located in the locations specified in the pixet.ini. A semicolon at the beginning of a line disables the line.

Example1:
```
[Hwlibs]
hwlibs\minipix.dll
hwlibs\zest.dll
```

Example2:
```
[Hwlibs]
minipix.dll
zest.dll
;zem.dll
```

(Examples is for a Minipix and a Widepix with Ethernet)
Example1 is for hwlibs in the "hwlibs" subdirectory.
Example2 is for hwlibs with all files in the active directory and with Advapix disabled.

Hwlibs list:	Minipix:	minipix.dll
(required)	Widepix with Eth:	zest.dll
	Widepix without Eth:	widepix.dll
	Advapix:	zem.dll, okFrontPanel.dll

3.2.1.3. Additional files for some devices

Device INI files list:	Minipix:	minipix.ini
(not required)	Widepix with Eth:	zest.ini
	Widepix without Eth:	widepix.ini
Firmware images list:	Widepix-L:	zestwpx.bit
(required)	Widepix-F:	zemwpxf.rbf
	Advapix-tpx3:	zemtpx3.rbf
	Advapix-tpx3-quad:	zemtpx3quad.rbf
	Advapix-Timepix:	zemtpx.rbf
	Advapix-Tpx2:	zemtpx2.rbf

3.2.1.4. The **factory** and the **configs** directories

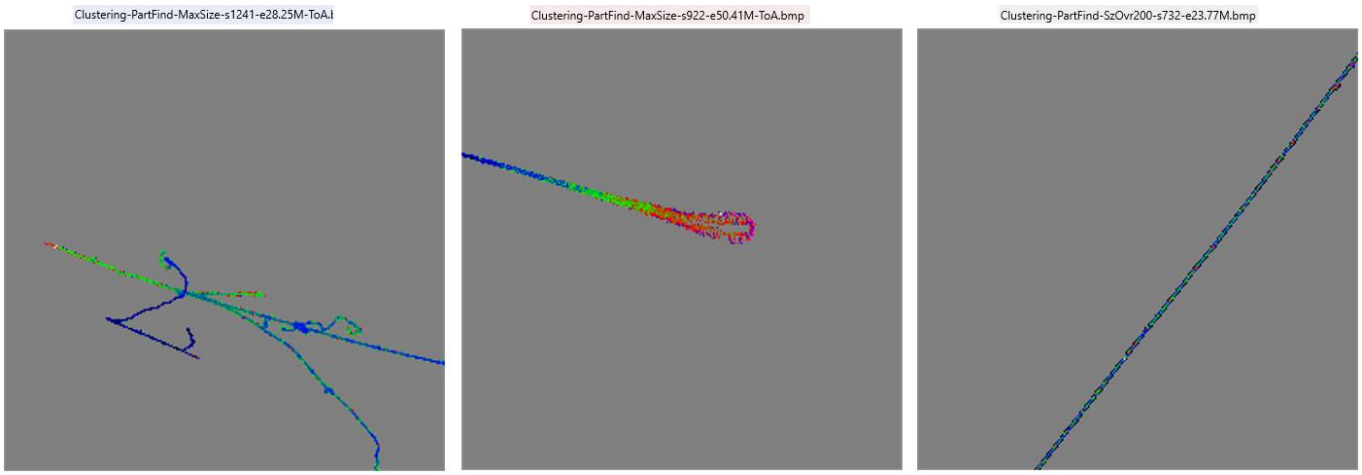
The **factory** directory should contain the factory default configuration XML files. The Pixet core use it while starting, if the configuration file is not in the configs directory or program can use it by the **loadFactoryConfig()** method. This directory not need if the device has an internal config memory (Minipix for example).

The **configs** directory contain configuration XML files. The Pixet core try to use it while starting in **pypixet.start()** method and automatically save the current settings to it, if the **pypixet.exit()** method is used.
This process works the same way when you start and quit the Pixet program.

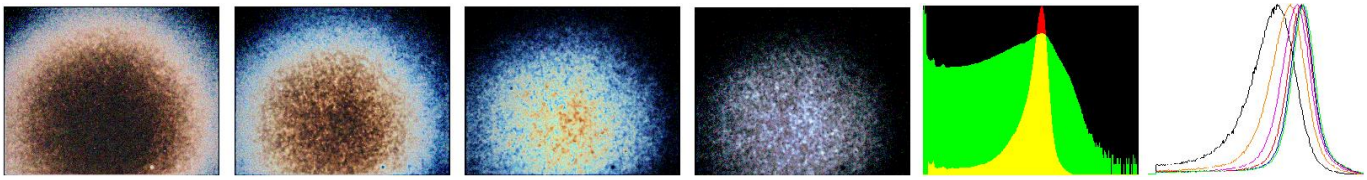
4. Overview of the Pxproc API

This is the simplified, specialized API for clustering and spectral imaging.

pxproc: The common library required for the Pixel processing API.
clustering: The specialized library for work with single clusters. Converts pixels to clusters. It can be processed statistically by information like as cluster size, energy, … or You can get pixel list of each cluster.
spectraimg: The specialized library for cluster-based imaging. Converts pixels to clusters and next converts this clusters to virtual pixels on image.



Clustering - Sample outputs from an experiment



SpectraImg - Sample outputs from an experiment

5. The Clustering and related

The Clustering is designed for easy work with clusters of the pixels. Typically used to determine energy and other parameters of high energy particles.

5.1. The Clustering API

This API is defined in the clusteringaip.h file. Typical usage:

1.a Load the Pixet core using the **pxpClLoadPixetCore**("pxcore.dll").
1.b Or normally start the application using the Pixet core API **pxcInitialize**() function, get the core pointer using **pxcGetIPixet**(), set the core pointer to the Clustering API using **pxpClSetIPixet**(iPixet);
2. Get the clustering instance handle using **pxpClCreate**(device_index);
3. Set-up the callbacks (not required).
4. Load the calibration (not required).
5. Start the measurement or repaly your data.
6. Use the data via one of NewClusters callbacks or from saved clog file.

5.1.1. LoadPixetCore and UnloadPixetCore

```
PXCLAPI int pxpClLoadPixetCore(const char* pxCoreLiPath);
```

Loads the pixet core library (pxcore.dll/so). When the measurement with a device is intented the user has to either load pixet core with this function, or if the core is already loaded in the application (pxcore.dll/so was loaded separatelly), the setIPixet function has to be called.

Example:
```
int rc = pxpClLoadPixetCore("pxcore.dll");
errorToList("pxpClLoadPixetCore", rc);
```

```
PXCLAPI void pxpClUnloadPixetCore();
```
Deinitializes ane unloads the Pixet core.

5.1.2. SetIPixet and GetIPixet

```
PXCLAPI void pxpClSetIPixet(void* pixet);
```

Sets the internal Pixet API pointer. This is used when pxcore library is loaded separatelly in application. The use must pointer obtained via function pxcGetIPixet and should not load the pixet core with pxpSiLoadPixetCore function.

Example:
```
iPixet = pxcGetIPixet(); // Warning: Use the pxcGetIPixet from pxcapi.h,
                         // not pxpClGetIPixet from clusteringapi.h
if (iPixet==0) msgToList("pxcGetIPixet=NULL"); else  msgToList("pxcGetIPixet OK");
pxpClSetIPixet(iPixet);
```

```
PXCLAPI void* pxpClGetIPixet();
```
Return internal Pixet structure pointer or 0 if not set.
Can be used if you want use functions of the pxcore API if the program was started using pxpClLoadPixetCore.

Warning: Do not confuse this with the **`pxcGetIPixet()`** function of the pxcore API.

