GeoPIXE Software Organization

This doc provides a quick sketch of GeoPIXE implementation under IDL, after notes about setup in the *IDL Development Environment* (base on Eclipse) following GitHub download.

It helps to have a working understanding of the Fundamental Parameter approach used for quantitative analysis in the workflow through GeoPIXE, and some experience with the Demo data. See the ***GeoPIXE Users Guide*** and the ***GeoPIXE Worked Examples*** PDF for worked examples with step-by-step tips. Both provide examples of the main tasks of fitting spectra to generate the ***Dynamic Analysis*** (DA) image projection matrix; using this DA matrix to process full-spectral data to deconvolute elemental components and project separated elemental images; and exploration and processing of the images to first verify their accuracy, including extracting spectra from observed features (region shapes or element-element Associations), and make corrections and then to explore their content.

Open source version

This doc describes the Open Source version. Some changes have been made to directory organization for Open Source and incorporation on GitHub.

Contents

GeoPIXE from GitHub

The archive on GitHub contains all source files and essential data and documentation. It does NOT include metadata (Workspace/.metadata directory) for the IDL DE Eclipse environment. What is missing are Eclipse project definitions and the build settings for each project. Import of the projects is easy (see below). But we must set a couple of settings regarding management of the !path in IDL. These are detailed below.

Building can be easily handled using “Builder.pro” (see below). However, you can setup Eclipse IDL build settings if you like, as described in section “Eclipse environment and organization”.

Downloading

If you are reading this, then perhaps this is done for the software… But for worked example data, you need to also download the Demo data, which is archived in the CSIRO DAP. There are two archives, a simple one (e.g. for Windows system at DOI: https://doi.org/10.25919/ff5b-wr11) and one designed to serve multiple users in a Linux workshop environment (at DOI: https://doi.org/10.25919/3yrz-7x38). See the “Read me.txt” file for details.

NOTE: See the Readme file notes on how to get the ***GeoPIXE Demo data*** for the worked example tutorials.

Install IDL

Running GeoPIXE requires at least IDL runtime support. With IDL installed, but not licensed, you can run GeoPIXE using IDL *Virtual Machine* runtime support. To program in IDL and build GeoPIXE source, you will need an IDL license.

Running GeoPIXE from the compiled 'geopixe' directory

Once IDL is installed, you can run GeoPIXE simply by double clicking on "GeoPIXE.sav" in the “geopixe” folder within the “workspace” tree. It will note the absence of a “geopixe.conf” file, but will then create one for you in your <home>/.geopixe directory.

There is also an example script for running GeoPIXE as a batch command and optionally giving it arguments equivalent to running it as part of an external workflow.

Setting up Eclipse IDLDE environment

To import all projects, use the “File->Import->General->Projects from folder or archive” menu and navigate to “Workspace” in the local downloaded GeoPIXE “Workspace” directory as the “Import source” folder. Then select all project folders and “Finish”. This will import all projects. See the section “Eclipse environment and organization” for more details of the Eclipse environment.

However, this does not import project settings for building, etc. Building can be handled using the “builder” PRO, as outlined below (see section “Building GeoPIXE”). But there are some settings that must be set now. For just the “geopixe” and “Fortran” projects, right click and select “Properties” for each. For the “IDL project properties” group, uncheck the option “update IDL path when project is opened or closed”, so these projects are not added to the path. You can also do that for the “Default” project, if you are not using that.

See below for details of the “***Eclipse environment and organization***”.
Building GeoPIXE is covered in the section below “***Building GeoPIXE***”.

IDL Environment and high-level organization

IDL

IDL provides a single application with a development environment (Eclipse) and integrated graphics and windows, and a large library of scientific routines optimized for vector and matrix operations. No separate windows system needs to be used. It is fundamentally platform independent, and we can deploy to Windows, Linux, Unix and Mac. We benefit from a paid license by having updates for each O/S upgrade as they happen and avoid complex dependencies issues.

IDL like many modern languages is interpreted at run-time. However, this is made efficient using a “complied” form of tokens in a SAV file. If code can make use of organization into vectors and matrices, then IDL is very fast and inherently multithreaded and efficient. For tedious data/bit fiddling, Fortran routines are used in GeoPIXE to do tight complex loops and benefit from an optimized compiled Fortran shared library. These must be compiled for the specific platforms (currently Windows 32 and 64 bit, Linux 32 and 64 bit and Mac).

IDL can also call out into other languages, such as python, and we use python (was 2.7 and now migrated to 3.8+) libraries for access to all Maia Mapper (MM) functionality such as ZMQ. This is not needed for normal GeoPIXE analysis. Because of incompatibilities between py2.7 and py3.8, GeoPIXE open source has been compiled for py3.8 (GeoPIXE v8.7).

Access to the MM python extensions in GeoPIXE is enabled by enabling the KVS in the "geopixe.conf" config file in a user's .geopixe directory. The python MM libs can also be accessed directly from IDL using a number of wrappers in the GeoPIXE IDL code. The MM Extensions are not provided in this open-source release.

IDL is multithreaded so that each widget program can essentially run in parallel. This used to have limitations, such as everything halting if a breakpoint was triggered in one event routine (all others would stop too). This made debugging fairly simple (unless you really wanted the other widget thread to continue). Now it works very well, with all loops running in parallel. But this can be tricky for debugging. For example, a break-point set in the timer event processing code will only stop 'that' timer event executing (and you typically delay resetting this timer again until after the debugging). However, it will not stop other timers and other widget events from continuing to be processed in the same event code. This is very handy most often (e.g. widgets remain active and functioning). But you need to have your wits about you at other times to prevent that break point being triggered multiple times.

GeoPIXE data philosophy

GeoPIXE has existed in one form or another since about 1987. It has run on a range of platforms, many of which are now extinct (e.g. DEC Station and VAX mainframes), initially in Fortran and later under IDL/PV-Wave (around 2000). It has acquired data on a range of platforms in various data formats (e.g. big/little endian byte order and IEEE or DEC floating point). However, it preserves the ability to use any modern platform to process data acquired on any current or earlier platform. This is partly enabled using IDL library routines, which maintain the long history (IDL dates back even further to the DEC PDP-8).

Furthermore, as features are added and various GeoPIXE data file formats grow and evolve, the philosophy is to maintain backward compatibility, so that the latest GeoPIXE can still read old GeoPIXE data files (e.g. spectra from the 1990's and image data from the first IDL based GeoPIXE in 2000). To achieve this, additions to data files are tagged by an increment to a **version** number (a negative Long by convention), the first value in most XDR files. Reading the data files are then conditional on the version. Most GeoPIXE files use platform independent XDR binary format. However, the raw data comes in a myriad of formats both binary and formatted. Currently, GeoPIXE recognizes 47 raw data formats from ion-beam and synchrotron laboratories around the World, including from our own CSIRO-BNL Maia detector arrays (as used in Maia Mapper).

GeoPIXE organization

GeoPIXE open source uses a central "main" dir tree for the main program. The main project is compiled to "GeoPIXE.sav", which contains the main program and can act as a library of routines for other processes. It is one of only a few projects compiled with "Resolve_all" enabled (do not enable resolve_all for plugin, device objects and wizard projects, as it will then resolve and include all GeoPIXE routines, which should only be loaded from GeoPIXE.sav at runtime). A nearby dir tree "geopixe" contains the runtime files (compiled SAV for the main program "GeoPIXE.sav" and support files for devices, plugins and wizards in subdirectories).

NOTE: The name "geopixe" of the runtime directory is assumed throughout. Please do not rename this.

GeoPIXE compiled from this tree can be called by program extensions like a library by simply "restoring" the "GeoPIXE.sav" file (most SAV files contain compiled IDL routines – however, "geopixe2.sav" contains database data only). These extensions include *Maia-Control* for control and real-time monitoring of a Maia detector (as in Maia Mapper) including real-time imaging, *Scan-List* for scan list management for Maia Mapper and a number of background processes that run behind these to retrieve data from Kandinski (running on the Maia detector Hymod processor) and the Blogger (which writes raw data from Maia and runs on the data storage server). Parallel processing for image construction using the *Dynamic Analysis* method (DA) from raw data uses this approach too. All these mostly will restore GeoPIXE.sav to access GeoPIXE library routines.

GeoPIXE also scans and loads various classes of SAV files to provide run-time plugins for (i) image analysis, (ii) spectrum analysis, (iii) background algorithms, as well as (iv) processing wizards (v) GUI extensions, and (vi) raw data import device objects. The device objects add specific functionality for input of raw data from various laboratories. The default "base" data device object provides default method fallbacks. Hence, only methods that are needed are written for a new device.

The source code is divided into a directory tree of IDL "Projects" within a workspace, with "main" as one of these. The various add-ons (plugins, wizards, devices) sit as separate projects in the workspace and are compiled separately, with the output SAV files going into the "geopixe" runtime directory. They are kept separate and compiled carefully so that each plugin SAV file does not incorporate any compiled main GeoPIXE routine. It is just assumed that any routines that the plugin needs will be already loaded when GeoPIXE is initially loaded and run.

IDL Quirk: A complication in compilation is that IDL will automatically load any SAV files that it finds, which is mostly NOT what we want. For this reason, all compiled SAV files are stored in another runtime project called "geopixe". Make sure that either "geopixe" is never "opened" in the Eclipse IDLDE environment, or uncheck the option for project "geopixe" so that it does NOT "update IDL path when project is opened or closed".

Raw Data types and conversion

Using internal knowledge of the processor being used to run GeoPIXE, which may be big or small endian, for example, and knowledge of the input raw data format, raw data is converted on the fly to the host's endian type (or floating point format). This means that we can process data collected on any platform on any other platform, without issues of endian type or floating-point format (e.g. read big endian Unix data on a little endian Windows machine or read old DEC files with Linux). This uses the GeoPIXE util routine "swap_bytes", which recursively translates the contents of any nested data structure. Similarly, the Fortran routines that are called for some device objects use a 'swap' argument to convey the byte-ordering information needed to swap bytes as needed in the raw data.

Parallel processing

Processing throughout is enhanced using parallel processing, which spawns multiple copies of GeoPIXE to tackle parts of a problem (e.g. raw data to images using the DA method). Transfer of data from GeoPIXE into the background processes is accomplished through shared memory segments. The same method is used between Maia-Control and the background processes. Once processing is complete (e.g. each part of an image has been built and saved to disk), a shared-memory flag transfers control back to the foreground GeoPIXE, which assembles the parts of the problem to construct the final images (the parts of the result are also passed back as filenames through shared memory).

Parallel processing requires an IDL Run-time license, as it uses the "execute" command and IDL Bridge objects, which are not available using the "Virtual Machine" version. Processing of large data sets benefits from parallel processing. The Maia detector processes also depend on the IDL Bridge Object.

The background processes (and parallel processes) are spawned using the *IDL Bridge object* mechanism. Upgrading from a more *ad hoc* approach of GeoPIXE managed spawned processes, the use of IDL Bridge objects simplifies run-time management and reduces the chance of orphaned processes.

However, it still provides little protection against the policy on some Linux VMs to not allocate adequate swap space and randomly kill processes when resources run low. This can leave a parallel processing session "missing" some background processes that have disappeared. New code will be needed to watch for this (e.g. heart-beat monitoring).

Wizards and plugins

Plugins are very simple, and users are encouraged to write and add them to perform very specific operations on images and spectra. Examples are provided in the "src/plugins" directory of the distribution. They have names of the form "*_image_plugin.pro" for image plugins, "*_spectrum_pluigin.pro" for spectrum plugins, in projects with names like "Image … plugin" and "Spectrum … plugin". Once compiled these become "*_image_plugin.sav" and "*_spectrum_plugin.sav". They are added to a "Process→User Plugins" menu at load time from a SAV file for each Plugin (from "geopixe/plugins" directory). By convention, the plugin is called passing a pointer 'p' which is a pointer to the image or spectrum data structure (a pointer array in the latter case, see "define(/image)" or "define(/spectrum)" routine) and an integer 'i', which is the index into the array of element image planes or spectra. Initially, at load time when all available plugin SAV files are loaded (from "geopixe/plugins" directory), each plugin is called with a single keyword argument '/title'; the plugin code must return a title for the plugin to add to GeoPIXE processing menus.

By convention, any plugin that processes/modifies all image planes (not just the plane 'i') includes a prefix "" to its title, which can be seen in the menus. See examples in the "src/plugins" dir tree.*

Wizards turn this around and are designed to call GeoPIXE windows to perform sequences of operations. The current Wizards orchestrate ***Standards*** analysis and particle ***Depth*** analysis. They send broadcast *Notify* requests to perform operations, passing a pointer to a linked list of operations to perform. Windows in GeoPIXE, which have registered the ability to process these *Notify* requests, respond and perform the work and return a reply *Notify*, passing back the pointer to the Wizard, perhaps with additional data added (see below for discussion of *Notify*, the GeoPIXE inter-window messaging approach). Wizards can be called from GeoPIXE and are added to a Wizard menu at load time from a SAV file for each Wizard (from "geopixe/wizard" directory). If a Wizard is run as a separate program, it will restore and run GeoPIXE and ensure that needed windows are open by sending out periodic 'open-test' commands in 'wizard-action' Notify requests broadcast to all windows (by the 'wizard_test_windows' routine).

Note that many core routines that all Wizards (e.g. the 'wizard_test_windows' routine) or plugins need are part of the "main" project and directory tree and compiled into and restored from "GeoPIXE.sav".

The sources for plugins and wizards are distributed with GeoPIXE so that users can potentially create new plugins, either as variations on existing ones or for new or *ad hoc* image and spectrum processing.

There are also GUI plugins. The only ones implemented at present add a tab each to the *Image Regions* window (image_table.pro), one for each plugin encountered in the plugin directory with names like "image_table_*_gui_plugin.sav"). See the source for "image_table_xfm_gui_plugin.pro" (project "GUI Image Table XFM plugin"), as a simple example, which adds a specialized export table button for coordinate output formatted for stage import at the XFM beamline at the Australian Synchrotron. In principle, this approach may be used in future to extend the user interface in other windows of GeoPIXE.

Device objects

Data device specific code is contained in *Device objects* and compiled separately. These too are loaded at run time (from "geopixe/interface" directory) and populate Device droplists to select the format of raw input data (e.g. in *Import Spectra* (import_select.pro) and *Sort EVT* (evt.pro) windows). The methods supported by Device objects manage 'read_setup' to setup for reading raw data streams, 'read_buffer", which reads buffers of raw data and returns event data vectors (e.g. for X, Y, E and flux and dwell time information), 'get_header_info' to retrieve header information either from the raw data file header or a companion file, and many more specialized methods that not all Device objects support. If a method is missing, control passes to a default method in the 'base_device' object class.

Device objects support ‘options’, which are optional parameters with matching graphical elements/controls that get drawn in “Device” tabs and frames in GeoPIXE. These allow device specific parameters to be selected by the user from the GeoPIXE GUI without GeoPIXE requiring knowledge of these parameters; they are essentially “internal” to the Device. A number of methods in a device object are used to render these ‘option’ widgets and for I/O of parameters. For example, GeoPIXE image files contain options parameter data written and read by a relevant Device Object method (e.g. see “write_geopixe_image” and “read_geopixe_image” routines, which invoke the device object methods to save/restore device-specific data to image DAI files). Further function methods return logical values, which simply reflect whether a Device supports a certain capability (e.g. ‘FlipX’, ‘ylut’ in the Maia Device and many serviced by the Base Device, such as ‘name’, ‘extension’, ‘pileup’, ‘cluster’, ‘FlipX’, ‘ylut’, ‘beam_type’, ‘get_dead_weight_mode’, …). A Device object can override a method provided by the Base object to modify its behaviour (e.g. Maia Device ‘ylut’ method).