5.1.3. Create and Free

```
PXCLAPI clhandle_t pxpClCreate(int deviceIndex=CL_NO_DEVICE);
```

Creates a new instance of clustering.

deviceIndex index of the device this clustering instance will manage.
If used offline (pxcore library not loaded), use CL_NO_DEVICE. The measurement will be not possible. Only replaying of data.
Note: If no device present, device with idx 0 is virtual file device. This is second way to offline use.

return Returns the handle of newly create instance of Spectra Imaging, or CL_INVALID_HANDLE if error

Examle:

```
clHandle = pxpClCreate(0);
if (clHandle==CL_INVALID_HANDLE) msgToList("pxpClCreate INVALID");
else  msgToList("pxpClCreate OK");
```

```
PXCLAPI int pxpClFree(clhandle_t handle);
```

Frees the created instance of clustering

handle clustering handle

return 0 if OK, otherwise error code (CL_ERR_XXX)

5.1.4. GetLastError

```
PXCLAPI int pxpClGetLastError(clhandle_t handle, char* errorMsgBuffer, unsigned size);
```

Gets the last error message - when a function returns error code. Gets either global message when handle = 0, or message for the specific clustering instance. You can use this function or the message callback.

handle clustering handle received from function pxpSiCreate

errorMsgBuffer output buffer where the error message will be stored

size size of the supplied errorMsgBuffer

return 0 if OK, otherwise error code (SI_ERR_XXX)

5.1.1. LoadCalibrationFromDevice and LoadCalibrationFromFiles

```
PXCLAPI int pxpClLoadCalibrationFromDevice(clhandle_t handle);
```

Loads the calibrations from the physically connected device. The device must support this feature. Minipix Tpx3 for example. Returns 0 if OK or error code (CL_ERR_XXX).

```
PXCLAPI int pxpClLoadCalibrationFromFiles(clhandle_t handle, const char* filePaths);
```

Loads the calibration files (a,b,c,t files or a single xml device config file).
Returns 0 if OK or error code (CL_ERR_XXX).

Examples:

```
pxpClLoadCalibrationFromFiles("mydetector.xml");
pxpClLoadCalibrationFromFiles("calibA.txt|calibB.txt|calibC.txt|calibT.txt");
```

5.1.2. StartMeasurement, ReplayData, IsRunning and Abort

```
PXCLAPI int pxpClStartMeasurement
            (clhandle_t handle, double acqTime, double measTime, const char* outputFilePath);
```

Starts measuremnt with the device for specified time and process the data. Only if the SI connected to the IDev. If calibration is loaded, energy values will be calibrated. Measurement works in the background. Use while-isRunning() to wait for end, if need it.

handle Clustering instance handle

acqTime acquisition time of a single frame / pixel measurement in seconds. Primary for **frame-based devices** (Medipixes, Timepix, no Tpx3): This is single frame time. Use a short enough time to prevent clusters overlapping. Too short time can cause too many losses between frames. On **data-driven devices** (Timepix3, no Timepix), this is the ToA limit. After exceeds, ToA is resets and acqIndex in the newClusters… callbacks is incremented. acqTime=measTime can be used.

measTime total time of measurement in seconds. Use 0 to endless measurement (progress will always 100%).

outptuFilePath output file where the process data (clusters) will be saved (*.clog). If saving not required, put "".

return 0 if OK, otherwise error code (CL_ERR_XXX)

```
PXCLAPI int pxpClReplayData
            (clhandle_t handle, const char* filePath, const char* outputFilePath, bool blocking);
```

Replays and process already measured data files (*.pmf, *.txt, *.t3r, *.t3pa, ...) and calls the corresponding callbacks. If calibration is loaded, energy values will be calibrated. If the output path defined and ending with .clog, cluster log will be saved.

handle Clustering instance handle

filePath full path to a data file to be replayed

outputFilePath full path to a output file (e.g. cluster log *.clog). if saving to output file not required, put ""

blocking whether the function will block until all clusters processed

return 0 if OK, otherwise error code (CL_ERR_XXX)

Clog files note:

For historical reasons, there are two **CLOG** formats. Tpx3 have one additional column. The **pxpClReplayData** cannot replay CLOG form Tpx3. Use the **T3PA** instead the CLOG. The T3PA files can be generated by data-driven measuring in the pxcore API.

```
PXCLAPI int pxpClIsRunning(clhandle_t handle);
```

Returns 1 if running, 0 = not running, < 0 error.

```
PXCLAPI int pxpClAbort(clhandle_t handle);
```

Aborts the measurement or replaying of the data. Returns 0 if OK or error code (CL_ERR_XXX).

5.1.3. SetNewClusters…Callback functions

This are functions that where sets the new cluster callback. There are 2 variants ot the outputs: Clusters parameters list or Cluster list including pixels lists of each cluster. Only one of the callbacks NewClustersCallback or NewClustersWithPixelsCallback can be set. If was set both, only last is working.

```
PXCLAPI int pxpClSetNewClustersCallback
                        (clhandle_t handle, ClNewClustersCallback callback, void* userData);
```

Callback when new clusters are measured/processed. Provides a statictics about clusters.

```
typedef void (*ClNewClustersCallback)
       (PXPCluster* clusters, size_t clusterCount, size_t acqIndex, void* userData);
```

clusters array of new clusters
clusterCount size of clusters array
acqIndex index of curent acquisition
userData pointer to data of the user that were set in set callback function

```
typedef struct _PXPCluster {
    unsigned eventID;       // event id to recognize coincidence events (same ID)
    float x;                // x coordinate of cluster
    float y;                // y coordinate of cluster
    float energy;           // energy of cluster in keV
    double toa;             // time of arrival
    unsigned short size;    // size of cluster in pixels (number of pixels)
    float height;           // maximal pixel value in cluster
    float roundness;        // roundness (0 - 1)
} PXPCluster;
```

```
PXCLAPI int pxpClSetNewClustersWithPixelsCallback
                    (clhandle_t handle, ClNewClustersWithPixelsCallback callback, void* userData);
```

Callback when new clusters are measured/processed. Provides a statistics and all pixel data.

```
typedef void (*ClNewClustersWithPixelsCallback)
       (PXPClusterWithPixels* clusters, size_t clusterCount, size_t acqIndex, void* userData);
```

clusters array of new clusters (with pixels)
clusterCount size of clusters array
acqIndex index of curent acquisition
userData pointer to data of the user that were set in set callback function

```
typedef struct _PXPClusterWithPixels {
    unsigned eventID;       // event id to recognize coincidence events (same ID)
    float x;                // x coordinate of cluster
    float y;                // y coordinate of cluster
    float energy;           // energy of cluster in keV
    double toa;             // time of arrival
    unsigned short size;    // size of cluster in pixels (number of pixels)
    float height;           // maximal pixel value in cluster
    float roundness;        // roundness (0 - 1)
    PXPPixel* pixels;
} PXPClusterWithPixels;

typedef struct _PXPPixel {
    unsigned short x;       // x coordinate of pixel
    unsigned short y;       // y coordinate of pixel
    double toa;             // time of arrival in nano seconds
    float energy;           // energy of pixel in keV or ToT count if no calibration
} PXPPixel;
```

5.1.4. Other Set…Callback functions list

```
PXCLAPI int pxpClSetMessageCallback
                        (clhandle_t handle, ClMessageCallback callback, void* userData);
```

Callback for messages and error messages returned from the SDK.

```
typedef void (*ClMessageCallback)(bool error, const char* message, void* userData);
```

error true if error message
message text of the message
userData pointer to data of the user that were set in set callback function

```
PXCLAPI int pxpClSetProgressCallback
                        (clhandle_t handle, ClProgressCallback callback, void* userData);
```