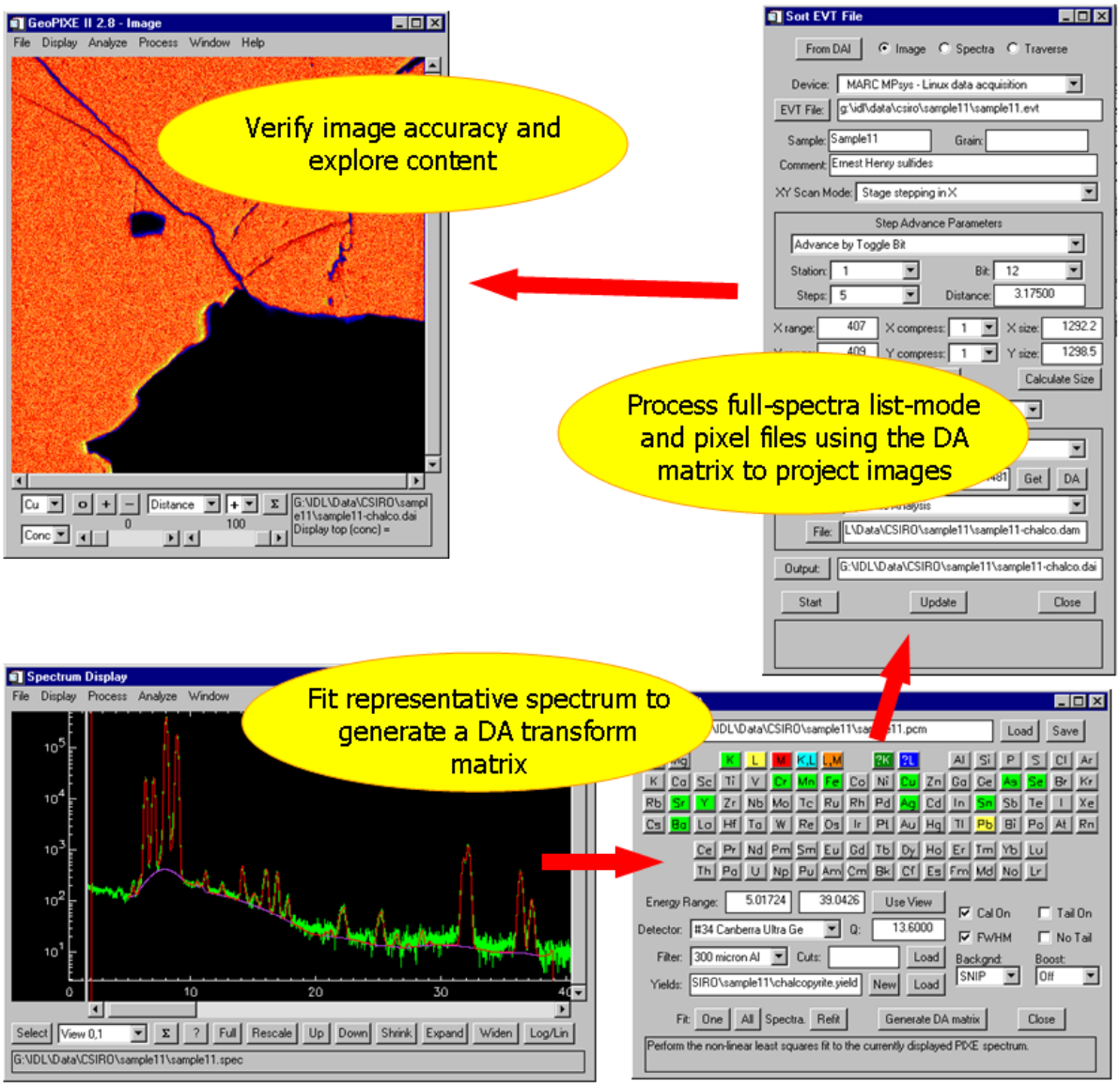
The sources for Device objects are distributed with GeoPIXE so that users can potentially create new Device objects, either as variations on existing ones or for new data formats.

GeoPIXE workflow

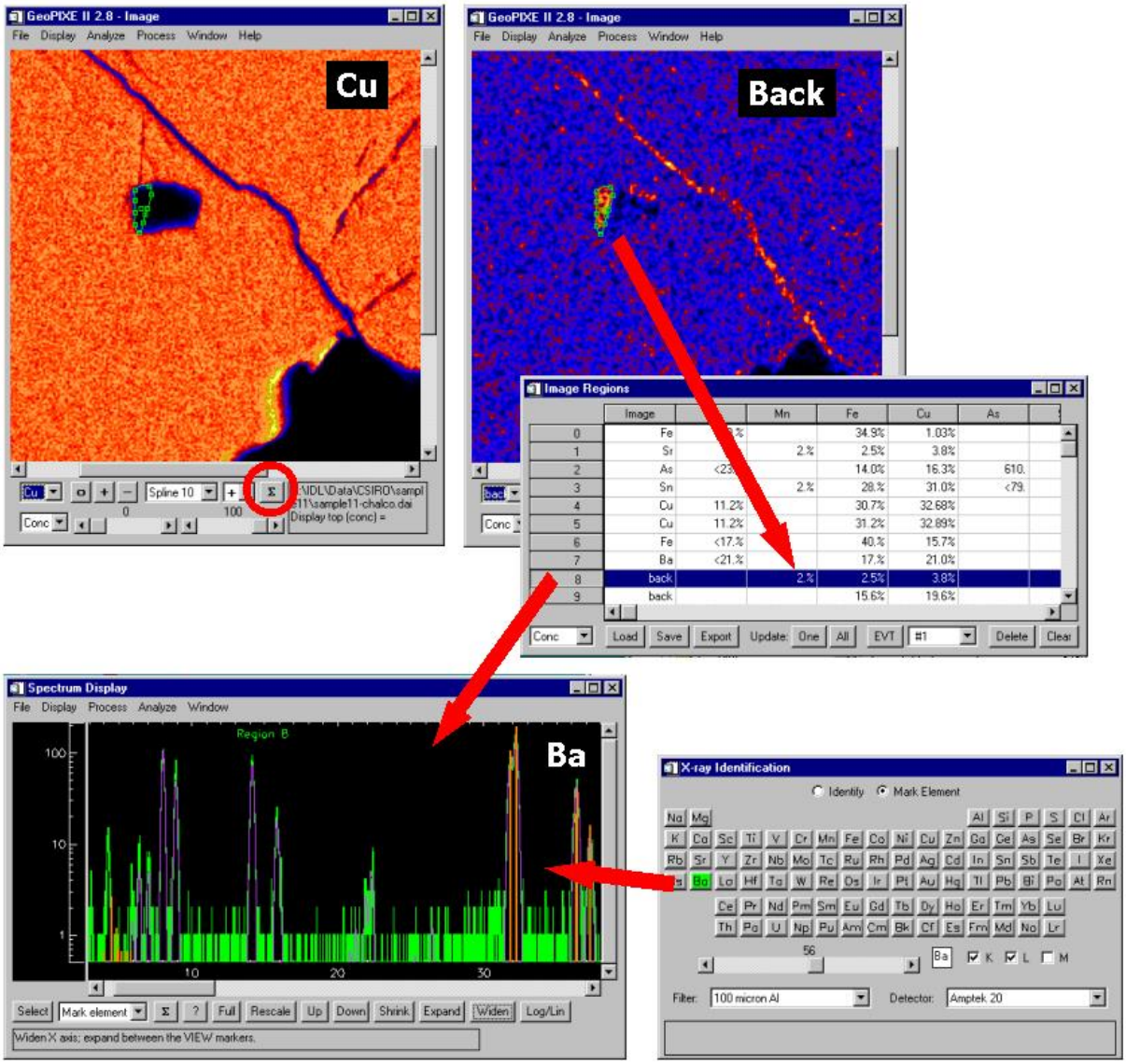
Interactive GeoPIXE

The data flow in GeoPIXE has two main phases: (i) transforming raw data into spectra or element images, and (ii) correction and exploration of these images, including extracting spectra from spatial (or concentration correlated) features seen in images (or element data). The first phase includes *Dynamic Analysis* (DA)*,* or the construction of the matrix transform used in processing raw event data, which entails modelling the physics of the interaction of the beam particles (X-rays, protons) with the sample and how they are then detected in a detector (or array of detectors). The second phase includes a host of tools for exploration and viewing data in various ways as well as a number of tools for the correction of various artefacts caused by a range of instrumental issues.

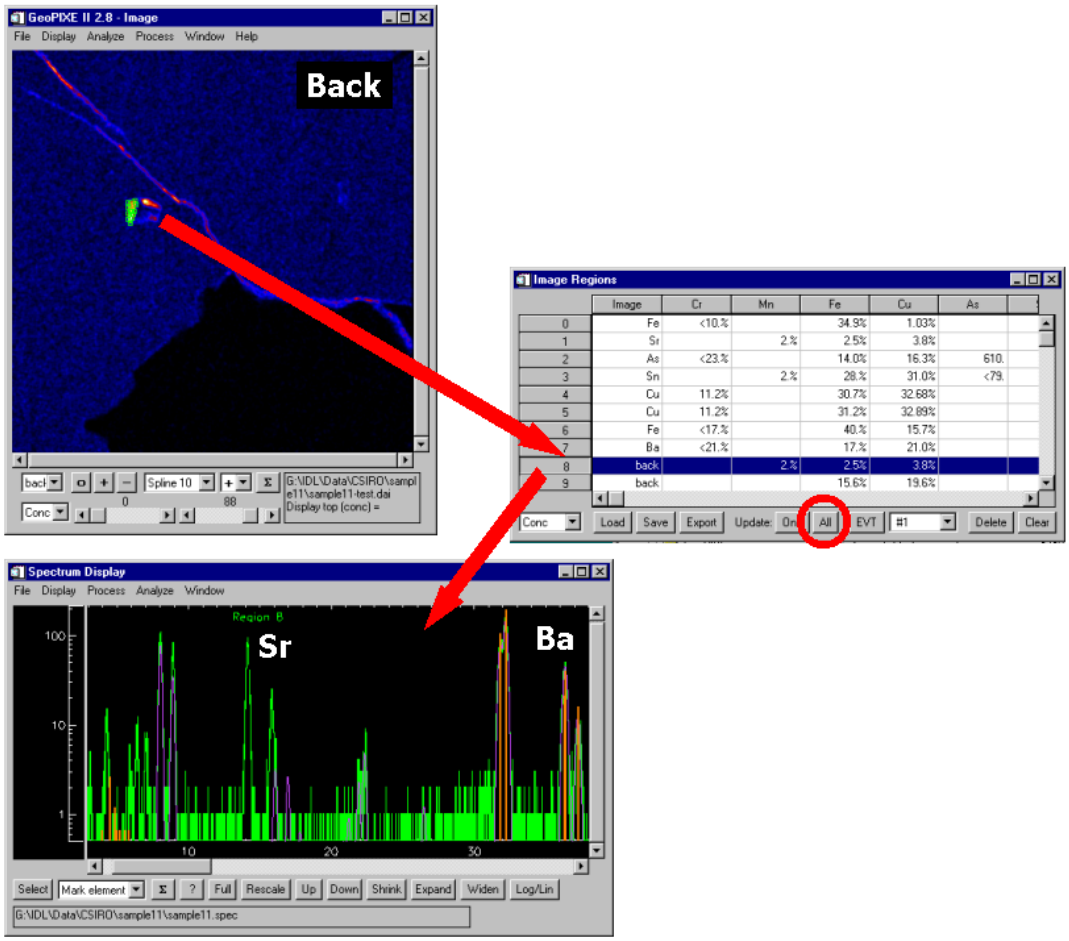
The workflow in GeoPIXE typically starts with fitting spectra (displayed in the *Spectrum Display* window (spectrum_display.pro) and fitted using the *Xray Spectrum fit* window (fit_setup.pro)) to generate the *Dynamic Analysis* (DA) image projection matrix, using this DA matrix to process full-spectral data to deconvolute elemental components and project separated elemental images (displayed in the main *GeoPIXE Image* window (gimage.pro)) and exploration and processing of the images to first verify their accuracy and make corrections or to explore their content. This involves these windows (*Spectrum Display*, *X-ray Spectrum Fit*, *Sort EVT* (evt.pro), *Image*) …



Once images are created, the focus shifts to exploring the data, which often involves selecting pixels (in the *Image Regions* window (image_table.pro)) and determining the average concentrations for these pixels and extraction of the spectra from these pixels. Pixels can be selected using a simple shape (Box, Circle, Spline curves, etc.) or by using element-element correlations in the *Associations* window (corr.pro).



A common thing to look out for is “missing” elements. In other words, an element that was not included in the fit to build the DA matrix, but later found in one of the regions, usually by examination of the spectrum extracted for the region’s pixels. Often “missing” elements show up as extra signal, or hot spots, in the “Back” image (e.g. Sr in this example – note that the Sr peaks are not “explained” by the model shown in violet because Sr was missing from the fit to generate the initial DA matrix).



See the *GeoPIXE Users Guide* section "GeoPIXE Analysis Scenarios, Data Flow" for more details and then try the Demo data, following the detailed follow-along notes in the "***GeoPIXE Worked examples***" PDF.

GeoPIXE as a callable module in another workflow

Certain repetitive data-intensive operations in GeoPIXE can be performed with GeoPIXE acting as a callable module in some other workflow, such as triggered by data acquisition. The operations supported in this way are:

(i) sorting raw data into images (i.e. as in *Sort EVT* window, evt.pro),
(ii) extraction of spectra from image pixels selected in *Image Regions*, and
(iii) export of image data (from DAI) to ZARR format for AWS data exploration and statistics.
(iv) exporting image (DAI) metadata.

GeoPIXE can now be called from the command-line of the O/S with additional parameters, which specify a "GeoPIXE Command File" and optionally input and output file-names to redirect the instructions in the command file to process the 'input' file(s) and produce the 'output' file. This then can be used in a script, for example, to call GeoPIXE to process a given (set of) input file(s) in some automation scheme, perhaps triggered by the data acquisition system following the finishing of a run.

The "geopixe" Linux script (without arguments launches GeoPIXE) accepts these new parameters ...

```
geopixe command-file input output
```

Arguments	contents

command-file	Full path to a **GeoPIXE Command File** (extension ".gcf"), which contains the name of a GeoPIXE command and all its parameters for execution, including "cluster=1" to trigger cluster-mode execution across multiple cores, if the data device supports it. Typically, enclose the GCF file-name (including full-path) in double quotes "", which enables paths with spaces to be used. The parameters in the GCF file are command dependent. The GCF files are simple ASCII text files, typically generated by GeoPIXE (e.g. "C*" buttons in the windows *Sort EVT*, *Image Regions, Image History* or the menu "*Create GeoPIXE Command File to Export as ZARR (C*)*" in the main *GeoPIXE Image* window.
input	An optional input data file-name (or list of file-names in "stringify" format) to replace the "file=" entry in the GCF file. There are three options here: 1. A simple data file-name (in "stringify" format, perhaps with a wildcard, enclosed in double quotes: "file1.*") 2. A file-name (preceded by a "@" character) of a text file (e.g. "@file"), which contains a list of all input files, one file per line. 3. A list of data file-names (in "stringify" format, i.e. as a string array or list with files enclosed in double quotes: ["file1","file2","file3", ...])
output	An optional output file-name to replace the "output=" entry in the GCF file. Generally, the 'input' and 'output' args would be set together. For 'input' file lists, the 'output' file would evolve following GeoPIXE rules for derived filenames.

Commands

imaging

The following commands generate 2D or 3D image files (extension .dai, .xan, respectively): 'da_evt' (2D image using a DA matrix), 'cut_evt' (2D image using CUTs), 'da_stack_evt' (3D image using a DA matrix and perhaps an energy table) and 'da_tomo_evt' (3D image, e.g. tomographic image). The GeoPIXE Command File can be generated using the "C*" button on the *Sort EVT* window, typically after successful processing of an initial data-set. This GCF file then provides a "template" for processing of further input data files, with the same process settings, specifying an output file for each.

region spectra extraction

The following commands generate a SPEC file containing an array of spectra: 'spec_evt' (spectra, one for each region in a REGION file). The GeoPIXE Command File can be generated using the "C*" button on the *Image Regions* window ("Extract" tab), typically after successful specifying regions on an image, for example using the *Associations* window and the Hot-spot separation tools. This GCF file then provides a "template" for processing the input data files offline, specifying an output file. In this case, the spatial regions are very image data-set specific and would not generally be applied to a different image data-set.

export

Some simple export options are supported. Using the "C*" button on the *Image History* window will produce a GCF file for metadata output from images. The *Image History* window outputs GCF that uses the '**print_image_metadata**' command. It has only one parameter, aside from the "files=" and "output=" arguments, called "stats=", which can be used to optionally enable the output of image pixel statistics for the selected input file ("stats=1"). The second is a ZARR export, '**export_zarr**', which just uses the "files=" and "output=" arguments to dump an image in ZARR format for applications in the AWS cloud. If 'files' is not found, including a local dir tree search, it will pop-up a file requester to select it. This is less useful in a work-flow environment, so make sure the file path is correct.

Batch GeoPIXE

This is a bit of a misnomer. The *Batch Sort* in GeoPIXE is a window, which allows you to queue up a number of raw data-sets to be processed into images, with options to also do image processing on each image or generate derived outputs (e.g. RGB images, HTML web page summaries). It uses the settings in *Sort EVT* as a template (after successfully processing the first data-set, for example). *Batch Sort* displays a table which shows many image parameters (e.g. X,Y size) and input and output file names and paths, and has buttons to enable image processing and output options.

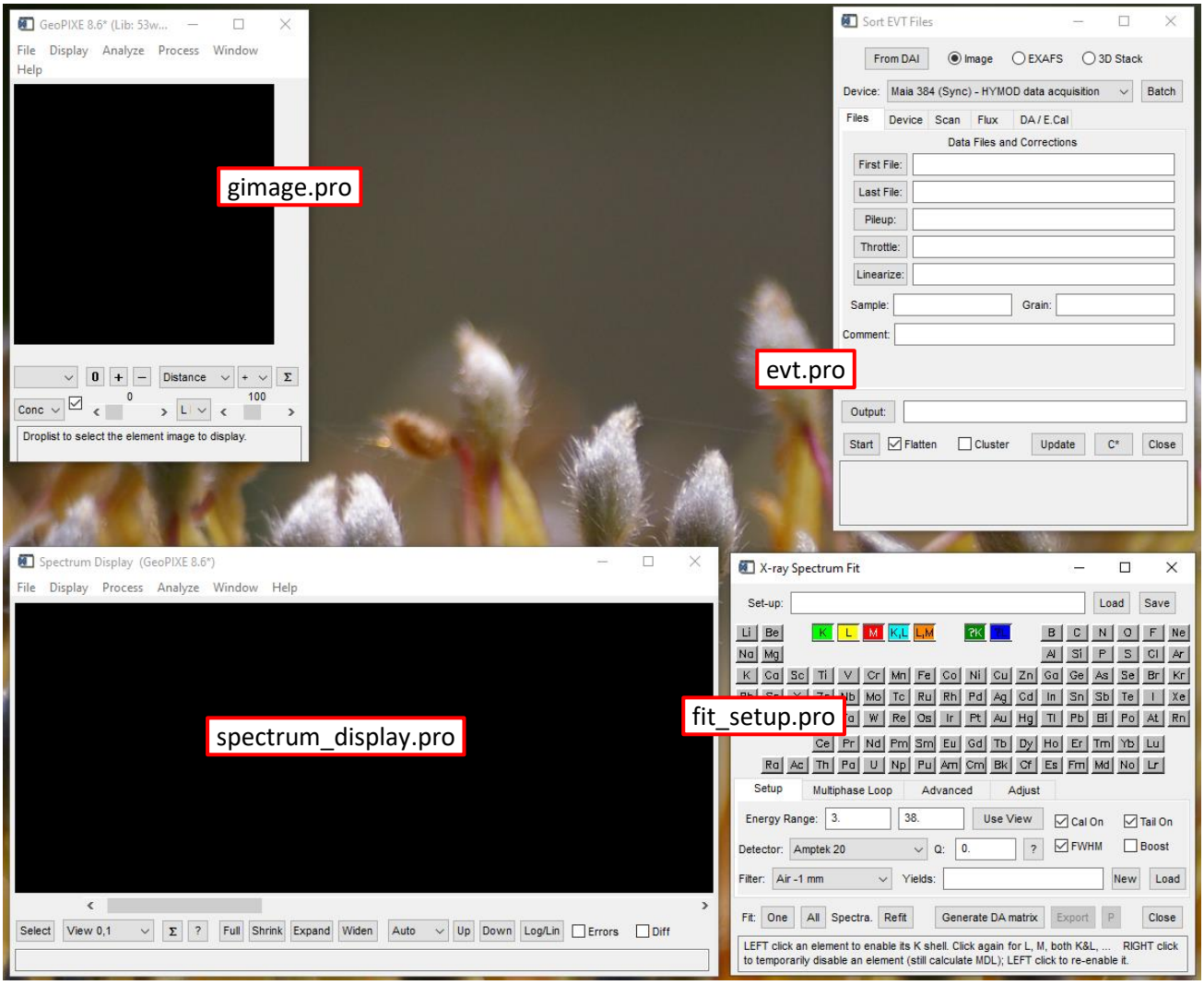
GeoPIXE Windows

Typical windows at startup

Some window can be flagged to be opened on GeoPIXE startup, using the "startup" entries in the "geopixe.conf" file.

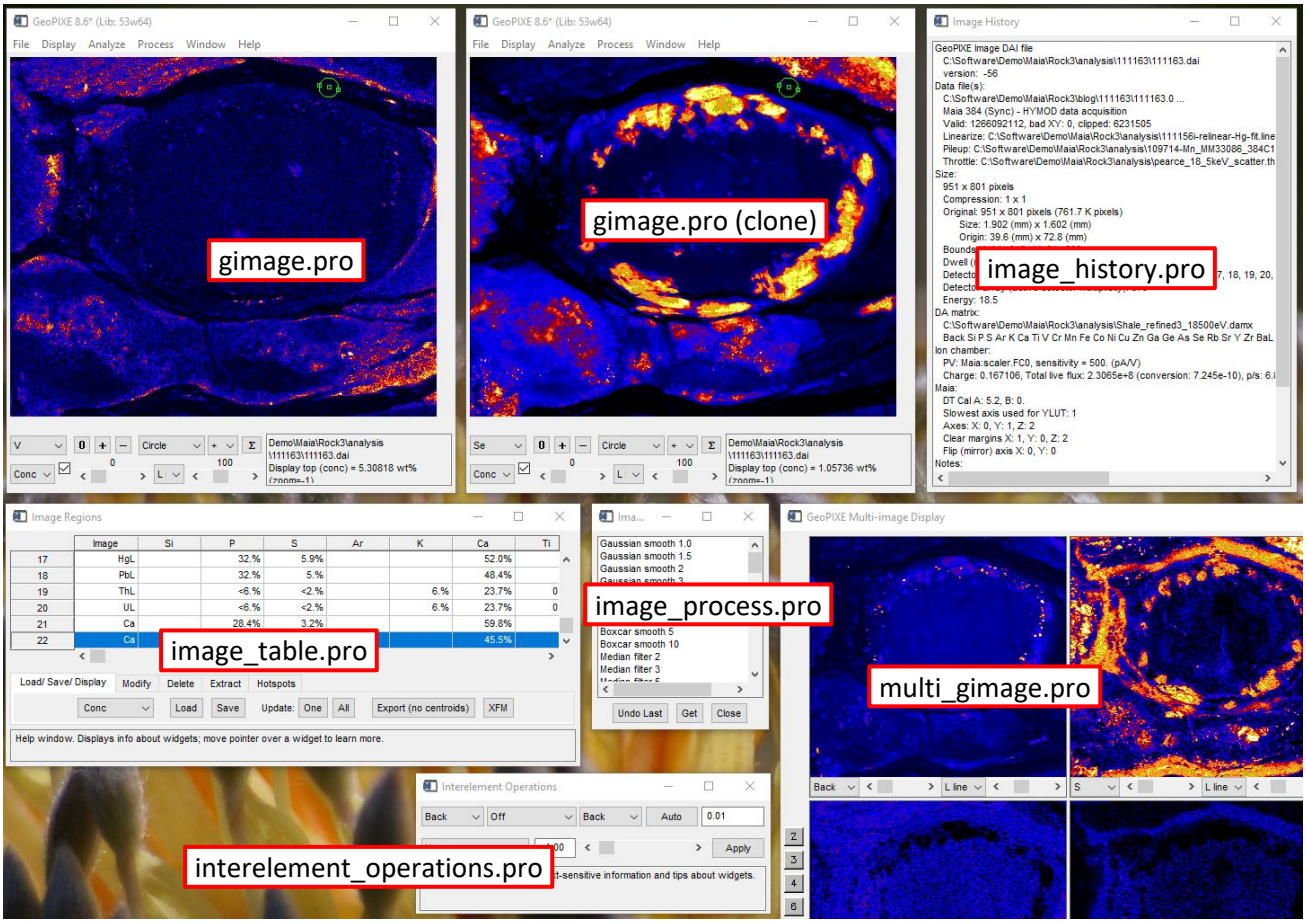
```
#
# startup
#       flags windows to open after GeoPIXE is started
#
        startup image_clone     0               # Extra clone of image window
        startup regions         0               # image regions window
        startup spectrum        1               # spectrum display
        startup identify        0               # X-ray line identification window
        startup fit             1               # Xray spectrum fit
        startup sort            1               # Sort EVT window
        startup select          1               # Spectrum select window
#
```

"*Image*" (gimage.pro) is the main GeoPIXE window. *Spectrum Display* (spectrum_display.pro) is launched from gimage.pro for display of (multiple) spectra. Multiple instances of each can be opened (e.g. use "display→clone" menu in "image" to open image clones or select from the "Windows" menus in *Image* or *Spectrum Display*). *Sort EVT* (evt.pro) is the window controlling sorting of raw data into images. "Xray spectrum fit" (fit_setup.pro) is for fitting spectra and generating a DA matrix for imaging. A number of other windows are launched from fit_setup.pro (see below).

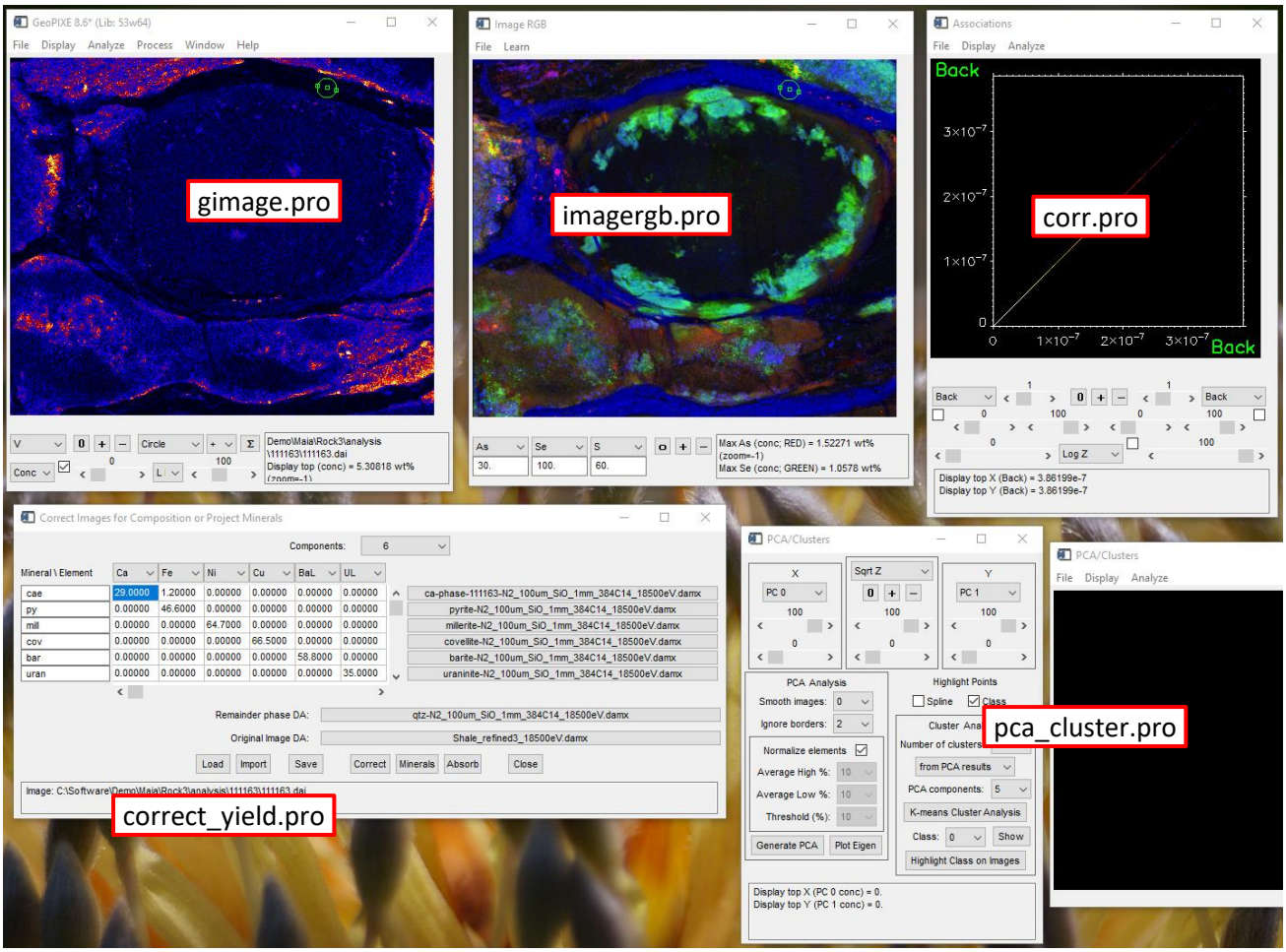

gimage.pro
evt.pro
spectrum_display.pro
fit_setup.pro

Windows launched from gimage.pro

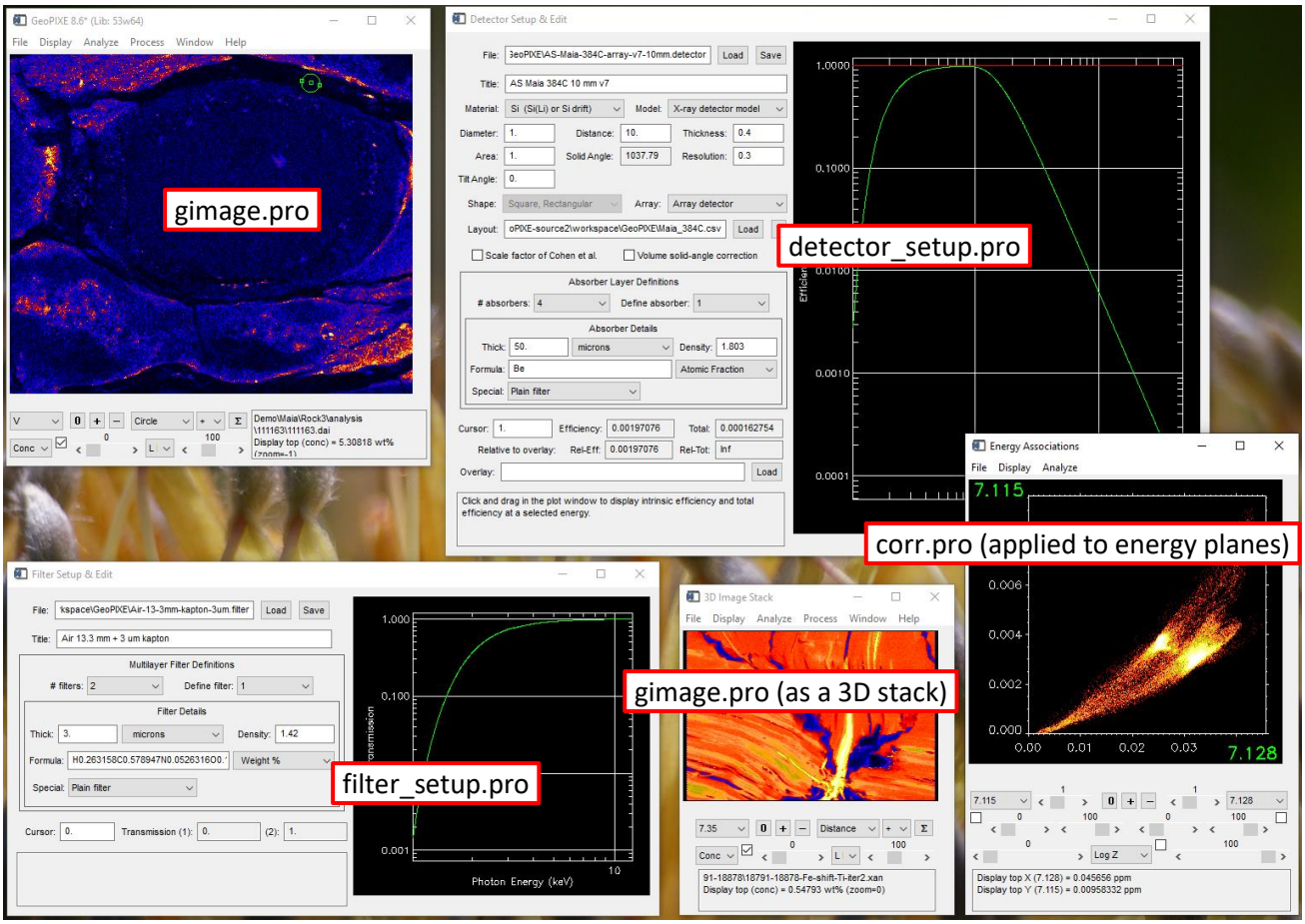
A number of windows are launched from gimage.pro, mostly for display of image data in various ways, to look at image regions (shapes on the image that define selected pixels, or pixels selected by element-element *Association* (corr.pro)) or to apply processing or corrections to images. This first set shows a clone of "gimage.pro", *Image History* (image_history.pro), which shows details of a loaded image, *Image Regions* (image_table.pro), which lists image regions applied to an image, *Image Process* (image_process.pro), which provides a one-click list of operations, digital filters, etc. to apply to an image, *Multi Image* (multi_image.pro), which shows a grid of many simple image windows (simple_image.pro) and *Interelement Operations* (interelement_operations.pro), which permits correction of artefacts caused by interaction between elements/ images.



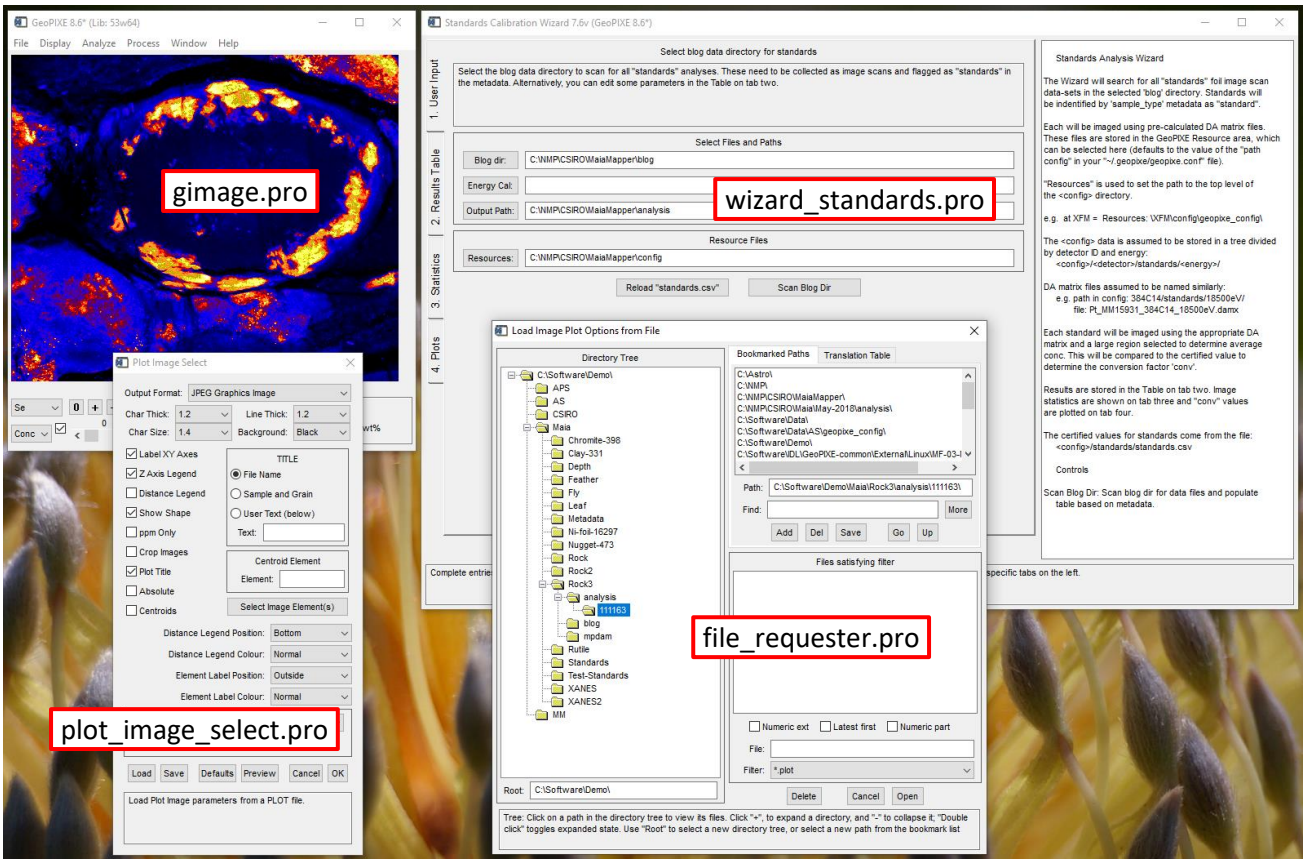
The second set of windows launched from gimage.pro shows *Image RGB* (imagergb.pro), which displays 3 element planes as Red, Green, Blue to make a 24-bit false-colour image, *Associations* (corr.pro), which displays the *Association* window to plot the image pixel data for 2 selected elements as a scatter plot/2D histogram (colour provides histogram intensity axis), *Correct Yields* (correct_yield.pro), which provides a tool for correction of images for complex matrix effects and *PCA Clusters* (pca_clusters.pro), which provides a tool for simple PCA and cluster analysis (K-means).