Callback for progress of an operation. Occurs every 1 second while measuring or processing.

```
typedef void (*ClProgressCallback)(bool finished, double progress, void* userData);
```

finished true if operation finished
progress percentage of progress 0 - 100
userData pointer to data of the user that were set in set callback function

```
PXCLAPI int pxpClSetAcqStartedCallback
                        (clhandle_t handle, ClAcqStartedCallback callback, void* userData);
```

Callback for when a single acquisition is started.

```
typedef void (*ClAcqStartedCallback)(int acqIndex, void* userData);
```

acqIndex index of the current acquisition
userData pointer to data of the user that were set in set callback function

```
PXCLAPI int pxpClSetAcqFinishedCallback
                        (clhandle_t handle, ClAcqFinishedCallback callback, void* userData);
```

Callback for when a single acquisition is finished.

```
typedef void (*ClAcqFinishedCallback)(int acqIndex, void* userData);
```

acqIndex index of the current acquisition
userData pointer to data of the user that were set in set callback function

5.1.5. Typical callbacks orders

Measure:

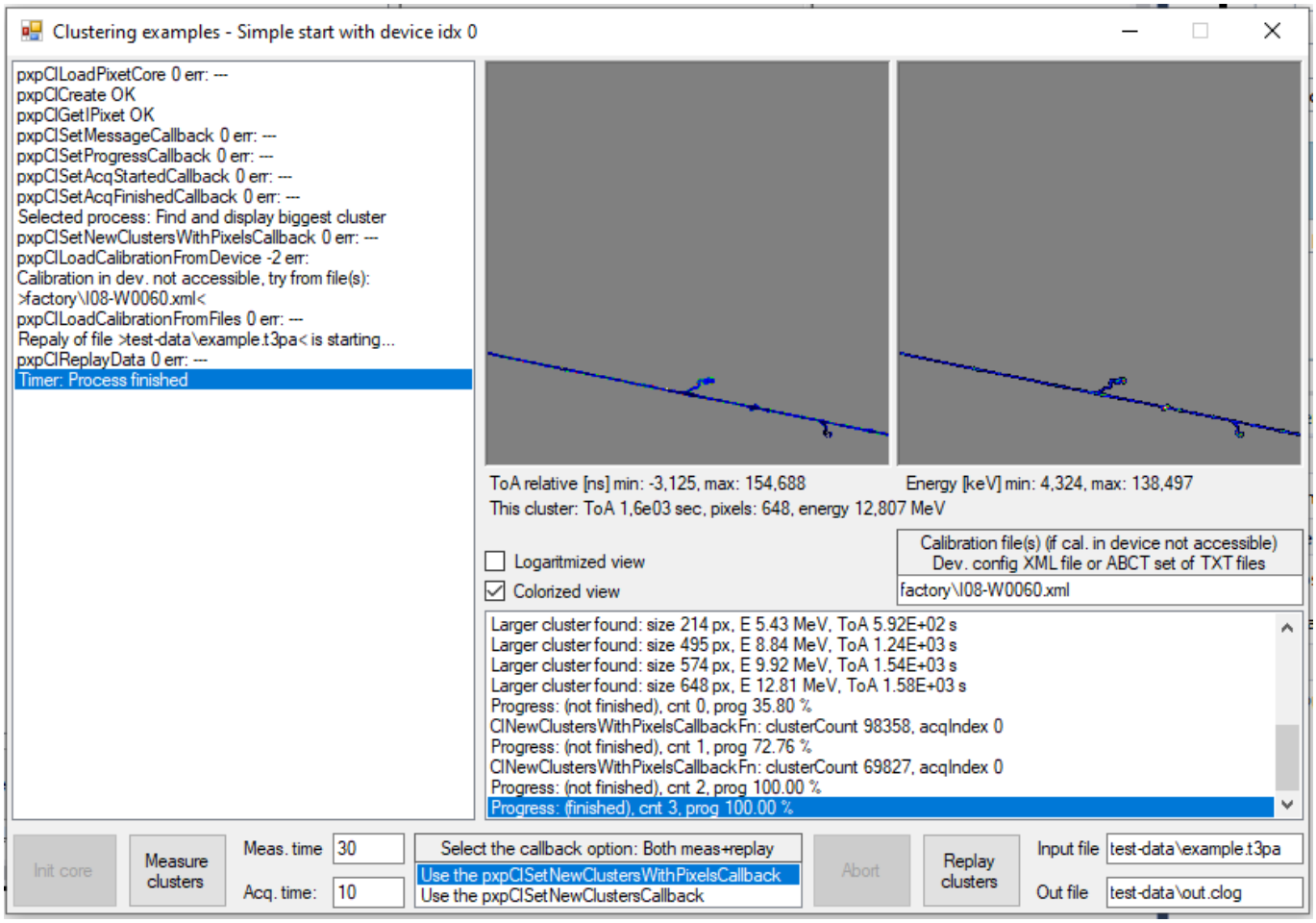
1. AcqStarted 0
2. Progress with finish false (0 to more occurance)
3. NewClusters or NewClustersWithPixels (0 to more occurance)
4. Progress with finish false (0 to more occurance)
5. AcqFinished 0
6. (repeats of 1-5 with Acq number 1 and more: depends of measTime/acqTime ratio and CPU usage)
7. Progress with finish true

Repaly:

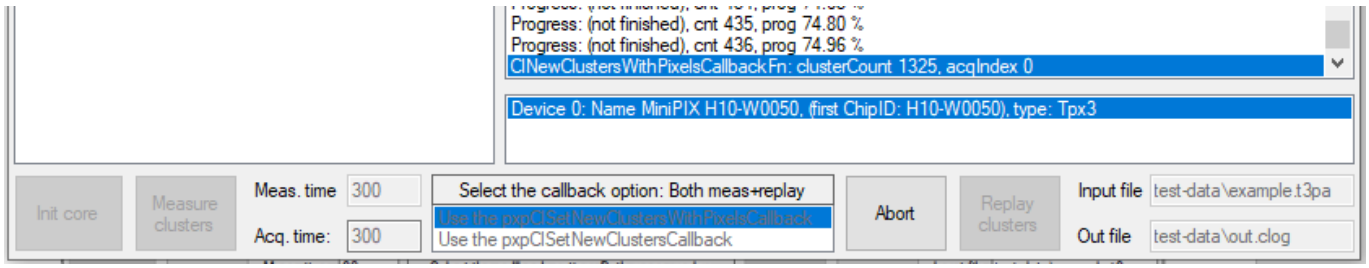
1. Message: err=0, Processing file filename…
2. Progress with finish false (0 to more occurance)
3. NewClusters or NewClustersWithPixels (0 to more occurance)
4. Progress with finish true

5.2. Clustering examples

This chapter containing parts of example projects clustering-1 and clustering-2 from the AdvacamAPIexamples collection. The projects are almost the same, differing only in the initialization and the ability to select devices in the second version.



The clustering-1 screenshot



The clustering-2 difference

5.2.1. Initializing using the pxpClLoadPixetCore

```
// (simplest with blind using of the device with index 0)

int rc = pxpClLoadPixetCore("pxcore.dll");
errorToList("pxpClLoadPixetCore", rc);

clHandle = pxpClCreate(0);
if (clHandle==CL_INVALID_HANDLE) msgToList("pxpClCreate INVALID");
else  msgToList("pxpClCreate OK");

// error messages
rc = pxpClSetMessageCallback(clHandle, ClMessageCallbackFn, NULL);
errorToList("pxpClSetMessageCallback", rc);

// progress %, is finished?
rc = pxpClSetProgressCallback(clHandle, ClProgressCallbackFn, NULL);
errorToList("pxpClSetProgressCallback", rc);

// occurs at every single acquisition is started
rc = pxpClSetAcqStartedCallback(clHandle, ClAcqStartedCallbackFn, NULL);
errorToList("pxpClSetAcqStartedCallback", rc);

// occurs at every single acquisition is finished
rc = pxpClSetAcqFinishedCallback(clHandle, ClAcqFinishedCallbackFn, NULL);
errorToList("pxpClSetAcqFinishedCallback", rc);

// arriving data containing properties of the clusters with all their pixels
rc = pxpClSetNewClustersWithPixelsCallback(clHandle, ClNewClustersWithPixelsCallbackFn, NULL);
errorToList("pxpClSetNewClustersCallback", rc);
```

5.2.2. Initializing from the active Pixet core

```
msgToList("Initializing the Pixet core...");
btnInit->Enabled = 0;
int rc = pxcInitialize();
errorToList("pxcInitialize", rc);

// (omited test of device count is > 0)

iPixet = pxcGetIPixet();
if (iPixet==0) msgToList("pxcGetIPixet=NULL"); else  msgToList("pxcGetIPixet OK");
// Warning: Use the pxcGetIPixet from pxcapi.h, not pxpClGetIPixet from clusteringapi.h

pxpClSetIPixet(iPixet);

// (omited feeding of the device list)

deviceIndex = listDevices->SelectedIndex;
msgToList("Selected device index: " + deviceIndex.ToString());

clHandle = pxpClCreate(deviceIndex);
if (clHandle==CL_INVALID_HANDLE) msgToList("pxpClCreate INVALID");
else  msgToList("pxpClCreate OK");

// (omited setting up the callbacks)
```

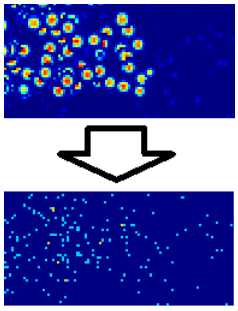
5.2.3. Using the NewClusters callback

```
// Callback "new clusters" (without each pixel data) - The process: Display some statistics
void ClNewClustersCallbackFn(PXPCluster* clusters, size_t clusterCount, size_t acqIndex, void*
userData) {
	printToBuffer("clusterCount %d, acqIndex %d", (int)clusterCount, (int)acqIndex);

	float maxClE = 0.0, maxClH = 0.0;
	unsigned short maxClsiz = 0;
	for (unsigned ci = 0; ci<(unsigned)clusterCount; ci++) {
		if (clusters[ci].energy>maxClE) maxClE = clusters[ci].energy;
		if (clusters[ci].height>maxClH) maxClH = clusters[ci].height;
		if (clusters[ci].size>maxClsiz) maxClsiz = clusters[ci].size;
	}
	printToBuffer("^ max single cluster size %d px, max energy %.2f MeV, max height %.2f MeV",
maxClsiz, maxClE/1000.0, maxClH/1000.0);
}
```

5.2.4. Using the NewClustersWithPixels callback

```
// Callback "Clusters with pixels" - The process: Find and display biggest cluster
void ClNewClustersWithPixelsCallbackFn(PXPClusterWithPixels* clusters, size_t clusterCount,
size_t acqIndex, void* userData) {
	printToBuffer("clusterCount %d, acqIndex %d", (int)clusterCount, (int)acqIndex);

	float maxClE = 0.0;
	for (unsigned ci = 0; ci<(unsigned)clusterCount; ci++) {
		if (clusters[ci].energy>maxClE) maxClE = clusters[ci].energy;
		if (clusters[ci].size>maxClusterSizeFound) {
			maxClusterSizeFound = clusters[ci].size;
			maxSizedClusterEnergy = clusters[ci].energy;
			maxSizedClusterToa = clusters[ci].toa;
			printToBuffer("Larger cluster found: size %d px, E %.2f MeV, ToA %.2E s",
				clusters[ci].size, clusters[ci].energy/1000.0,
				clusters[ci].toa/1000000000.0);
			for (unsigned n = 0; n<cDevPixels; n++) {
				rawData1[n] = 0; rawData2[n] = 0;
			}
			for (unsigned pi = 0; pi<(unsigned)clusters[ci].size; pi++) {
				unsigned adr =
				   clusters[ci].pixels[pi].x + cDevXsize * clusters[ci].pixels[pi].y;
				rawData1[adr] = clusters[ci].pixels[pi].toa - clusters[ci].toa;
				rawData2[adr] = clusters[ci].pixels[pi].energy;
			}
			newDataToView = true;
		}
	}
}
```

6. The SpectraImg and related



The SpectraImg is designed for easy working with an energy spectras. It can work with previous saved data using the **pxpClReplayData** or with physical device using the **pxpClStartMeasurement**. Measured data will be clusterized, clusters will be will be divided into the required number of channels. Next you can use some frame-generating functions and data will be filtered by applied criteria and converted to pixels on output images. Or use graph-generating functions to generate graphs.

6.1. The SpectraImg API

This API is defined in the spectraimgapi.h file. Tyical usage:

Measuring, processing and use of the processed data:

1.a Load the Pixet core using **pxpSiLoadPixetCore**("pxcore.dll");
1.b Or normally start the application using the Pixet core API pxcInitialize function, get the core pointer using **pxcGetIPixet**(), set the core pointer to the SpectraImg API using **pxpSiSetIPixet**(iPixet);
2. Get the SpectraImg instance using **pxpSiCreate**(device_index);
3. Set-up the callbacks (not required).
4. Load device calibration (not required).
5. Set the measurement parameters using the **pxpSiSetMeasParams**.
6. Set the X-ray fluorescence compensation parameters (not required) **pxpSiSetXrfCorrectionParams**.
7.a Start the measurement using **pxpSiStartMeasurement**.
7.b Or replay old data using the **pxpSiReplayData** function.
8. Wait for measurement and processing is complete (and display the progress) using while-**pxpSiIsRunning**.
9. Use some get… method and use the processed data.
10. Deinitialize the Pixet core.

Using BSTG files to save processing time:

1. After processing is complete (end of waiting steps above), use the **pxpSiSaveToFile(**handle, "file.bstg"**)** method.
2. Anytime later use the **pxpSiLoadFromFile(**handle, "file.bstg"**)**.
3. Use the **pxpSiGetMeasParams** to evaluate measure settings or set it to your program variables or GUI
4. Continue using the data as it was processed. The program now is in the step 9 of the list above.

6.1.1. LoadPixetCore and UnloadPixetCore

```
PXSIAPI int pxpSiLoadPixetCore(const char* pxCoreLibPath);
```

Loads the pixet core library (pxcore.dll/so). When the measurement with a device is intented the user has to either load pixet core with this function, or if the core is already loaded in the application (pxcore.dll/so was loaded separatelly), the setIPixet function has to be called.

Example:

```
int rc = pxpSiLoadPixetCore("pxcore.dll");
errorToList("pxpSiLoadPixetCore", rc);
```

```
PXSIAPI void pxpSiUnloadPixetCore();
```

Deinitializes ane unloads the Pixet core.

6.1.2. SetIPixet and GetIPixet

```
PXSIAPI void pxpSiSetIPixet(void* pixet);
```

Sets the internal Pixet API pointer. This is used when pxcore library is loaded separatelly in application. The use must pointer obtained via function pxcGetIPixet and should not load the pixet core with pxpSiLoadPixetCore function.