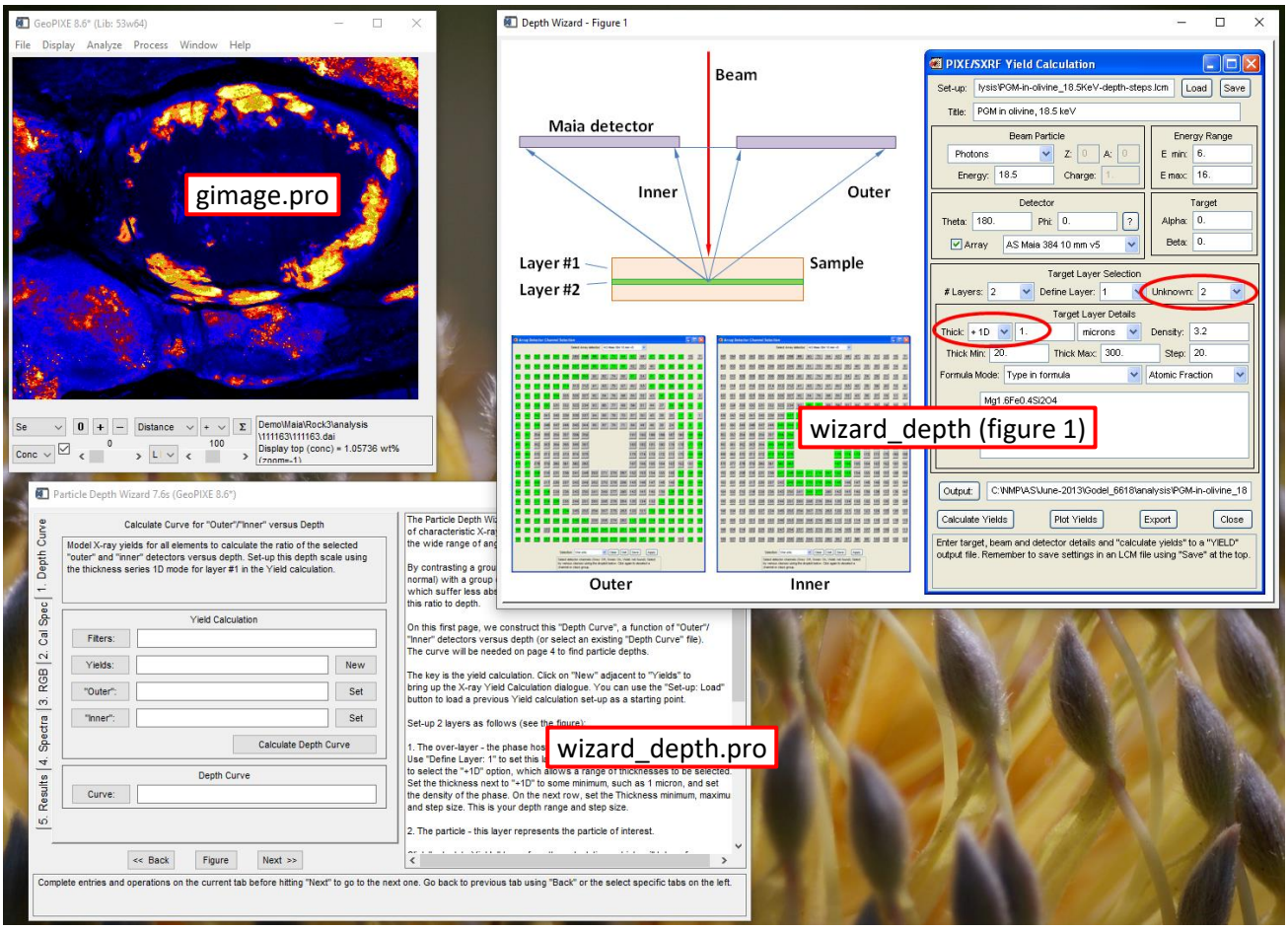
The third set of windows launched from gimage.pro shows *Detector Setup* (detector_setup.pro), which defines the parameters for a detector specification, *Filter Setup* (filter_setup.pro), which defines the parameters for a filter stack, *3D Stack*, which uses the same "gimage.pro" (with /xanes set) to open a 3D stack for XANES images and an *Energy Association* window, which uses "corr.pro" again (but with transformed data) to display a scatter plot/2D histogram of the association between energy planes in a XANES image stack or Line-XANES data.



The fourth set of windows launched from gimage.pro shows *Standards Wizard* (wizard_standards.pro), which is a Wizard for processing standards samples for flux calibration, *Export Images* (plot_image_select.pro), which is a modal/blocking popup to setup for the export of image data as a plot with axes, etc. and “file_requester.pro”, which is also a modal/blocking widget used throughout GeoPIXE for file browsing and selection.

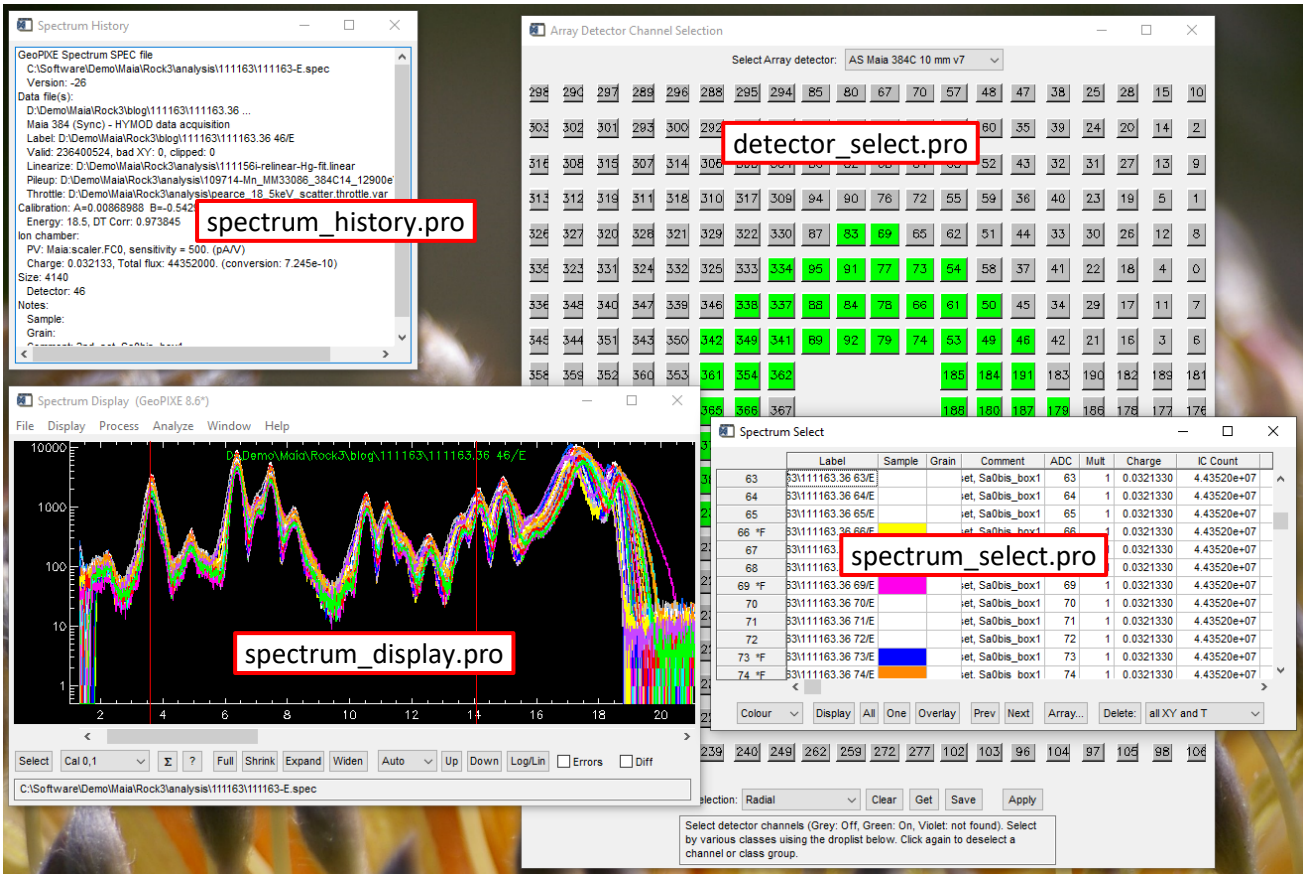


The fifth set of windows launched from gimage.pro shows *Depth Wizard* (wizard_depth.pro), which guides the user through depth analysis using the Maia detector array geometry (it also displays instructions and pops up figures to illustrate the process).

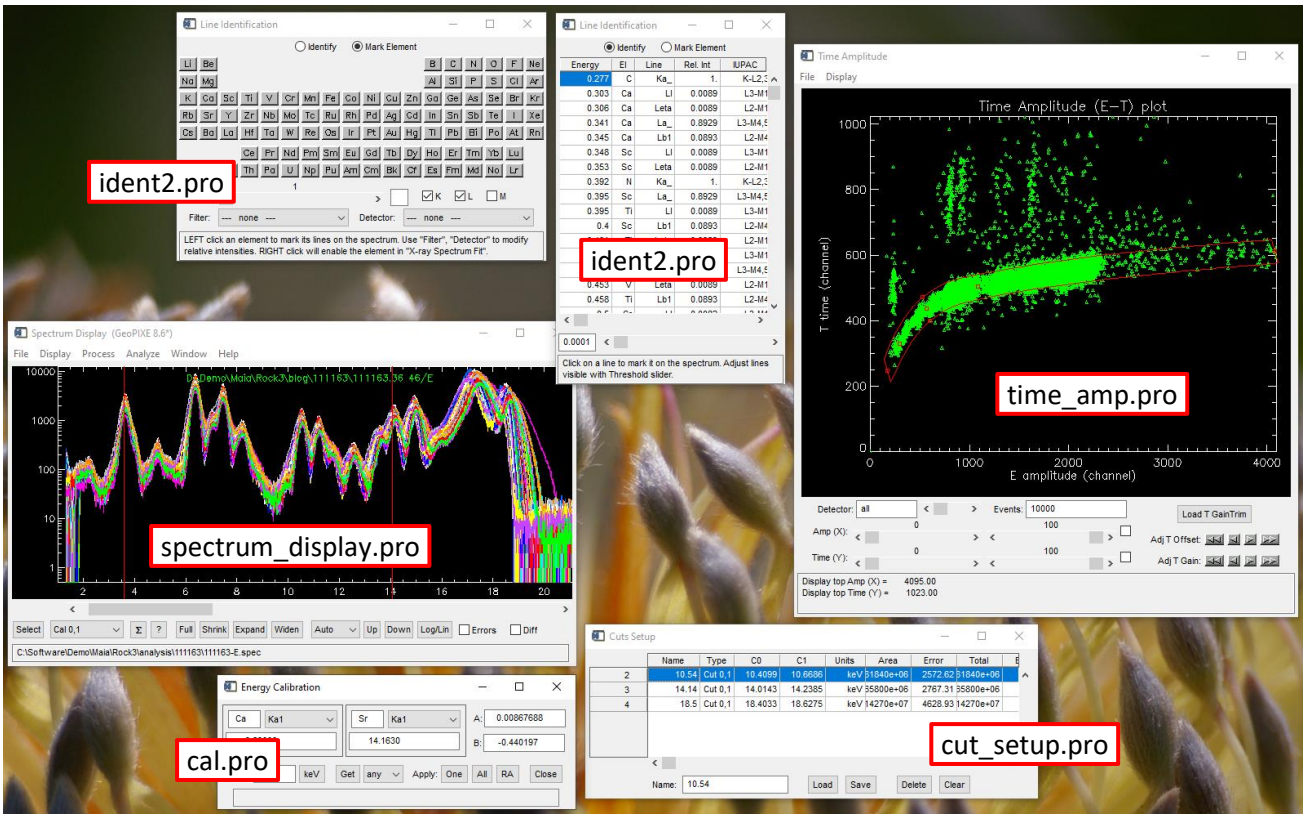


Windows launched from spectrum_display.pro

The first set of windows launched from spectrum_display.pro shows *Spectrum History* (spectrum_history.pro), which shows details for spectral data, *Spectrum Select* (spectrum_select.pro), which shows a list of spectra loaded and enables selection of selected spectra and *Detector Select* (detector_select.pro), which is launched from *Spectrum Select* (spectrum_select.pro) using “Array …” button, to permit detailed selection of members of the detector array.

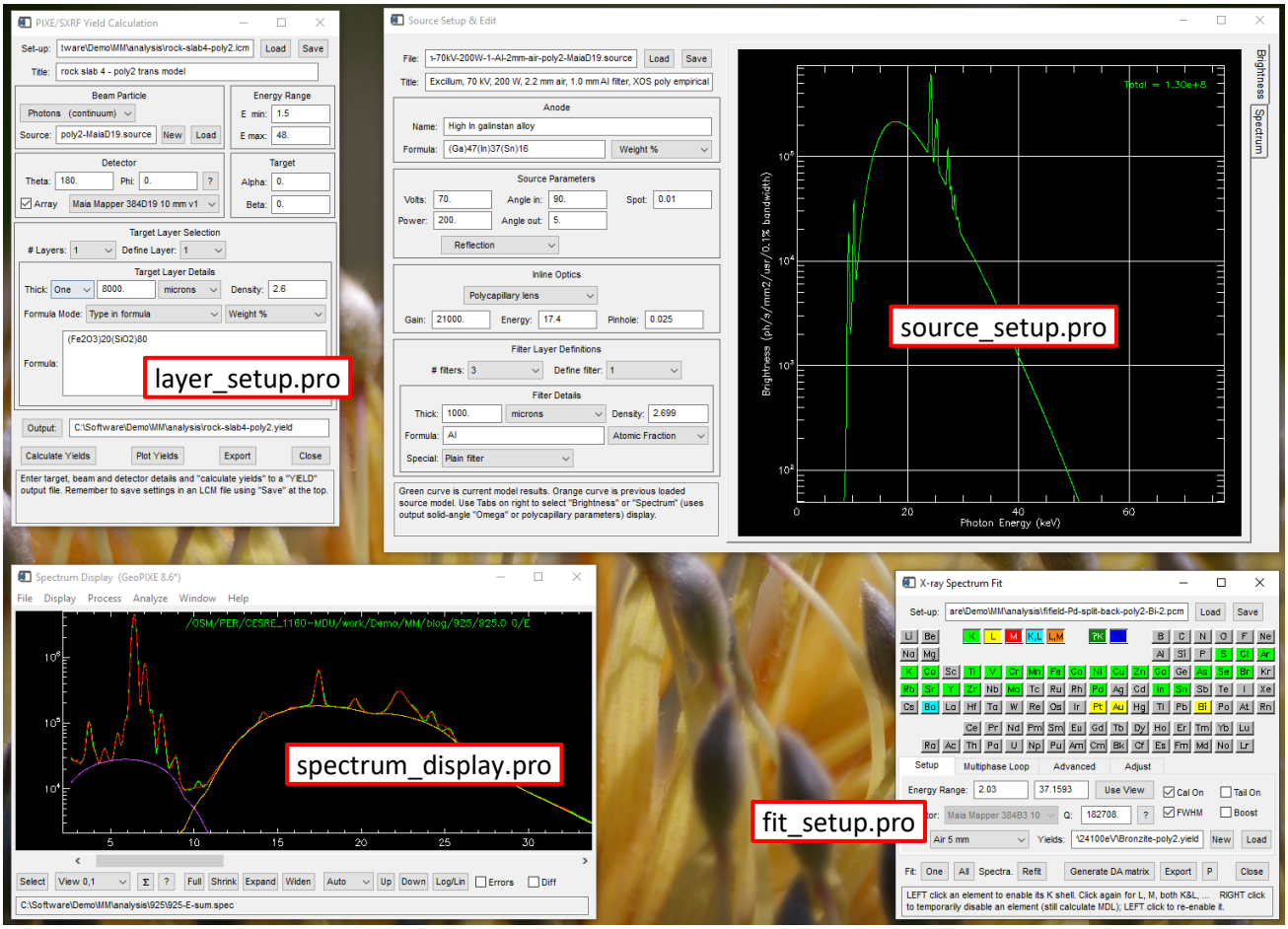


The second set of windows launched from spectrum_display.pro shows *Identify* (ident2.pro) shown in two instances as a periodic table to mark lines for an element on the spectrum and as a list to search for a match in the sorted list of X-ray lines, *Time Amplitude* (time_amp.pro), which displays raw Maia events as a scatter plot of E versus T to enable a pileup field to be defined, Cal (cal.pro), which permits energy calibration of spectra and *CUT Setup* (cut_setup.pro), which allows portions of spectra to be defined as CUTs using X (with background subtraction) or Cut (simple energy range) markers.