Example:
```
iPixet = pxcGetIPixet(); // Warning: Use the pxcGetIPixet from pxcapi.h,
                         // not pxpSiGetIPixet from spectraimgapi.h
if (iPixet==0) msgToList("pxcGetIPixet=NULL"); else  msgToList("pxcGetIPixet OK");
pxpSiSetIPixet(iPixet);
```

```
PXSIAPI void* pxpSiGetIPixet();
```

Return internal Pixet structure pointer or 0 if not set.
Can be used if you want use functions of the pxcore API if the program was started using pxpClLoadPixetCore.

Warning: Do not confuse this with the **pxcGetIPixet()** function of the pxcore API.

6.1.3. Create and Free

```
PXSIAPI sihandle_t pxpSiCreate(int deviceIndex=SI_NO_DEVICE);
```

Creates a new instance of SpectraImg.

deviceIndex index of the device this SpectraImg instance will manage.
If used offline (pxcore library not loaded), use SI_NO_DEVICE. The measurement will be not possible. Only replaying of data.
Note: If no device present, device with idx 0 is virtual file device. This is second way to offline use.

return Returns the handle of newly create instance of Spectra Imaging, or SI_INVALID_HANDLE if error

Examle:
```
siHandle = pxpSiCreate(0);
if (siHandle==CL_INVALID_HANDLE) msgToList("pxpClCreate INVALID");
else  msgToList("pxpClCreate OK");
```

```
PXSIAPI int pxpSiFree(sihandle_t handle);
```

Frees the created instance of SpectraImg.

handle SpectraImg handle

return 0 if OK, otherwise error code (SI_ERR_XXX)

6.1.4. GetLastError

```
PXSIAPI int pxpSiGetLastError(sihandle_t handle, char* errorMsgBuffer, unsigned size);
```

Gets the last error message - when a function returns error code. Gets either global message when handle = 0, or message for the specific SpectraImg instance. You can use this function or the message callback.

handle SpectraImg handle received from function pxpSiCreate

errorMsgBuffer output buffer where the error message will be stored

size size of the supplied errorMsgBuffer

return 0 if OK, otherwise error code (SI_ERR_XXX)

6.1.5. Load calibration functions and test if calibration loaded

```
PXSIAPI int pxpSiLoadCalibrationFromDevice(sihandle_t handle);
```

Loads the calibrations from the physically connected device. The device must support this feature. Minipix Tpx3 for example. Returns 0 if OK or error code (SI_ERR_XXX).

```
PXSIAPI int pxpSiLoadCalibrationFromFiles(sihandle_t handle, const char* filePaths);
```

Loads the calibration files (a,b,c,t files or a single xml device config file).
Returns 0 if OK or error code (SI_ERR_XXX).

Examples:

```
pxpClLoadCalibrationFromFiles("mydetector.xml");
pxpClLoadCalibrationFromFiles("calibA.txt|calibB.txt|calibC.txt|calibT.txt");
```

```
PXSIAPI int pxpSiIsCalibrationLoaded(sihandle_t handle);
```

Returns > 0 is loaded, = 0 not loaded, < 0 error

6.1.6. SetMeasParams, GetMeasParams and SetXrfCorrectionParams

```
PXSIAPI int pxpSiSetMeasParams(sihandle_t handle, int spectFrom, int spectTo, double spectStep,
bool maskNoisyPixels, bool doSubPixCorrection, bool correctXrf);
```

Sets the parameters of the future measurement and processing

handle	Spectra imaging instance handle
spectFrom	minimal spectrum energy
spectTo	maximal spectrum energy
spectStep	step of energies in the selected energy range
maskNoisyPixels	if measuring, periodically checks for noisy pixels and masks them
doSubPixCorrection	if special filter should be applied to sub pixel images
correctXrf	if the CdTe sensor internal fluorescence should be corected
return	0 if OK, otherwise error code (SI_ERR_XXX)

```
PXSIAPI int pxpSiGetMeasParams(sihandle_t handle, int* spectFrom, int* spectTo, double*
spectStep, bool* maskNoisyPixels, bool* doSubPixCorrection, bool* correctXrf);
```

Reads the processing parameters of the data in the SpectraImg instance memory. Returns 0 if OK, or error code. (The parameters are pointers to output variables with the some meaning as the pxpSiSetMeasParams parameters) Usesfull after loading a BSTG file.

```
PXSIAPI int pxpSiSetXrfCorrectionParams(sihandle_t handle, double minVol, double maxVol, double
toaDiff, bool remove);
```

Sets the paramaters of the CdTe XRF correction in the future measurement and processing

handle	Spectra imaging instance handle
minVol	minimal energy volume of event to be considered as XRF
maxVol	maximal energy volume of event to be considered as XRF
toaDiff	maximal toa difference between primary and secondary XRF event
remove	if XRF events should be removed (true) or assigned to original event (false)
return	0 if OK, otherwise error code (SI_ERR_XXX)

6.1.7. StartMeasurement, ReplayData, IsRunning and Abort

```
PXSIAPI int pxpSiStartMeasurement
(sihandle_t handle, double acqTime, double measTime, const char* outputFilePath, bool processData);
```

Starts measuremnt with the device for specified time and process the data. Only if the SI connected to the device. If calibration is loaded, energy values will be calibrated. Measurement works in the background. Use while-**pxpSiIsRunning**() to wait for end, if need it.

handle Spectral imaging instance handle

acqTime acquisition time of a single frame / pixel measurement in seconds. Primary for **frame-based devices** (Medipixes, Timepix, no Tpx3): This is single frame time. Use a short enough time to prevent clusters overlapping. Too short time can cause too many losses between frames. On **data-driven devices** (Timepix3, no Timepix), this is the ToA limit. After exceeds, ToA is resets and acqIndex in the newClusters… callbacks is incremented. acqTime=measTime can be used.

measTime total time of measurement in seconds. Use 0 to endless measurement (progress will always 100%).

outptuFilePath output file where the process data (clusters) will be saved (*.clog). If saving not required, put "".

processData if the measured data should be processed online (clustering, filtering).

return 0 if OK, otherwise error code (SI_ERR_XXX)

Example:

```
// measure for 100 secs, output file OFF, process the data ON
rc = pxpSiStartMeasurement(siHandle, 100, 100, "", true); // tpx3 version: acq=meas or acq<meas
//   pxpSiStartMeasurement(siHandle, 0.1, 100, "", true); // tpx/mpx version: acq safe time
errorToList("pxpSiStartMeasurement", rc);
```

```
PXSIAPI int pxpSiReplayData(sihandle_t handle, const char* filePath, const char* outputFilePath);
```

Replays and process already measured data files (*.pmf, *.txt, *.t3r, *.t3pa, ...). If calibration is loaded, energy values will be calibrated. If the output path defined and ending with .clog, cluster log will be saved.

handle Spectral imaging instance handle

filePath full path to a data file to be replayed

outptuFilePath output file where the process data (clusters) will be saved (*.clog). If saving not required, put "".

return 0 if OK, otherwise error code (SI_ERR_XXX)

Clog files note:

For historical reasons, there are two **CLOG** formats. Tpx3 have one additional column. The **pxpSiReplayData** cannot replay CLOG form Tpx3. Use the **T3PA** instead the CLOG. The T3PA files can be generated by data-driven measuring in the pxcore API.

Example:

```
// replay data from a file
rc = pxpSiReplayData(siHandle, "testdata.t3pa", ""); // tpx3 version
//   pxpSiReplayData(siHandle, "testdata.clog", ""); // tpx/mpx version
errorToList("pxpSiReplayData", rc);
```

```
PXSIAPI int pxpSiIsRunning(clhandle_t handle);
```

Returns 1 if running, 0 = not running, < 0 error.

```
PXSIAPI int pxpSiAbort(clhandle_t handle);
```

Aborts the measurement or replaying of the data. Returns 0 if OK or error code (SI_ERR_XXX).

6.1.8. BSTG files: pxpSiSaveToFile and pxpSiLoadFromFile

These functions allow you to save CPU time: Save the processed data with all settings and reuse it in future.

```
PXSIAPI int pxpSiSaveToFile(sihandle_t handle, const char* filePath);
```

Saves the measurement, processing results and settings to a file
handle Spectral imaging instance handle
filePath full path to a measurement will be saved
return 0 if OK, otherwise error code (SI_ERR_XXX)

```
PXSIAPI int pxpSiLoadFromFile(sihandle_t handle, const char* filePath);
```

Loads the saved measurement including settings and processed data from a file
handle Spectral imaging instance handle
filePath full path to the saved measurement file (*.bstg)
return 0 if OK, otherwise error code (SI_ERR_XXX)

Example:

```
int    rc;
int    spectFrom = -1;
int    spectTo = -1;
double spectStep = -1;
bool   maskNoisyPixels, doSubPixCorrection, correctXrf;

rc = pxpSiLoadFromFile(siHandle, "testfile.bstg");
errorToList("pxpSiLoadFromFile", rc);

rc = pxpSiGetMeasParams(siHandle, &spectFrom, &spectTo, &spectStep,
                        &maskNoisyPixels, &doSubPixCorrection, &correctXrf);
errorToList("pxpSiGetMeasParams", rc);

msgToList(String::Format("From {0}, To {1}, Step {2}, Mask {3}, SubPix {4}, XRF {5}",
spectFrom, spectTo, spectStep, maskNoisyPixels, doSubPixCorrection, correctXrf));
```

6.1.9. Callbacks: SetMessageCallback and SetProgressCallback

```
PXSIAPI int pxpSiSetMessageCallback
                        (clhandle_t handle, SiMessageCallback callback, void* userData);
```

Callback for messages and error messages returned from the SDK.

```
typedef void (*SiMessageCallback)(bool error, const char* message, void* userData);
```

error true if error message
message text of the message
userData pointer to data of the user that were set in set callback function

```
PXSIAPI int pxpSiSetProgressCallback
                        (clhandle_t handle, SiProgressCallback callback, void* userData);
```

Callback for progress of an operation. Occurs every 1 second while measuring or processing. If measuring and processing, occurs twice.

```
typedef void (*SiProgressCallback)(bool finished, double progress, void* userData);
```

finished true if operation finished
progress percentage of progress 0 - 100
userData pointer to data of the user that were set in set callback function

6.1.10. ProcessedPixelsPerSecond and MeasuredPixelsPerSecond

```
PXSIAPI double pxpSiMeasuredPixelsPerSecond(sihandle_t handle);
```

Returns the number of measured pixels per seconds.
handle Spectral imaging instance handle
return if positive, processed pixel count per second, if negative error code (SI_ERR_XXX)

```
PXSIAPI double pxpSiProcessedPixelsPerSecond(sihandle_t handle);
```

Returns the number of processed pixels per seconds
handle Spectral imaging instance handle
return if positive, processed pixel count per second, if negative error code (SI_ERR_XXX)

6.1.11. Progress callback with px/sec example

```
void ProgressCallbackFn(bool finished, double progress, void* userData) {
	char *fin = "(working)", *finF = "(finished)";
	static int cnt = 0;

	double mpps = pxpSiMeasuredPixelsPerSecond(siHandle);
	double ppps = pxpSiProcessedPixelsPerSecond(siHandle);
	if (finished) fin = finF;
	sprintf(clbStatText, "Progress: %s, cnt %d, prog %.2f %%, measured: %.2f px/s, processed:
%.2f px/s", fin, cnt, progress, mpps, ppps);
	cnt++;
	// acquisition and processing both generating this callback
	// (double cnt per new percents while measuring, single while offline processing)
	if (finished) {
		cnt = 0; // This not occurs if error occured and process failed.
	}
}
```

6.1.12. Functions that using the output data

```
PXSIAPI int pxpSiSpectrumSize(sihandle_t handle);
```

Returns spectrum size
handle Spectral imaging instance handle
return if positive, processed pixel count per second, if negative error code (SI_ERR_XXX)

```
PXSIAPI int pxpSiSaveDataAsFramesToFile(sihandle_t handle, const char* filePath, bool oneFile);
```

Saves generated spectra images (for each energy) to frame files.
handle Spectral imaging instance handle
fielPath full path of a base file name.
Each frame file will have and number suffix corresponding to the energy bin.
oneFile whether the images should be saved into a single multi frame file (*.pmf)
return 0 if OK, otherwise error code (SI_ERR_XXX)

```
PXSIAPI int pxpSiSaveDataAsSpectrumToFile(sihandle_t handle, const char* filePath);
```

Saves generated spectra images as a list of spectra for each pixel (file with 65536 spectra)
handle Spectral imaging instance handle
filePath full path to the file where spectra will be saved
return 0 if OK, otherwise error code (SI_ERR_XXX)

```
PXSIAPI int pxpSiGetFrameForEnergy(sihandle_t handle, unsigned energyIndex, bool sumFrame, bool
normalize, int zoom, double* frameData, size_t* width, size_t* height);
```

Gets the data of the image (frame) for a selected energy index or sum image if selected.
handle Spectral imaging instance handle
energyIndex index of the energy bin to get the frame. If sum frame selected, this is ignored.
sumFrame if sum frame from all energy bins should be returned
zoom zoom factor (1, 2, or 3) for sub pixel frame. Only avaiable when sum frame is selected
normalize whether the image should be normalized by the spectrum
frameData buffer where the frame will be saved
width [in/out] pointer to the width of the frame. *
height [in/out] pointer to the height of the frame. *
return 0 if OK, otherwise error code (SI_ERR_XXX)

* Width and height are for energy bin frame and single detector = 256.
If SumFrame used and with zoom factor, then 256 * 2^zoom.

```
PXSIAPI int pxpSiGetFrameForEnergyRange(sihandle_t handle, unsigned energyIndexFrom, unsigned
energyIndexTo, bool normalize, double* frameData, size_t* width, size_t* height);
```

Gets the data of the image (frame) for a selected energy range (frame is sum of frames for each energy bin).
handle Spectral imaging instance handle
energyIndexFrom first energy index in the range
energyIndexTo last energy index in the range
frameData buffer where the frame will be saved
normalize whether the image should be normalized by the spectrum
width, height [in/out] pointers to the width and height of the frame
return 0 if OK, otherwise error code (SI_ERR_XXX)

```
PXSIAPI int pxpSiGetGlobalSpectrum(sihandle_t handle, unsigned* data, double* step, size_t* size);
```

Gets the global energy spectrum

handle	Spectral imaging instance handle
data	data buffer where the spectrum will be stored
step	step of the spectra
size [in/out]	pointer to the size of the supplied buffer. if buffer small, size will contain correct size. Correct spectrum size can be obtained via **pxpSiSpectrumSize** function.
return	0 if OK, otherwise error code (SI_ERR_XXX)