Windows launched from fit_setup.pro

The set of windows launched from fit_setup.pro (which is launched from *spectrum_display* (spectrum_display.pro) shows *Layer Setup* (layer_setup.pro), which defines a layered sample structure and calculates yields and X-ray line intensities and *Source Setup* (source_setup.pro), which defines a laboratory X-ray source model and calculates a continuum (plus characteristic anode lines) spectrum to be used in the yield calculation. “source_setup” is launched from *Layer Setup* (layer_setup.pro) and is located in a separate project (i.e. it’s SAV file is loaded like a plugin at run-time).



GeoPIXE program structure and communication

Routine organization

Each window program (e.g. see “evt” PRO file, which launches the *Sort EVT* window) has a definition routine (e.g. “evt”), which defines the widget hierarchy and calls more routines to realize the widgets for the window and an event routine (e.g. “evt_event”), which acts on widget events (and extra *Notify* events from other windows – see below). The event routine accesses the “state” data structure (see below), which contains all needed info about the window, including data (usually by pointer reference passed from the parent) and widget IDs and performs actions or updates parameters selected in the widgets by the user.

Widget events

Events come from various sources: (i) widgets (when a widget is selected or changed by the user), (ii) tracking (when mouse pointer passes over a widget with /tracking enabled), which is used for context sensitive help, (iii) timers (periodic events when enabled on a widget), (iv) kill request (to close the window) and (v) GeoPIXE ***Notify*** events from other windows (see below). Generally, for most GeoPIXE windows,

events are processed by detecting the 'tag_name' of the 'event' structure, which helps identify what type of event it is. A case statement on 'tag_name' is used to detect Tracking, Timer, Kill and Notify events. Anything else is assumed to be from a widget in the window's widget hierarchy and GeoPIXE assumes that the 'uname' has been set for each widget providing a unique and meaningful name for each widget, which makes the code more readable. A case statement on 'uname' follows for processing each widget's actions.

Notify events

IDL regards each window as a separate "program" and handles events from screen widgets in an event handler for the window. In order to get some transfer of information between windows, we needed to develop some simple tools. It is possible to jam an event structure into the top-level base of a window program to be acted upon like a widget event (see "send_event" argument to 'widget_control' routine in IDL). This mechanism is used (via the "**Notify**" routine) to 'send' an event to a window (with 'tag_name' "NOTIFY").

The approach used is for all windows to declare or "register" (see "register_notify" routine) their ability to receive certain events, either from any source, or from a specific window source (keyed to base widget ID). Typically, when a parent window opens a child window, it registers that it wants to receive events of certain types from the child based on the widget ID of the child's top-level base. The child also registers that it wants to receive events of certain types from the parent based on the widget ID of the parent's top-level base. Chains of windows with parent-child relationships can repeat the registering of a certain Notify key, which enables the Notify to be "passed on" along the chain, even if a specific window in the chain does not need to act on it. For example, the Notify action "image-region-select" can originate in the *Image Region* window when a row in the region table is selected (see "image_table" and "image_table_eventcb" PRO files) and be passed on by the parent main *GeoPIXE Image* window (see "image" and "image_eventcb" PRO files) and get acted on by the *Spectrum Display* window (see "spectrum_display" and "spectrum_display_eventcb" PRO files) or the *Association* and *PCA Cluster* windows (see "corr", corr_eventcb", "pca_cluster", "pca_cluster_eventcb" PRO files), which are children of the *Image* window. This enables the selection of a region row in the *Image Region* table to propagate and result in the corresponding spectrum and overlay to be displayed in the *Spectrum Display* or *Association* windows. *Image Regions, Association* and *Spectrum Display* are child windows opened from the main *Image* window parent.

Some programs simplify this a little. For example, the Wizards simply broadcast certain event Notify requests (not targeted at a specific base ID), and any capable window can act on them and return a result back to the Wizard. This makes them simple. However, for this reason, the Wizards are used alone and only with a single instance of various GeoPIXE windows. The Wizard also uses this mechanism to check whether the needed windows are available or unnecessarily duplicated.

For example, see the end of the "evt" PRO file, which defines the *Sort EVT* window. It registers to receive a number of Notify requests from the *Image* parent (variable 'group' is the parent top-level base widget ID) and also "wizard-action" requests from any Wizard. The parent is the main *GeoPIXE Image* window, which also registers to receive requests back from 'evt' (see "image_EVT" routine in "image_eventcb" PRO file), such as the return of a new processed image to display. For example, the "images" Notify request tells the *Image* window that the data processing in *Sort EVT* has constructed a new image data-set and returns a pointer to these as the argument to a Notify after the sort completes. *Image* responds to this (see "OnNotify_image" routine in "image_eventcb" PRO file) by adopting the pointer to the new image data structure as the new image to display and accepts responsibility for the pointer heap data.

Notify events often pass data, by pointer reference. Care must be taken with these, as the destination program will act on the data, on its timescale. The initiating program does not 'know' when the destination has finished processing and the pointer heap area is available for another purpose (e.g. sending another Notify) (unless a reply mechanism is used, as in Wizard requests). Hence, it is best to have a set of alternate

pointers to populate and send in subsequent Notify requests, so that by the time you come back to reuse one of these, you can be sure that its previous invocation has been completed. It is a good idea to use a separate pointer/heap area for each planned destination purpose. This can be seen in the "evt.pro" source (see 'state' struct definition), where there are a number of defined pointers (e.g. pimage, pspec, pdev, pdet, pval, pval2, pwiz) with allocated heap storage (i.e. `ptr_new(/allocate_heap)`) for each purpose. Not doing this can result in weird things happening, if the data is changed before a destination has handled it. Such are the joys of a multi-threaded environment. Alternatively, have a convention that the receiver takes ongoing responsibility for the pointer heap data, as is the case when *Sort Evt* passes new image data to the *gimage* window.

See the spreadsheet table of Notify event tags ("Notify-map-GeoPIXE.pdf").

Window "State" data

Data local to a window is by convention in GeoPIXE, stored in the 'uval' of the *first-child of the top-level base* of the window as a pointer "pstate" which points to a structure containing all local window data. Non-window data, such as X-ray and image data is typically stored in another pointer to structure, which has storage allocated in the parent and passed to the child, so that it lives on if the child window is closed. Re-opening the child window restores the data seen previously. Commonly, a copy of this pointer is kept as 'p' in the 'pstate' structure for local use. Initially, the 'p' structure does not exist and is created by the child window program ('p' is tested on entry using the 'bad_pars_struct' routine) to populate the heap storage allocated in the parent.

For example, see routine "evt", which handles the *Sort EVT* window. It uses 'bad _pars_struct' to test 'p' as passed (with storage allocated in the parent *GeoPIXE Image* window, see "image_eventcb" PRO file) and if not defined constructs a 'pars' struct for the first time, which gets stored in the 'p' allocated heap memory back in the parent. A copy of 'p' for *Sort Evt* is stored in the "state" structure, which gets saved as 'pstate' in the 'uval' of the first-child of the top-level base (see end of the "evt" PRO file). If the *Sort EVT* window gets closed and then re-opened, the newly created "pstate" will inherit a new copy of the 'p' data pointer from the parent so that the parameters in the window are restored.

This approach to routines also means they can often be run stand-alone for testing, and then they create the needed data structure 'p' from scratch. It also means that you can have multiple instances of windows without issues, such as using *Image* clones. However, these windows will share the same 'p' data in the parent (hence, it makes little sense to have 2 *Sort EVT* windows open).

Program data

Complex data structures that are reused around GeoPIXE are typically declared using the "define" function, which takes a key-name as its argument. This guarantees consistent declarations around GeoPIXE. But take care – if "define" is used in another program (which loads GeoPIXE as a library), its use must be consistent with the current compiled GeoPIXE.sav. Any addition or modification to 'define' may require compiling both GeoPIXE and the other program(s) (e.g. plugins, devices and wizards). Take care – additions at the end of a structure can be simply handled while changes may be dangerous and demand much re-compilation across projects that use it. Historically, modifications can be better handled with an explicit new version of a data structure (e.g. "filter", "old_filter1").

Pointers can be very efficient, passing an entire data structure in a call as a simple pointer reference, for example. But that can cause problems, if you'd like the fate of the data passed to be independent of the parent's copy. For example, freeing memory in the child may free memory in the parent's copy as well. The util routine 'copy_pointer_data' allows either a clone of an entire data structure to be made (when /init is set) or the entire source data structure traversed, and values copied to a matching destination structure (e.g. created initially from the source using /init). Importantly, the contents of any pointers encountered are

replicated into new pointer heap across in the destination so that the source and destination share no common pointer memory allocations and can be henceforth used independently, and memory freed without upsetting the source.

Example of key elements of a widget program in GeoPIXE

Based on the “evt.pro” routine (the “Sort EVT” window for sorting raw data into images), these code snippets illustrate the structures used throughout GeoPIXE.

The main routine for the *Sort EVT* window (“evt” in “evt.pro”) constructs the widget tree. This snippet shows the main elements, but only a fraction of the widgets (skip compile_opt, error catch and most widgets) … The call passes in the local data structure pointer ‘p’ stored in the calling parent, an initial file ‘path’, ‘group_leader’ of the parent to set *Sort EVT* as part of a window group hierarchy (which can be closed *en masse*) and ‘tlb’ will return back the widget ID of the EVT top-level base once created and realized. The optional IDL passed argument ‘_extra’ can pass an unspecified number of extra arguments, which are passed to the definition of the top-level base widget.

The function ‘`bad_pars_struct( p, make_pars=make_p)`’ tests if the passed ‘p’ points to valid heap data in the form of a struct. If not, ‘`make_p`’ will be set to flag building the struct here (not all shown) and copy it into the parent’s heap that ‘p’ points to (*p). This usually happens on first entry into ‘evt’, and the data is stored in the ‘image’ window and will be used next time *Sort EVT* is opened. ‘`define_devices`’ looks for all SAV files in the GeoPIXE directory “interface” with names “*_device__define.sav”, which are the raw data import device objects.

Main widget definition routine (partial) for Sort EVT window

```
pro evt, group_leader=group, TLB=tlb, image=image, pars=p, path=path, _extra=extra

define_devices, titles=device_titles, names=device_names
p = bad_pars_struct( p, make_pars=make_p)

if make_p then begin
        DevObjList = instance_device_objects( device_names, error=err)
        if err then begin
                warning,'EVT',['Failed to open Device Objects "xxx_device__define.sav"’]
                return
        endif
        if obj_valid(DevObjList[0]) eq 0 then begin
                warning,'EVT',['Failed to open Device Objects.','Obj array invalid.']
                return
        endif

        pars = {        $
                device:         0, $                            ; list-mode device index
                pDevObjList:    ptr_new(DevObjList), $          ; pointer to device obj array
                evt_file:       '', $                           ; EVT file name
                xsize:          0.0, $                          ; X size (microns)
                ysize:          0.0, $                          ; Y size (microns)

                . . .

                enable:         intarr(max_adcs), $             ; enable station
                type:           intarr(max_adcs), $             ; type (PIXE)
                cal_a:          replicate(1.0,max_adcs), $      ; cal A
                cal_b:          fltarr(max_adcs), $             ; cal B
                mode:           intarr(max_adcs), $             ; mode (DA) index
                file:           replicate(' <--- select file ',max_adcs) $  ; file name
        }
        pars.enable[0] = 1
        *p = pars
```

```
endelse

DevObj = (*(*p).pDevObjList)[(*p).device]                    ; current device object

tlb = widget_base( /column, title='Sort EVT Files', /TLB_KILL_REQUEST_EVENTS, $
       group_leader=group, _extra=extra, uname='evt_TLB', /base_align_center, $
       xpad=0, ypad=space2, space=space2, xoffset=xoffset, yoffset=yoffset )
tbase = widget_base( tlb, /column, xpad=1, ypad=0, space=space2, /base_align_center)

dbase = widget_base( tbase, /row, /base_align_center, xpad=3, ypad=0, /align_center)

lab = widget_label( dbase, value='Device:')
device_mode = widget_combobox( dbase, value=device_titles, uname='device_mode', /tracking, $
       notify_realize='OnRealize_device_mode', xsize=device_xsize, $
       uvalue='Select input device driver for the list-mode (event-by-event) file(s).')

batch_button = widget_button( dbase, value='Batch', uname='batch_button', /tracking, $
       uvalue='Batch operation of Sort EVT: Set-up "Sort EVT".')

tab_panel = widget_tab( tbase, location=0, /align_center, uname='tab-panel')
tab_names = ['Files','Device','Scan','Flux','DA / E.Cal']

files_base = widget_base( tab_panel, title=tab_names[0], /column, xpad=1, ypad=1,
       scr_xsize=tab_xsize, space=space1, /base_align_right, /align_right)
lab = widget_label( files_base, value='Data Files and Corrections', /align_center)

evt_base = widget_base( files_base, /row, /base_align_center, xpad=2, ypad=0, /align_right)
evt_button = widget_button( evt_base, value=name, uname='evt_button', /tracking, $
       uvalue='Browse to select the list-mode file to sort.', scr_xsize=evt_button_xsize)
evt_file = widget_text( evt_base, value=(*p).evt_file, uname='evt_file', /tracking, /editable, $
       notify_realize='OnRealize_evt_file', scr_xsize=evt_file_xsize2, $
       uvalue='Enter a file-name for the list-mode file to sort.')

               . . .

help = widget_text( tbase, scr_xsize=help_xsize, ysize=4, /wrap, uname='HELP', /tracking, $
       uvalue='Help window. Displays info about widgets.',frame=0)

state = {  $
               tlb:                tlb, $               ; top level base
               p:                  p, $                 ; pointer to parameters
               path:               ptr_new(path), $     ; pointer to current path
               dpath:              ptr_new(dpath), $    ; pointer to current path
               pprefs:             pprefs, $            ; preferences
               device_mode:        device_mode, $       ; device droplist ID
               pimage:             ptr_new(), $         ; pointer to images for Notify
               pval:               ptr_new(/allocate), $ ; pointer Notify parameter passing
               pwiz:               ptr_new(/allocate), $ ; pointer for return to Wizard

               . . .

               evt_file:           evt_file, $          ; EVT file-name text ID
               help:               help $               ; help text ID
       }

child = widget_info( tlb, /child)
widget_control, child, set_uvalue=ptr_new(state, /no_copy)
widget_control, tlb, /realize

; Register to receive Notify events from elsewhere …

register_notify, tlb, ['path', $                        ; new path
                       'dpath', $                       ; new raw data path
                       'done-filter'], $                ; batch filters applied
                      from=group
```

```
register_notify, tlb, ['wizard-action']                    ; global Notify from a wizard

xmanager, 'evt', tlb, /no_block                            ; start 'evt' under xmanager control
return
end
```

The 'state' struct is constructed and by convention stored as a pointer 'pstate' in the 'uvalue' of the child of the top-level base. Then the whole widget hierarchy is realized on screen, which will call any optional 'OnRealize' routines. Then '`register_notify`' is called to register that this routine (via the top-level base 'tlb') wishes to receive *Notify* messages. The first set are only accepted if they come from '`group`', the parent window. The '`wizard-action`' Notify will be accepted from anywhere.

There are often some platform dependant tweaks that can be used to adjust GUI elements (some size and spacing adjustments) that differ under Windows, Linux and the Mac, which are placed in a case statement (`case !version.os_family of`).

Widgets that need to be initialized based on data in the local data 'p', will be flagged as having a "OnRealize" routine, which gets executed after the widget hierarchy is constructed on window open. Note that the **tlb_id()** function call (`top = tlb_id( wWidget)`) provides a simple method to search the widget hierarchy up from the local widget ID (`wWidget`) to find the top-level base widget ID. Then 'pstate' is found as the 'uvalue' of the first child of the top-level base, and then the parameters 'p' is found in (*pstate).p.

OnRealize routine to set the Device droplist in Sort EVT

```
pro OnRealize_device_mode, wWidget

COMPILE_OPT STRICTARR
top = tlb_id( wWidget)
child = widget_info( top, /child)
widget_control, child, get_uvalue=pstate
p = (*pstate).p

if ptr_valid( p) then begin
       widget_control, wWidget, set_combobox_select=(*p).device

       DevObj = (*(*p).pDevObjList)[(*p).device]                ; current device object
       *(*pstate).pdev = DevObj
       evt_set_array, pstate, DevObj->array_default()
endif
end
```

The event routine uses state data stored in pointer 'pstate', which by convention is stored in the 'uvalue' of the first child widget of the top-level base (the base of the widget program). The state is usually tested for validity (exists and is a valid pointer with data of type "struct"; newer programs use '**ptr_good(pstate, /struct)**', which does all this in one call) and we get a local copy of the data pointer 'p', which is usually stored in pstate. 'p' will typically be passed into the widget program from its parent (or main GeoPIXE 'image' window), so that the data survives if this widget window is closed (open it again and the data is still there).