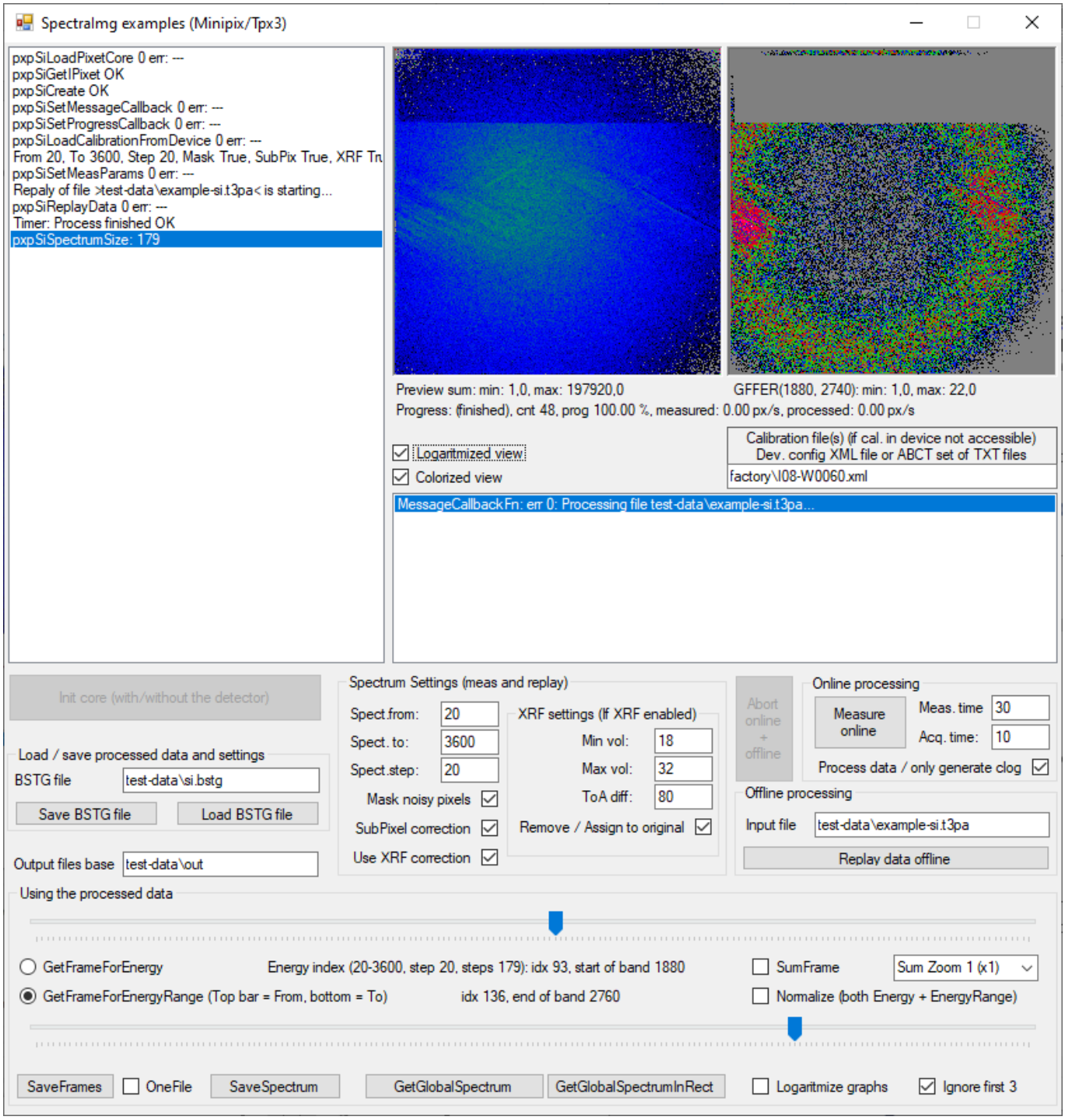
```
PXSIAPI int pxpSiGetGlobalSpectrumInRect(sihandle_t handle, unsigned x1, unsigned y1, unsigned x2, unsigned y2, unsigned* data, double* step, size_t* size);
```

Gets the global energy spectrum in selected frame rectangle

handle	Spectral imaging instance handle
x1	left coordinate of the rectangle
y1	top coordinate of the rectangle
x2	right coordinate of the rectangle
y2	bottom coordinate of the rectangle
data	data buffer where the spectrum will be stored
step	step of the spectra
size	size of the supplied buffer. if buffer small, size will contain correct size. Correct size is calculated as round((spectTo-spectFrom)/spectStep)
return	0 if OK, otherwise error code (SI_ERR_XXX)

6.2. SpectraImg examples

This chapter containing parts of example project SpectraImg from the AdvacamAPIexamples collection.



6.2.1. Simple measuring and show the spectrum

```
int rc;
size_t spSiz;
unsigned *data;

rc = pxpSiLoadCalibrationFromDevice(siHandle);
errorToList("pxpSiLoadCalibrationFromDevice", rc)
if (rc!=0) msgToList("Calibration failed! - ToT counts will be instead of energies");

// from 40, to 4040, step 20 keV, mask noisy pixels ON, subpix corr in sum OFF, XRF corr OFF
rc = pxpSiSetMeasParams(siHandle, 40, 4040, 20, true, false, false);
errorToList("pxpSiSetMeasParams", rc);

// measure for 100 secs, output file OFF, proces the data ON
rc = pxpSiStartMeasurement(siHandle, 100, 100, "", true); // tpx3 version: acq=meas or acq<meas
//   pxpSiStartMeasurement(siHandle, 0.1, 100, "", true); // tpx/mpx version: acq safe time
errorToList("pxpSiStartMeasurement", rc);

// wait for measure complete
while(pxpSiIsRunning(siHandle)) System::Threading::Thread::Sleep(100);

// number of elements in the spectrum = (4040-40)/20 in this example
spSiz = pxpSiSpectrumSize(siHandle);
if (spSiz>=0) msgToList("pxpSiSpectrumSize: "+ spSiz.ToString());
else { errorToList("pxpSiSpectrumSize", spSiz); return; }

data = (unsigned*)malloc(spSiz * sizeof(unsigned));
if (data==NULL) { msgToList("malloc error!"); return; }

// get the spectrum to the "data" array
rc = pxpSiGetGlobalSpectrum(siHandle, data, &spectStep, &spSiz);
errorToList("pxpSiGetGlobalSpectrum", rc);
if (rc<0) { free(data); return; }

// show the graph (internal function of the example project)
graph(data, spSiz, gcnew Pen(Color::Black, 3.0), GRAPH_clear);
free(data);
```