A standard template error catch is also present. Any serious error within event handling will jump back here to pop up a warning (using 'warning' routine) and typically kill the window or continue. Error/warning pop ups are enabled globally in GeoPIXE using the parameter '`catch_errors_on`' in common. The '`COMPILE_OPT STRICTARR`' directive ensures no ambiguity between vector notation and functions (forces use of "[]" for vectors and "()" for function calls).

Samples from Sort EVT widget program ('evt.pro')

```
pro evt_event, event
```

```
COMPILE_OPT STRICTARR

ErrorNo = 0
common c_errors_1, catch_errors_on
if catch_errors_on then begin
	Catch, ErrorNo
	if (ErrorNo ne 0) then begin
		Catch, /cancel		; jump here on severe error in routine
		on_error, 1
		help, calls = s
		n = n_elements(s)
		c = 'Call stack: '
		if n gt 2 then c = [c, s[1:n-2]]
		warning,'EVT_event',['IDL run-time error caught.', '', $
				'Error:  '+strtrim(!error_state.name,2), $
				!error_state.msg,'',c], /error
		MESSAGE, /RESET
		goto, kill
	endif
endif

child = widget_info( event.top, /child)
widget_control, child, get_uvalue=pstate

if n_elements(pstate) eq 0 then goto, bad_state
if ptr_valid(pstate) eq 0 then goto, bad_state
if size(*pstate,/tname) ne 'STRUCT' then goto, bad_state
p = (*pstate).p
if ptr_valid(p) eq 0 then goto, bad_ptr
```

The event loop is usually structured to capture specific event types first (given by '`tag_names( event, /structure)`') for events like Notify (`NOTIFY`), timer (`WIDGET_TIMER`), tracking (`WIDGET_TRACKING`) and window close (`WIDGET_KILL_REQUEST`) and then handle all widget events within the window's widget hierarchy. By convention all widgets are given a unique 'uname' to identify them, so we just need to decode the source 'uname' for the event and have a case statement on this string. The 'event' struct contains fixed members for all events ('id' the ID of the widget originating the event, 'top' the top-level base ID for the widget hierarchy, 'handler' for the ID of the widget which has been assigned an event handler for the tree) and variable members (e.g. 'index' for a droplist or combobox [a droplist with scroll bars], 'value' for index into a set of checkboxes, 'select' for state of that checkbox).

Event handling in Sort EVT (partial code snippets)

```
case tag_names( event, /structure) of
	'NOTIFY': begin
		case event.tag of
			'path': begin
				if ptr_valid( event.pointer) then begin
					*(*pstate).path = (*event.pointer)
				endif
				end
			'wizard-action': begin
				if ptr_valid( event.pointer) then begin
					if (*event.pointer).window eq 'Sort EVT' then begin
						case (*event.pointer).command of
							'open-test': begin
								pw = (*pstate).pwiz
								*pw = *event.pointer
								(*pw).top = event.top
								(*pw).error = 0
								notify, 'wizard-return', pw
								end
. . .
```

```
                                                else:
                                                endelse
                                        endcase
                                endif
                        endif
                        end
                else:
        endcase
        goto, finish
        end
    'WIDGET_TRACKING': begin
        widget_control, event.id, get_uvalue=s
        if event.enter eq 1 then begin
            if size(s,/tname) eq 'STRING' then begin
                widget_control, (*pstate).help, set_value=s
            endif else if size(s,/tname) eq 'STRUCT' then begin
                t = tag_names( s)
                q = where( t eq 'HELP', nq)
                if nq gt 0 then begin
                    if size(s.Help,/tname) eq 'STRING' then begin
                        widget_control, (*pstate).help, set_value=s.Help
                    endif
                endif
            endif
        endif else begin
            widget_control, (*pstate).help, set_value=' '
        endelse
        goto, finish
        end
    'WIDGET_KILL_REQUEST': begin
        print,'Kill request evt ...'
        goto, kill
        end
    else:
endcase

uname = widget_info( event.id, /uname)
case uname of
    'start_button': begin
        (*p).flux = 0.0
        evt_start, pstate, group=event.top, pprefs=(*pstate).pprefs, /verify
        end
    'close_button': begin
        print,'Close evt ...'
        goto, kill
        end
    'device_mode': begin
        (*p).device = event.index
        DevObj = (*(*p).pDevObjList)[(*p).device]          ; current device object
        name = DevObj->name()
        evt_set_device, pstate, name, event.top
        end
    'throttle_file': begin
        widget_control, event.id, get_value=s
        evt_set_throttle_file, pstate, s
        end

        . . .
    else:
endcase
```

The kill routine will typically free allocated memory (not the ‘p’ data heap storage, which is handled by the parent) and close the widget tree (**widget_control**, event.*top*, /destroy).

The next snippet shows how “evt” is launched from the parent ‘image’ window. The pointer to EVT window data is stored locally as ‘pevt’ in the image window ‘pstate’ (`(*pstate).pevt`), which is passed as ‘pars’ argument to ‘evt’. ‘tlb’ will return the top-level base for the new EVT window. This is used in a ‘`register_notify`’ call to register that ‘image’ wants to receive *Notify* events of various types coming back from the new ‘evt’ window (`from=tlb`).

‘image_evt’ routine in ‘image’ to open the *Sort EVT* window (evt)

```
pro Image_EVT, Event

COMPILE_OPT STRICTARR
child = widget_info( event.top, /child)
widget_control, child, get_uvalue=pstate

EVT, group_leader=event.top, TLB=tlb, pars=(*pstate).pevt, path=*(*pstate).path, $
         test=(*pstate).test, dpath=*(*pstate).dpath, pprefs=(*pstate).pprefs, /image

register_notify, event.top, $
       ['images', $                          ; new images loaded
       'path', $                             ; new path
       'dpath', $                            ; new raw data path
       'spectra', $                          ; pass on notify of new spectra loaded
       'batch-operations-open', $            ; open Image operations window
       'batch-rgb-open', $                   ; open RGB Image window
       'batch-filter', $                     ; digital filter from Batch_Sort
       'batch-save' $                        ; save images/HTML from Batch_Sort
       ], from=tlb
end
```

GeoPIXE routines

Some key routines (a useful subset) are outlined here, grouped by the sub-directory in the “main” directory (e.g. util, device, xray, …). They are all compiled into “GeoPIXE.sav” and can be used like a library once this SAV file is loaded (using IDL “restore” command). Or, for debugging and testing in the IDL Eclipse environment, they will be found and compiled by IDL, as they are typically on the search Path (!path). The main GeoPIXE program is called simply “gimage”. (Note that IDL routines and variables are NOT case sensitive.) Typically, *Gimage* is called in debug mode (i.e. “GImage, /debug”), which will stop at the line of any error encountered to permit debugging in the IDLDE Eclipse environment. Otherwise, the error Catch mechanism described above will take effect.

There are about 3423 routines in GeoPIXE.sav in 747 source PRO files, out of 1218 PRO files in the entire project workspace.

GeoPIXE windows

spectrum_display	Main *Spectrum display* window (see also “spectrum_display_eventcb” for widget routines and “spectrum_routines” for further spectrum functions).
spectrum_select	Selection of spectra overlays (see also “spectrum_select_eventcb” for widget routines).
spectrum_history	Display internal parameters and statistics for a set of spectra (see also “spectrum_history_eventcb” for widget routines) and “spectrum_details” for the content of the window.
fit_setup	X-*ray Spectrum fitting* window to perform fits to spectra, generate DA matrix.
layer_setup	Setup and perform an *X-ray yields calculation*.
cal	Energy calibration window for spectra (see also “cal_eventcb” for widget routines).

gimage	Main *Image* display window (see also “image_eventcb” for widget routines and “image_routines” for further Image functions). Image can also function with 3D image stacks for XANES and tomography (/XANES).
image_history	Display internal parameters and statistics for an image data-set (see also “image_history_eventcb” for widget routines) and “image_details” for the content of the window.
image_table	*Image regions* table for regions analysis and extraction from images (see also “image_table_eventcb” for widget routines).
image_process	*Image processing* functions to apply to images (see also “image_process_eventcb” for widget routines).
evt	*Sort EVT* window for processing raw data to images.
correct_yield	*Correct image* for composition effects on yields and project onto phases.
Inter_element	*Inter-element* operations for correction for inter-element effect artefacts.
imagergb	*RGB image* window for RGB element overlays (see also “imagergb_eventcb” for widget routines and “imagergb_routines” for further RGB functions).
corr	Element *Associations* window (see also “corr_eventcb” for widget routines and “corr_routines” for further Corr functions).
pca_cluster	*PCA/Cluster* window for PCA/Cluster analysis (see also “pca_cluster_eventcb” for widget routines and “pca_cluster _routines” for further PCA functions).
batch_sort	Batch sorting control of ‘evt’ plus image processing workflow.

GUIs for setup

filter_setup	setup a filter specification and struct. Results returned as ‘pars’ pointer.
detector_setup	setup a detector specification and struct. Results returned as ‘pars’ pointer.
source_setup	setup an X-ray source specification and struct.
cut_setup	setup spectrum CUTs

Support routines

append_da_fit	Construct the spectrum overlay for a region based on DA components.
build_da_fit	Called by “append_da_fit” to do the work.
build_image_scale	Look at display parameters and set the 'low' and 'high' display range for the current display mapping (linear, log, sqrt, ...)
set_image_minmax	Set initial display ranges for images.

Data processing

da_evt	process event data from a device into images using DA.
spec_evt	process event data from a device to extract total E,X,Y,T spectra.
. . .	

Parallel processing

cluster_client	Foreground control of parallel processing, including a progress bar, and set-up of shared memory using the routines in "parallel_client".
cluster_merge_spectra	merge results from parallel processing extraction of region spectra
cluster_merge_images	merge results from parallel processing extraction of images
parallel_client	library of routines used by ‘cluster_client’ to launch, monitor, control and free processes and shared memory.
geopixe_execute	Decode (potentially very long) command line string for command, keywords and arguments and use the IDL 'execute' command to execute a command-line. Called from background process “geopixe_parallel”. The command string uses the “*stringify*” format.

Misc routines

define	used to set-up or 'define' consistent instances of many common structs used in GeoPIXE.
startupp	Setup IDL environment (e.g. screen device, colour table), load optionally devices (/device), X-ray database (/database), Maia_control (/maia) and DAQ_control (/DAQ).

Wrapper routines for Fortran

da_accumulate…	Call "da_accumulate…" Fortran to accumulate image using DA matrix.
cut_accumulate…	Call "cut_accumulate…" Fortran to accumulate image using CUTs.
. . .	

Util

Files

file_requester	File requester popup window and file search function.
file_search2	A version of 'file_search' that uses a progress bar and can be aborted if taking too long. It will return ALL matches to a wildcard 'spec' and traverse directory trees.
find_file2	Find files, like 'file_search' for a single file. If /virtual, then limit number of (dir) returns to a manageable number, by grouping them with "...". Group into purely numeric (sorted numerically) and non-numeric (sorted alphabetically) virtual returns. Number in 'groups' controlled by 'n_big'.
build_output_path	Determines the common 'root' of an input/output path pair to manage 'input' and create 'output' data paths.
close_file	Safe file unit closing and free the unit allocation.
extract_path	Return the path component of a full file-name.
extract_extension	Return file extension ('file_type' is the same)
dir_up	Traverse up a dir tree.
file_write_test	Test whether a new file is writable. Create the dir if needed.
file_pop_dir	Pop directory up one level, keep same file name.
file_lowcase	Force lower case only for systems without case-sensitive filenames.
fix_path	Make sure a path is correct (e.g. ends in path separator).
fix_file_name	Remove illegal characters from a filename.

Data handling

bad_pars_struct	Validates a passed parameters struct and flags a need for its creation.
big_endian	Returns true for a host O/S that uses big endian byte order.
binary_search	Recursive binary search
centroid	Form weighted centroid of a data vector, returns variance too
clip	Clips a value (or vector) to a range.
compress	Compress a vector by a factor (scale to maintain sum)
copy_pointer_data	Clone a complex data structure (/init), or copy its contents to another, maintaining separate pointer heap storage if encountered.
convol_self	Convolute a spectrum by itself for pileup estimation.
count_steps	Count the number of steps in vector argument (number of times it changes) and return step indices.
dimensions	Return data dimensionality (you can use the IDL "size" function too).
extend_image	Extend an image array (to permit certain image operations on it)
error_function	Error function and derivatives for 'Curvefit'.
finite_image	Test multi-dimensional image for non-finite pixels. Replace these with various options (mean, zero, norm) and return count.

gaussian_smooth	Convol with a Gaussian.
gauss_function	Gaussian function and derivatives for ‘Curvefit’.
hash_to_struct	Convert a hash to a struct.
Hayden	Perform a Hayden smooth (H.C. Hayden, Comp. in Phys., Nov (1987) 74)
image_increment	Increment 'image' by 'new', optionally compressed by 'xcompress', 'ycompress' starting at 'yoffset'. 'xoffset', 'yoffset' are defined in pre-compressed coordinates. If 'plane' specified, then only act on this plane of 3D array
is_a_hash	This detects a hash (e.g. on return from Python).
is_a_list	This detects a List (e.g. on return from Python).
is_a_struct	This detects a struct.

strings and variable types

sort_file_numeric	Sort a file list with numeric extension or file-names, or both.
build_result	Tailors result entries in tables into a string of controlled length/visibility
chop_string	Chops up a string – tend to use new IDL ‘strsplit’ these days
contains	tests whether a string/list contains a term (uses ‘str_equal’)
cross_reference	Form an array of indices that give the matching entry in array 'name2' for each in 'name1'.
extract	Extract sub-strings from strings (tend to use strmid these days).
is_lower	Is 'c' a lower-case alphabetic character.
is_upper	Is 'c' an upper-case alphabetic character.
lenchr	Return non-blank/white-space length of a string.
stringify	Package arbitrary structures of data (which may include pointers to data) into a (long) string. Unpack this string using “unstringify”. Optionally embed new pointer keys (use /embed_ptr).
unstringify	Decode a “stringify” encoded string of arbitrary data structure. Optionally, reconstruct any pointers to new allocated data structures (if /embed_ptr was used in *stringify*).
hide_embedded	Hide any character 'c' (e.g. ',') that are embedded within containers '[]', '{}', '""', '()', etc. in a *stringify* string. Used with ‘chop_string’ or ‘strsplit’ to avoid chopping within a bracketed term in a *stringify* string. /Unhide to restore something hidden (e.g. see ‘geopixe_execute’ for examples).
str_tidy	Function to convert argument to a string with controlled format (e.g. decimal places, strip trailing zeroes, etc.). Calls “strip_trail_zero”.
strip_trail_zero	Strip trailing and sundry zeroes in floating string values. Trim to significant digits. Ignore strip for non-floating value strings. Only strip after decimal point, or leading zeroes in value and exponent. Set places after decimal point.
strmid2	Like IDL ‘strmid’, but accepts the case of a string vector matched to an equal length vector for offsets or length, which strmid does not handle well.
float2, fix2, double2, long2, long2_64	safe versions of the IDL type change functions, that do not throw errors for non-numeric arguments
fnumeric	Logical, tests whether a string is a valid fixed-point float (e.g. Fortran F format)
inumefic	Logical, tests whether a string is a valid integer (e.g. Fortran I format)
gnumeric	Logical, tests whether a string is a valid float (e.g. Fortran E,G format)
pnumeric	Logical, is 'arg' a pure integer numeral string with no sign? Can't have any sign, modifier (byte length), white-space, tabs, etc. either. Keyword: /hex hexadecimal allowed.
vector_merge	If args are compatible, then combine as new vector 'x'. If necessary, upgrade type of 'x' to cope with 'y'. If incompatible, then convert to Lists first. If both structs, then use soft/ forgiving ‘struct_assign’.

Misc

alarm_popup	Popup an annoying flashing popup, if it's not already up.
ftp_connect	Connect to an FTP site

Enviroment

geolib_name	Returns name of routine in a Fortran Library – used in wrapper for 'call_external'.
geolib_cdecl	Returns CDECL flag for arg convention for O/S for the Fortran Library – used in wrapper for 'call_external'.
geopixe_library	Return the external library used for GeoPIXE on this platform.
geopixe_environment	Return the HOME directory used for user GeoPIXE files on this platform. Also return the TEMP path for temporary files.
geopixe_case_sensitive	Flags file-names are case sensitive on this platform.
Idl_version	Return the version of IDL (e.g. "8.5.1") and optionally also returns the revision (e.g. "8.5" for IDL 8.5.1 or "8.8" for IDL 8.8.0)
init_library	Detect O/S and set options for other geopixe environment calls.

Graphics

allocate_pixmap	Allocate memory for a graphics frame store "pixmap".
add_widget_vector	Link a series of Device object 'options' widgets together to be updated collectively (they may be located in separate windows). See Device Objects.
check_widget_vector	Checks widget validity and removes invalid widgets from Device options lists.
char_height	Returns graphics character height in Norm plotting coordinates.
circle_diam	Return X,Y vector coords around a circle of 'n' points.
data_to_norm	Convert *Data* plotting coordinates to *Norm* plotting coordinates
cw_fslider2	Extension of IDL float slider compound widget to add new features.
cw_bgroup2	Extension of IDL button group compound widget to add new features.
default_plot	Determines character size and line thicknesses for graphics output.
ellipse_diam	Make a vector of x,y coords for an ellipse marked on the major diameter and minor diameters.
figure	A popup designed to display figures for Wizards, etc.
find_leaf	Search a widget_tree 'root' for 'uvalue'.
find_id	Search down a widget hierarchy for 'uname', starting at 'id'.
get_widget_table	Get certain parameters from a table widget.
grab24	Grab the current window in True colour and output to 128 step GIF file.
grow_tree	Populate a directory widget_tree from 'path'.
histogram_plot	Histogram plot (filled bars), vertical or horizontal with options
intersection	Calculates the intersection of a 'ray' with a 'box'. Box: simple rectangular prism shape aligned with x,y,z axes. Ray: parametric equation for a line in terms of point 'p', a unit vector 'a' and a variable scaler 'lambda': vector x = p + lambda*a.

Bits and words

bit_jam	Create a byte/word/long/long64 from bits specified in a byte array
bit_split	Form a 0/1 byte array of the bits in an argument
hex_jam	Convert the Hex values in string (array) 'str' to an integer value (array). Assume they are padded to correct byte length.

Geometry

angle_lines	Angle between line vectors

experiment_angles	Calculated direction cosines for beam-target-detector 3D geometry

Objects

clone_object	clone an Object and handle Device objects specially.

Time

date_from_utc	Return a date/time string from a UTC seconds time

Chemistry

el_code	Decode element mnemonics with optional 'L', 'M' appended to indicate shell.
element_select	A popup to select elements (there is also a compound widget version called "select_element").

(see also "Xray" routines below …)

Xray

Some conventions for the indices used in the X-ray routines … See also the "database" routines.

Atomic shells are given a shell index:

```
Shells are:  K=1, L=2, M=3   (0 = all)
```

Edges are given an edge index:

```
1  K          6  Mii        11 Nii        16 Nvii       21 Ov
2  Li         7  Miii       12 Niii       17 Oi         22 Pi
3  Lii        8  Miv        13 Niv        18 Oii        23 Pii
4  Liii       9  Mv         14 Nv         19 Oiii       24 Piii
5  Mi         10 Ni         15 Nvi        20 Oiv
```

X-ray energies are organized by a line index. Each index corresponds to a unique line mnemonic, using a Siegbahn notation (e.g. Ka1, Lb2). Routines accept either index or mnemonic, and others convert between these:

```
Index Line      Index Line      Index Line    Index Line

  1   Ka1         2   Ka2         3   Ka3         4   Kb1
  5   Kb2         6   Kb3         7   Kb4         8   Kb5
  9   Ka_        10   Kb_        11   Kb11       12   Kb12
 13   Ll         14   Leta       15   La1        16   La2
 17   Lb1        18   Lb2        19   Lb3        20   Lb4
 21   Lb5        22   Lb6        23   Lg1        24   Lg2
 25   Lg3        26   Lg4        27   Lg5        28   Lg6
 29   La_        30   Lb_        31   Lg_           32   Ma1
 33   Ma2        34   Mb_        35   Mg_        36   Mz_
 37   M-NO       38   K-LL       39   K-MM
 40   sum        41   esc        42   gamma      43   Compton
 44   F197       45   Na440      46   elastic    47   Compton2
 48   M3N1       49   M2N1       50   M3O5       51   Lt
 52   Lb9,10
```

Siegbahn can be related to IUPAC notation (see "iupac.pro"):

```
Index Line  IUPAC   Index Line  IUPAC  Index Line  IUPAC  Index Line  IUPAC

 1   Ka1   K-L3      2   Ka2   K-L2     3   Ka3   K-L1     4   Kb1   K-M3
 5   Kb2   K-N2,3    6   Kb3   K-M2     7   Kb4   K-N4,5   8   Kb5   K-M4,5
 9   Ka_   K-L2,3   10   Kb_   K-MN    11   Kb11  K-M2-5  12   Kb12  K-N2-7
```

```
13   Ll    L3-M1    14   Leta  L2-M1   15   La1   L3-M5   16   La2   L3-M4
17   Lb1   L2-M4    18   Lb2   L3-N5   19   Lb3   L1-M3   20   Lb4   L1-M2
21   Lb5   L3-O4,5  22   Lb6   L3-N1   23   Lg1   L2-N4   24   Lg2   L1-N2
25   Lg3   L1-N3    26   Lg4   L1-O2   27   Lg5   L2-N1   28   Lg6   L2-O4
29   La_   L3-M4,5  30   Lb_   L-MNO   31   Lg_   L-NO

32   Ma1   M5-N7
33   Ma2   M5-N6    34   Mb_   M4-N6   35   Mg_   M3-N5   36   Mz_   M4,5-N2,3
37   M-NO  M-NO

38   K-LL  K-LL     39   K-MM  K-MM

48   M3N1  M3-N1    49   M2N1  M2-N1   50   M3O5  M3-O5
51   Lt    L3-M2    52   Lb9,10 L1-M4,5
```

Layers of samples are described in terms of a ‘layer’ struct, which contains { N:int, Z:intarr(), F:fltarr() } where N is the number of elements present, Z are the atomic numbers and F are the atomic fraction of each. Skip elements with F=0.

A layer used as a filter will include extra terms and options in a ‘filter’ struct. See the “define” routine for definitions of all common structs, such as /filter. However, layer is defined below.

Utility

make_layer	Make a ‘layer’ struct used for sample and detector layers.
make_filter	Make a ‘filter’ struct, as defined in “define”, used for filters.
make_detector	Make a ‘detector’ struct, as defined in “define”, used for detectors.
detector	Set default escape energies based on detector type/composition.

Formula string analysis

decode_formula	Decode a chemical formula string with groups (e.g. "(Fe2O3)12.3(Al2O3)23.1") into: ‘n’ number of valid elements found, ‘z’ atomic number vector, and ‘f’ atomic fraction for each z. The weights (outside of brackets) can be atomic weights (default) or weight % weightings (if keyword weight=1).
radical_split	Split between radicals (e.g. (Al2O3)12.3). If there are no "(",")", then there is only one radical.
decode_radical	Decode a chemical formula/molecular component of a formula string (e.g. "Fe\3\2O3"), including valence (within “\\”)
chop_radical	Chop a radical expression into a chemical formula part and a weighting factor part, e.g. (Al2O3)13.1 --> formula="Al2O3", x=13.1.
formula_split	Split formula between element and weight groups.
chop_element	Chop the 'str' into element name, and hence 'z', weighting 'x' and detect any qualifying valence number 'v', or K, L, M indicators, between "\".

X-ray lines

e_line	Line energy for a given ‘z’ and line ‘index’ (see line index table above).
line_e	Line energy for a given ‘z’ and ‘line’ mnemonic.
line_energy	More general version of ‘line_e’ that accepts more variations of the line mnemonics (e.g. ‘Ka’, ‘Ka_’ and ‘Ka1’, etc. for K alpha).
line_id	Line mnemonic string for a given ‘index’.
iupac	The IUPAC notation string for a given Siegbahn transition string.
line_index	Line ‘index’ for a given line mnemonic string.
list_line_index	List all line indices for a given ‘z’ and ‘shell’.
list_line_index_sort_e	List all line indices for a given ‘z’ and ‘shell’, sorted by energy.

index_shell	The X-ray shell index for the X-ray line index ‘n’.
index_edge	The X-ray edge/sub-shell index for the X-ray line index ‘n’.
include_shell	Returns a logical true (1) if any of the lines of the shell of element of atomic number 'z' are in the range e_min to e_max keV.
auger_rel_intensity	Relative radiative Auger line intensity.
auger_energy	Energy of radiative Auger transitions (KLL, KMM).
angstroms	Convert E (keV) to wavelength in Angstroms.
ekev	Convert wavelength W (Angstroms) to E (keV).
all_lines	Return all X-ray line energies (and relative intensities) for 'z' for selected shell(s).
major_line	Returns the index for the major line of the requested shell.
relative_intensity_xrf	Relative intensity of X-ray index 'n' for element 'z' for XRF.
relative_intensity	Relative intensity of X-ray index 'n' for element 'z' for ions or XRF.
escape_fraction	Escape fraction for detector for alpha escape or beta.
escape_energy	Escape energy for detector for alpha escape or beta.
energy_compton	Energy of Compton scattering of 'e' keV into angle 'thetad' (degrees). Return for every theta, unless multiple scattering, in which case calculate multiple scattering, for sequence of scattering angles theta, for each e.

NOTE: Many of these accept arguments as vectors and return a vector result.

X-ray absorption

absco	X-ray mass absorption coefficient for a given ‘z’ and ‘E’, where either may be vectors.
atten	Effective mass absorption coefficient (by Bragg rule) for a mixture of elements in a ‘layer’ for a given ‘E’, where either ‘layer’ or ‘E’ can be vectors. ‘layer’ could also be a filter layer.
transmit	Transmission through ‘filter’ layers for ‘E’ energies. If ‘E’ is a vector, then transmission is through all filters at each ‘E’.
filter	Like transmit but accepts direct filter description parameters.
det_eff	Detector efficiency for ‘detector’ struct and ‘E’ energies.

Device

base_device__define	Base Device Object class and base methods.
charge_sensitivity	Convert a 'val' multiplier and a 'unit' string into a sensitivity value.
clone_device_object	Make a new “cloned” instance of a device object.
copy_device_object_data	Copy the internal data for a device object to another object instance.
define_devices	Loads all device objects and returns ‘titles’ and ‘names‘.
device_specific	A wrapper to call the current device object’s “read_setup” method.
find_device_objects	Search for all XXX_device__define.sav files. Do not return "BASE_DEVICE" or "GENERIC_DEVICE". This needs to find devices even if called from SAV clients in sub-dirs (e.g. "maia").
find_device	Find an object called 's' in a vector of device objects 'objects', or a list of device names 'names'.
flux_scan	A wrapper to call the current device object’s “flux_scan” method to scan raw data files for flux Ion chamber (IC) variable (PV) information.
free_device_objects	Free device object(s)
generic_flux_select	Modal pop-up window to confirm flux parameter selections.
get_header_info	A wrapper to call the current device object’s “get_header_info” method
instance_device_objects	Return an instance of all listed device names.

list_device_imports	Return the vector of structs specifying the import data formats. If 'find' set, then search for a title to match.
list_device_objects	Return the device object list vector from common, as setup using “define_devices".
new_device_object	Make a new device object and inherit its display 'options' and 'header' from previous object instances.
open_device_objects	Search for all XXX__define.pro (or XXX__define.sav) files and probe them to build a list of device names and object definitions.
open_socket	Open a TCP socket for communication (e.g. to Maia).
read_buffer	A wrapper to call the current device object’s “read_buffer” method.
read_event_buffer	Read a buffer of raw data in format defined in device. Swap bytes as needed.
socket_command_get	Get selected data over a socket from Kandinski (Maia Hymod) variables.
socket_command_set	Set selected Kandinski variables to data over a socket
socket_command_mode	Used to silence "DDM communication error (token)" errors if DDM link in Maia is down (DAM power off).
socket_retry	Retry opening socket on read/write error. Keep count of retries.

Database

Some of the X-ray and other routines access data tables. These are generally kept in the “database” directory. Generally, rather than read in database tables on each invocation, they are kept in common data arrays, which get loaded into GeoPIXE (from file “geopixe2.sav”) using the routine “startupp, /database” or directly using “restore_database”. Note then that file “geopixe2.sav” does not contain any routines, just data. To enforce using “geopixe2.sav” and catch any errors/omissions that may not be seen in Eclipse until tested at run-time using the compiled GeoPIXE, the directories for the database data files are often renamed “…-off” so they are not found by the “init” routines called by each if the common is not setup. This helps test that the commons are setup and all data is available during testing.

startupp	Determines local ‘geopixe_root’ directory where “GeoPIXE.sav” resides and other data using the options:
/database	pre-compiled arrays of data
/colours	set default image and spectrum plot colours
/devices	load all device objects (“*_device__define.sav”) from “interface”.
/Maia	loads “maia_control.sav” for Maia background processes
/DAQ	loads “daq_control.sav” for DAQ36 background processes
atomic_number	Atomic number given element name string.
element_name	Element name for a given atomic number ‘z’.
density	Density of common form of simple elemental atomic number ‘z’.
edge	Edge energy (keV) for all indices 'n' for each 'z'.
mass	Atomic mass of given atomic number ‘z’.
mass_element	Atomic mass of given element name string.
weight	Molecular weight for a given molecular formula string.
abundance	Abundance (%) for isotope string 'str' (e.g. "Mg 25").
abundance_za	Abundance (%) for isotope of atomic number 'Z' and mass number 'A'.
excess	Isotope mass excess (MeV) for Z, A.
valence	Common valence charge for a given element name string.
ion_xsect	Ion X-ray production cross-section (cm^2) for element 'Z2', shell 'Shell' and at energies (MeV) 'E'.
photo_cross_section	The photon cross-sections (photoelectric by default) for the element of atomic number Z at energy E keV, based on the tables of Hubbell and Seltzer. If 'shell' is omitted or zero, it returns total photoelectric cross-section. If 'shell' and 'subshell' are included, it returns just the cross-section for this sub-shell.

	If 'subshell' is zero, and 'shell' is not, then return total cross-section for that shell, summing all sub-shells.
photo_subshell	Sub-shell SXRF absorption cross-section (cm2) for energy 'E' and element 'z', subshell 'n'.
cross_thomson	Thomson free electron scattering cross-section that Rayleigh is built on.
cross_klein_nishna	Klein Nishna free electron scattering cross-section that Compton scattering is built on.
fluor_yield_xrf	Fluorescent yield of subshells from the tabulations of Elam.
fluor_yield	Fluorescent yield of the K, Li, Lii, Liii and Miv subshells for Z= 15 - 92 from the parameterization of M.O. Krause J. Phys. Chem. Ref Data 8 (1979) pp307.
jump_ratio_xrf	The absorption sec fluor yield jump ratio correction for SXRF, for each shell using the absorption jump-ratio of subshells from the tabulations of Elam (W.T. Elam et al., Radiation Physics and Chemistry 63 (2002) 121-128).
jump_ratio	The absorption sec fluor yield jump ratio correction for ions or SXRF.
coster_kronig	Coster Kronig transition probabilities 'n' for element 'z'.

Layer

Yield calculation

geo_array_yield	Loop geo_yield2 over all elements of an array detector to form a total X-ray yield set and average X-ray relative intensities for the array.
array_yield	Correct relative intensities for variation across a detector array.
geo_yield2	Calculate yields for all X-ray lines within a given energy range for all layers in a sample. X-ray relative intensities calculated from the yields of lines.
detector_geometry	Builds an array of structs that define geometry to each element in an array for given detector distance, global rotation (theta, phi) and global tilt angles.
experimental_angles	Calculate the directional cosines for incoming beam and outgoing X-rays to the detector.
get_lines	Get all X-ray lines and parameters, for all shells, for all line energies in the range 'e_min' to 'e_max'. Treat K, L, M as separate elements.
calc_slices2	Calculate array of *slices* (subdivision of layers) {N:n, Z:z, F:f, thick:t} based on input array of layers and the base slice thickness.
slow_beam	Slow down the incoming ion beam through all slices.
merge_slices	Merge all slices after proton end-of-range into full layers.
harden_beam	Harden a continuum X-ray beam based on progressive absorption in sample.
init_xrf_lines	Setup X-ray relative intensities for a selected beam energy.
calc_abs	Calculate cumulative μx (cmux) for all X-ray lines cumulated from surface. Can use slices of different thickness; 'cmux' returns in half-slice steps, starting at surface.
calc_cross	Calculate PIXE (or XRF) cross-sections for beam z1, a1, target z2, shell and energies E. For XRF cross-sections, will assume that all E[] are the same, unless in /continuum mode, then it's a vector across the source spectrum.
calc_yield	Calculate PIXE/SXRF yield values for all lines/elements at each slice. Lines are in descending intensity order, so major line is always index 0.
secondary_fluoresence	Calculate secondary fluorescence for a list of source lines on a list of destination elements. Do all lines for dest element.
sec_significant	Check to see if secondary fluorescence due to X-rays of energy 'e' from elements in 'slice' will be significant on destination element of atomic number 'z' and edge energy 'e_bind'. This is used to filter calculations for speed.

Pixe_fit

Fit spectra

pixe_fit	Spectrum fitting loop
pixe_setup	Set initial values for general spectrum fitting parameters.
pixe_initial	Set initial values for element X-ray peak fitting parameters.
pixe	X-ray lines function for a given set of fitting parameters.
line	Function for an X-ray line
calc_da_matrix2	Generate a DA matrix after an additional linear fit iteration.
calc_da_loop	Call “calc_da_matrix” in a loop over XANES energies and also merge scatter.
sum_peaks	Calculate sum peak lines and intensities from current parameters.
split_back	Develop the two-component “split” background
strip_clip	Main SNIP peak-stripping and background estimation routine

RBS

rutherford	Rutherford cross-section (lab frame, mb/str), with arguments THETA = scattering angle (centre of mass), PSI = scattering angle (lab.), E = beam energy (MeV). /Alt for recoil cross-section.

Stop

Ion stopping

dedx	dE/dx stopping powers for ion ‘z1,a1’ in target element ‘z2’ at energy ‘E’ (MeV). Uses Anderson and Ziegler tabulations and formulae.
dedrox	Stopping power (MeV / mg/cm^2) of particle 'Z1,A1' in composition 'layer' at energy 'E' (MeV).