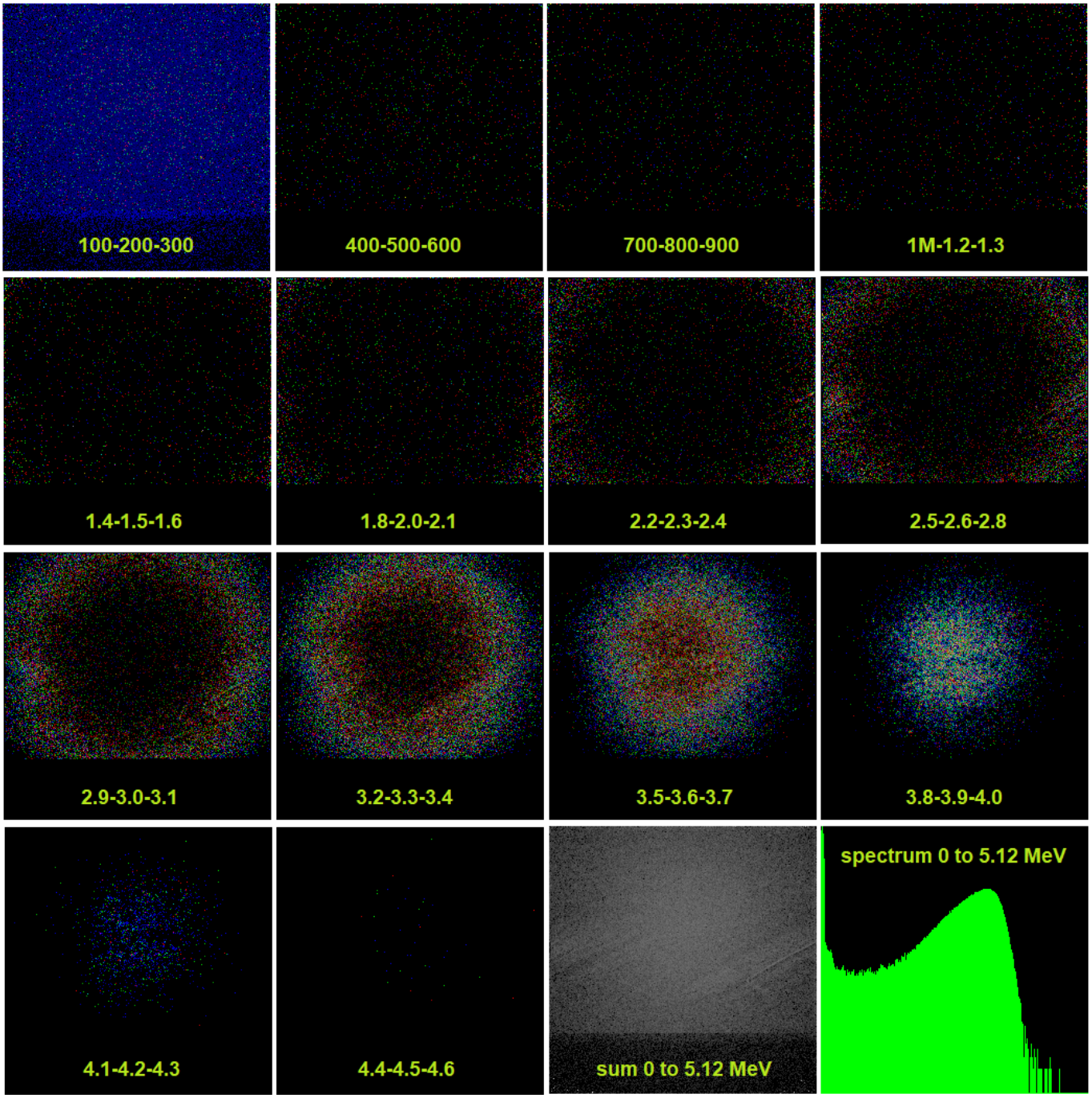
6.2.2. Measuring (replay) and getFrameForEnergy

```
int rc;
size_t w=256, h=256;
double *data;

rc = pxpSiLoadCalibrationFromDevice(siHandle);
errorToList("pxpSiLoadCalibrationFromDevice", rc)
if (rc!=0) msgToList("Calibration failed! - ToT counts will be instead of energies");

// from 40, to 4040, step 20 keV, mask noisy pixels ON, subpix corr in sum OFF, XRF corr OFF
rc = pxpSiSetMeasParams(siHandle, 40, 4040, 20, true, false, false);
errorToList("pxpSiSetMeasParams", rc);

// measure for 100 secs, output file OFF, proces the data ON
rc = pxpSiStartMeasurement(siHandle, 100, 100, "", true); // tpx3 version: acq=meas or acq<meas
//   pxpSiStartMeasurement(siHandle, 0.1, 100, "", true); // tpx/mpx version: acq safe time
errorToList("pxpSiStartMeasurement", rc);

// replay data from file – alternative to pxpSiStartMeasurement
// rc = pxpSiReplayData(siHandle, "testdata.t3pa", ""); // tpx3 version
// rc = pxpSiReplayData(siHandle, "testdata.clog", ""); // tpx/mpx version
// errorToList("pxpSiReplayData", rc);

// wait for measure complete
while(pxpSiIsRunning(siHandle)) System::Threading::Thread::Sleep(100);

data = (unsigned*)malloc(w * h * sizeof(double));
if (data==NULL) { msgToList("malloc error!"); return; }

// pxpSiGetFrameForEnergy(handle, energyIndex, sumFrame, normalize, zoom, frameData, w, h);
rc = pxpSiGetFrameForEnergy(siHandle, 1, false, false, false, data, &w, &h);
errorToList("pxpSiGetFrameForEnergy", rc);
if (rc<0) { free(data); return; }

// normalize data and show the image with caption (internal function of the example project)
viewFrame(data, String::Format("GetFrameForEnergy {0} keV", 40 + 1*20));
free(data);
```

6.2.3. Measuring (replay) and getFrameForEnergyRange

```
int rc;
size_t w=256, h=256;
double *data;

rc = pxpSiLoadCalibrationFromDevice(siHandle);
errorToList("pxpSiLoadCalibrationFromDevice", rc)
if (rc!=0) msgToList("Calibration failed! - ToT counts will be instead of energies");

// from 40, to 4040, step 20 keV, mask noisy pixels ON, subpix corr in sum OFF, XRF corr OFF
rc = pxpSiSetMeasParams(siHandle, 40, 4040, 20, true, false, false);
errorToList("pxpSiSetMeasParams", rc);

// measure for 100 secs, output file OFF, proces the data ON
rc = pxpSiStartMeasurement(siHandle, 100, 100, "", true); // tpx3 version: acq=meas or acq<meas
//   pxpSiStartMeasurement(siHandle, 0.1, 100, "", true); // tpx/mpx version: acq safe time
errorToList("pxpSiStartMeasurement", rc);

// replay data from file alternative
// rc = pxpSiReplayData(siHandle, "testdata.t3pa", ""); // tpx3 version
// rc = pxpSiReplayData(siHandle, "testdata.clog", ""); // tpx/mpx version
// errorToList("pxpSiReplayData", rc);

// wait for measure complete
while(pxpSiIsRunning(siHandle)) System::Threading::Thread::Sleep(100);

data = (unsigned*)malloc(w * h * sizeof(double));
if (data==NULL) { msgToList("malloc error!"); return; }

// pxpSiGetFrameForEnergy(handle, energyIndex, sumFrame, normalize, zoom, frameData, w, h);
rc = pxpSiGetFrameForEnergy(siHandle, 1, false, false, false, data, &w, &h);

// pxpSiGetFrameForEnergyRange(handle, From, To, normalize, frameData, width, height);
rc = pxpSiGetFrameForEnergyRange(siHandle, 100, 120, false, data, &w, &h);
errorToList("pxpSiGetFrameForEnergyRange", rc);
if (rc<0) { free(data); return; }

// normalize data and show the image with caption (internal function of the example project)
viewFrame(data, String::Format("GetFrameForEnergyRange {0}-{1} keV", 40 + 100*20, 40 + 120*20));
free(data);
```

6.2.4. Example results (getFrameForEnergyRange to RGB)



^{241}Am from smoke detector, located 3 mm above the chip, MinipixTpx3, CdTe 2 mm, acq. time 45 seconds.
Settings used: `setMeasParams(0, 5120, 20, 1, 0, 0) //(from, to, step, maskNP, doSPC, XRF)`

Each RGB color channel in images was generated from sum of 5 energy ranges (100 keV), using the **getFrameForEnergyRange** method, log2(val+0.5) applied and normalized to 0-255. Order is blue-green-red.
The blue at the first image is gamma byproduct 59.5 keV and noise in the first band. The images with MeV ranges shows a 5.48 MeV alpha particles attenuated in air.

Summary frame shows all hits in the set range. It was get using the **getFrameForEnergy** method with sumFrame=true. A “scrathes” on the picture: SpectraImg is very sensitive to small differences in px sensitivity.

The spectrum on last image was get using the **getGlobalSpectrum** method and processed like as pixels data, include using log2. A gaps on the right are caused by no hits.