Wizard support

Wizard_Load_plugins	Load all Wizard plugin SAV files, return list and titles.
define	Use “define(/wizard_notify)” to define a Wizard command node.
clear_wizard	Clears a Wizard command linked list, and frees extra nodes.
free_wizard	Clears a Wizard command linked list, frees extra nodes (like clear_wizard) and also frees the pointer and heap.
wizard_test_windows	Use `notify, 'wizard-action'` with command ‘open-test’ to probe all needed windows, which need to reply (else a warning that some are missing).
wizard_window_count	Return count of needed windows with name 'name'.
wizard_check_windows	Check if needed windows (pstate 'windows_needed' list) are open. If not pop-up warning as user guidance.
wizard_check_window_id	Check the window 'id' with name 'name' against those stored for this window 'name' in 'windows_open' ptr array indexed against names list in 'windows_needed'.
wizard_resize	Resize all widgets in a hierarchy flagged with 'xresize:fx' or 'yresize:fx' in their uvalue struct, where ‘fx’, ‘fy’ are the fractions of the size change to apply to the widget. Determine change in TLB geom scr size from previously ('oldx', 'oldy') as supplied. Don't allow TLB geom any smaller than 'minx', 'miny'. Remember to use ‘widget_info(event.top, /geometry)’ to update TLB size (stored elsewhere) after the resize has finished.
wizard_instructions_file	Read a text file to be used in the instructions panel.
figure	Display a JPEG file as a figure window for the Wizard (uses ‘picture_button’).

Fortran

Fortran support routines (in "image_lib.f") get compiled with optimization into libraries for Windows, Linux and/or Mac. The source is located in project "Fortran". Generally, tight loops and intensive bit fiddling benefits from compiled Fortran code (rather than interpreted code, such as IDL or Python). C could also be used, which is a little simpler. Fortran needs a wrapper routine for each to convert from a C 'argc,argv' call convention to a Fortran argument list (by reference). Keep Fortran code simple and use strong type declaration, because no real-time debugger options are available for debugging Fortran libs called from IDL.

All these Fortran routines have an IDL wrapper routine (PRO file) of the same name (in "main" GeoPIXE directory). This enables strict checking of variable type and calling conventions to use libraries from various hardware platforms. For many of the routines here, older ones (with a small trailing numeral in the name) are also present for historical/backward compatibility reasons. The older/superseded ones are not listed here.

Presently, there is a *Makefile* defined for the compilation of the various Windows and Linux libraries. The same *Makefile* includes instruction for compilation under Mac O/S as well.

General

geopixe_lib_version	Version of the Fortran library, access using function geopixe_library(version=v)
cmit_event_build	build a simple blog data format event stream (data format conversion)
cmit_event_maia	build a simple Maia blog data format event stream (data format conversion)
da_build_mpda_table	build table for MPDA (superseded in later MPDA?)
low_stats_filter	original Fortran implementation of the old GeoPIXE SNIP (1988) low-stats filter
throttle_q	apply Throttle filtering to an 'e' vector

Accumulation

cut_accumulate5	accumulate images from spectrum CUTs
da_accumulate_cube	accumulate compressed data cute using DA
da_accumulate_stack	accumulate 3D stack (XANES) images using DA
da_accumulate_tomo	accumulate tomo images (3D) using DA
da_accumulate11	accumulate images using DA
hist_accumulate4	accumulate a histogram (arbitrary and non-unique 'e', IDL has problems)
hist_xy	accumulate a 2D histogram
image_accumulate	Simple accumulation of simple events {x,y,e,value}
phase_accumulate	accumulate phase (not used anymore)
spec_accumulate4	accumulate spectra for regions
spec_det_accumulate4	accumulate spectra for individual detector channels
time_accumulate	accumulate time (not used anymore)
xanes_accumulate3	accumulate XANES spectra (used for APS data – 20ID?)

Device specific

get_maia_32	internal function to read bytes from input byte array for Maia device
init_maia_32	Initialize commons for 'get_maia_32'
maia_32_events2	decode blog data into event vectors for Maia 32 (1x SCEPTER) device
maia_384_events6	decode blog data into event vectors for Maia 384 (12x SCEPTER) device
maia_et2_events2	simpler decode of blog data for Maia 384 device for RT execution
maia_accumulate_dtfx2	accumulate dead-time and flux for Maia 384 device stream
maia_accumulate_dtfx2_3D	accumulate dead-time and flux for 3D stack for Maia 384 device stream
unidaq_32_events	decode raw data into event vectors for UniDAQ device
init_unidaq_32	initialize first
daq_36_events	decode raw data into event vectors for DAQ 36 (Hymod) device

init_daq_32	initialize first
daq_et_events	simpler decode of blog data for DAQ 36 device for RT execution
daq_accumulate_dtfx	accumulate dead-time and flux for DAQ 36 device stream
get_ixrf	internal function to read bytes from input byte array for Atlas iXRF device
init_ixrf	Initialize commons for ‘get_ixrf’
ixrf_events1	decode raw data into event vectors for iXRF (UQ Atlas) device
ixrf_accumulate_dtfx	accumulate dead-time and flux for iXRF device stream
get_midas	internal function to read bytes from input byte array for Midas device
init_midas	Initialize commons for ‘get_midas’
midas_events1	decode raw data into event vectors for Midas (iThemba) device
midas_accumulate_dtfx	accumulate dead-time and flux for Midas device stream
get_fx	internal function to read bytes from input byte array for FalconX device
init_fx	Initialize commons for ‘get_fx’
falconx_events4	decode raw data into event vectors for XFM FalconX device
falconx_accumulate_dtfx	accumulate dead-time and flux for FalconX device stream
falconx_accumulate_dtfx_3D	accumulate dead-time and flux for 3D stack for FalconX device stream
bd12_events	decode raw data into event vectors for Sandia Rontec BD12 format device
tohoku_events	decode raw data into event vectors for Tohoku LABO format device
mpa4_events2	decode raw data into event vectors for FastcomTec MPA4 (Madrid) device
mpa4_accumulate_dtfx	accumulate dead-time and flux for FastcomTec MPA4 device stream
mpa3_events	decode raw data into event vectors for FastcomTec MPA3 (Sandia) device
init_mdaq2	Initialize commons for ‘mdaq2_events’ (uses ‘get_fx’)
mdaq2_events	simpler decode of MDAQ2 data for UMelb MDAQ2 (LabView) device
mdaq2_accumulate_dtfx	accumulate dead-time and flux for MDAQ (labView) device stream
mdaq2_accumulate_dtfx_3D	accumulate dead-time and flux for MDAQ (LabView) device 3D stack stream
mpsys_events	simpler decode of event data for UMelb MPsys device
xsys_events	simpler decode of event data for iThemba XSYS (Indiana cyclotron) device

Wizard organization and communication

A Wizard is a window built for some special purpose, which orchestrates functions and features found elsewhere in other GeoPIXE windows and guides the user through a workflow. This description follows the hand-shaking with other windows to get desired operations performed elsewhere. The Wizard will broadcast a "wizard-action" Notify request to a nominated window program with the command requested (with qualifiers and additional data using a pointer). The idea is that the target window will recognize a “wizard-action” Notify request, check that it is addressed to it (to designated window name, not a specific window base ID), execute the requested operation and return a reply as another broadcast “wizard-return” Notify message. Again, on receipt of this Notify, the Wizard checks that it’s intended for it, then executes any call-back routine provided. It then checks for a valid ‘pnext’ pointer reference (indicating the next command in a linked list) and if found sends a Notify, "wizard-action" broadcast using the wizard command struct pointer ‘pnext’, thus continuing down the linked list of actions.

The Wizard constructs a linked list of commands to be executed using a node structure of this form (created using “node = define(/wizard_notify)”):

```
{   wizard:     '', $           ; originating wizard name
    window:     '', $           ; specific name label for destination window
    command:    '', $           ; command to be executed by window
    qual1:      '', $           ; some extra qualifier1
    qual2:      '', $           ; some extra qualifier2
    error:      0, $            ; error return code, on reply
```

```
top:        0L, $              ; return 'event.top' in case of 'open-test' command
pdata:      ptr_new(), $       ; general data to be transferred
local:      1, $               ; indicates 'pdata' managed (e.g. freed) by Wizard
callback:   '', $              ; name of call-back routine in Wizard
pnext:      ptr_new() }        ; next action node in linked list
```

Note: The source code for "wizard_depth" and "wizard_standards" contain lots of comments to explain the approach, events, call-backs and commands.

An example is the building of a command list in wizard_depth to compare two image data-sets using the "compare" feature in *Image RGB*, which loads another image to compare and then sets R,G,B element planes using elements from the original or compare image data-sets. In the *Depth Wizard*, this is used to overlay element `(*pstate).rgb_map_element` for the "inner" detectors in Red and the "outer" detectors in Green to show a depth contrast (deep features appear red while shallow appear yellow and intermediate depth is a mixed Red-Yellow shade, such as orange). The first command ('load-image') is aimed at the 'Image' window, to load the "inner" image (the *Image RGB* window gets Notified of this change and sees this image data), while the second command ('compare-image') is aimed at the 'Image RGB' window, to load the "outer" image for comparison and sets the R and G planes.

The commands are linked together using 'pnext' and stored in one of the Wizard heap storage areas allocated for Notify messages ((*pstate).pwizard4). This storage area is cleared first (to free any pointer data from a previous usage). The command is then sent using Notify to the *Image* window. The use of a number of separate heap storage variables (pwizard…) in pstate for different purposes in the Wizard ensures that referenced data is not freed prematurely while still in play elsewhere.

Code fragment from wizard_depth (**wizard_depth_rgb_display** routine) to compare two image data-sets using "compare" feature in *Image RGB*

```
wz = define(/wizard_notify)  ; create a command struct for Notify, "wizard-action"
wz.wizard = 'depth'
wz.window = 'Image'
wz.command = 'load-image'
wz.pdata = ptr_new( (*pstate).rgb_map_inner_file)  ; "inner" images file
wz.local = 1                                       ; pdata will be freed by wizard
p = ptr_new(wz, /no_copy)
pl = p
p0 = p                                             ; head command in list

wz = define(/wizard_notify)
wz.wizard = 'depth'
wz.window = 'Image RGB'
wz.command = 'compare-image'
wz.qual1 = (*pstate).rgb_map_element
wz.pdata = ptr_new( (*pstate).rgb_map_outer_file)  ; "outer" images file
wz.local = 1                                       ; pdata will be freed by wizard
p = ptr_new(wz, /no_copy)
(*pl).pnext = p                                    ; link this command to first
pl = p

clear_wizard, (*pstate).pwizard4                   ; clear previous data
*(*pstate).pwizard4 = *p0                          ; copy struct *p0 to send heap
ptr_free, p0
notify, 'wizard-action', (*pstate).pwizard4
```

One issue with Wizards is knowing if a necessary target window is open and available to use. This is handed (in "wizard_test_windows") by sending periodically a 'open-test' Notify, 'wizard-action' message to all windows that are needed by the wizard (list stored in (*pstate).windows_needed). The target windows

recognize this command and simply return their top-level base widget ID. The wizard maintains a list of the IDs of target windows (and checks whether they are valid, i.e. still open). If any are missing it warns the user to open them. If a duplicate is found, it warns the user to close one (this avoids multiple windows responding and executing the same command).

As an example of the response to a 'wizard-action' Notify, this is a code fragment to show how the *Image* window responds to either 'open-test' or 'load-image' commands sent as Notify, 'wizard-action' messages. When the command is done (or fails with an error), a Notify, 'wizard-return' message is sent as a reply to the originating Wizard.

Portion of event routine that responds to 'wizard-action' Notify events (in "OnNotify_image")

```
'wizard-action': begin
	if ptr_valid( event.pointer) then begin
		if (*event.pointer).window eq 'Image' then begin
			case (*event.pointer).command of
				'open-test': begin
					pw = (*pstate).pwiz
					*pw = *event.pointer
					(*pw).top = event.top
					(*pw).error = 0
					notify, 'wizard-return', pw
					end

				'load-image': begin
					pw = event.pointer
					file = *(*pw).pdata
					Image_Load2, pstate, file, error=err
					(*pw).error = err
					notify, 'wizard-return', pw
					end
```

The next piece of code shows how *wizard_depth* responds to the 'wizard-return' Notify messages. In the case of 'open-test', it calls 'wizard_check_window_id' to check the returned window against those that are desired and open and returns the 'count' of how many of this type ((*pw).window) are available. If the count is 2 or more it warns the user to close one. The 'open-test' actions are requested periodically to ensure the correct number of windows are open.

For any other action that originated in this wizard, it first looks for a call-back routine to execute (uses the IDL *call_procedure* routine to execute procedure named in ((*pw).callback) and then checks to see if there is another command in the list, linked via (*pw).pnext. If it is valid, then it sends another Notify, 'wizard-action'. If none are available, the Wizard command sequence has finished.

Wizard_depth event code in case statement: `case tag_names( event,/structure) of'NOTIFY'`

```
;	These are the events returned from GeoPIXE windows after 'wizard-action' operations.
;	We check for an error return, and for callback operations to perform, and then if there
;	is a next 'wizard-action' event to send.

'wizard-return': begin
	if ptr_valid( event.pointer) then begin
		pw = event.pointer

		if (*pw).wizard ne 'depth' then begin
			(*pstate).windows_veto = 5	; veto tests while this pop-up is open
							; count down gives time to close some windows
			warning, 'wizard_depth_return', ['Another Wizard appears to be open', $
					'This will cause many problems.']
			goto, finish
		endif
```

```
;               A reply from a window to 'open-test' shows that it is open ...
;               increment a count to test if unwanted duplicates are open.

                if (*pw).command eq 'open-test' then begin
                        wizard_check_window_id, needed=(*pstate).windows_needed,
                                open=(*pstate).windows_open, name=(*pw).window, id=(*pw).top, $
                                count=count, error=error
                        if error then goto, finish

                        if count gt 1 then begin
                                (*pstate).windows_veto = 4    ; veto tests for a while
                                                              ; gives time to close some windows
                                warning, 'wizard_depth_return', ['Multiple windows open.', '', $
                                        'This may cause problems.','Close any duplicate windows.']
                        endif
                        goto, finish
                endif

                if (*pw).error then goto, finish              ; on error do not continue

;               If there is a callback, execute it, passing the returned wizard notify pointer

                if (*pw).callback ne '' then begin
                        call_procedure, (*pw).callback, pstate, pw, error=error
                        if error then goto, finish
                endif

;               If there is a valid notify struct pointed to by 'pnext' in the returned event,
;               then notify that in turn ...
;               Note that we don't free linked list memory here.

                if ptr_good( (*pw).pnext) then begin
                        notify, 'wizard-action', (*pw).pnext
                endif
        endif
        goto, finish
        end
```

Parallel processing routines

Parallel processing (e.g. for sorting raw data into images using DA) uses some routines in the foreground GeoPIXE process to orchestrate the process (divide the problem into parts, launch the background processes display a progress bar, and assemble the results) and ‘worker’ routines that run in the launched background processes (*IDL Bridge Objects*) to do the work on their part of the problem.

Looking at routine “evt_start”, which is called from “evt” to sort raw data into images, after making sure entered parameters are correct (e.g. it makes sure all file paths are found, perhaps searching for them, otherwise it pops up file requesters) and then to do the sort, it can call “da_evt” directly if no “cluster” mode is selected to use parallel processing. If “cluster” mode is enabled, it builds a long command and argument string using “stringify” (including the command name “da_evt”) and passes that to “cluster_client” to orchestrate parallel processing. This uses routines in “parallel _client” to launch background processes using the IDL Bridge object and creates some shared memory segments to communicate the long argument string and collect results. The shared memory also has flags to control communication. The background processes execute “geopixe_parallel” to execute the passed argument string, which includes the command name. It flags that it’s finished and returns results through the shared memory. ‘Cluster_client‘ monitors progress, displaying the fraction completed and estimated processing time remaining and detects the done flags from the background processes and gathers results. It then assembles the parts of the resulting images into the full resulting image for display (in “cluster_merge_images”).

A similar process is used to extract spectra from image regions, by sorting through raw data, and assemble the final spectra for *Image Regions*. This is done in the *Image Regions* window in routine “Image_Table_EVT” (“image_table_eventcb” PRO file), which constructs a command string (“spec_evt” command) and uses “cluster_client” to execute it across a set of background processes. Once finished, the contributions to each spectrum are accumulated from the various background processes to build the final spectra (in “cluster_merge_spectra”).

GeoPIXE routines

cluster_client	Foreground control of parallel processing, including a progress bar, and set-up of shared memory using the routines in "parallel_client".
cluster_merge_images	Merge the parts of the images (stripes) returned from each process.
cluster_merge_spectra	Merge the contributions to region spectra returned from each process.
parallel_client	Library of routines used by ‘cluster_client’ to launch, monitor, control and free processes and shared memory.
geopixe_execute	Decode (potentially very long) command line string for command, keywords and arguments and use IDL 'execute' to execute a command-line. Called from background process “geopixe_parallel”.

Parallel worker routines

geopixe_parallel	The executive routine for background processing in GeoPIXE. Uses the IDL "execute" command (see routine “geopixe_execute”), so must use a full IDL license, as VM mode does not support "execute".
parallel_worker	Library of routines used by ‘geopixe_parallel’ to communicate back to foreground controls (e.g. completion flags, results, progress status).

Eclipse environment and organization

The Eclipse environment for the IDL Development Environment (IDLDE) uses a “workspace” to contain all “projects”, which can be built separately. It manages a PATH, so that any open project is on the path, and so when debugging, IDL can find and compile any routine as it goes.

*This is nice but has some gotchas. Specifically, it will load any compiled SAV file that it finds on the path, if it has the **same name as a routine that is encountered** and not yet compiled. This is not a good way to execute the latest version of code, and so we need to avoid this scenario. For this reason, we try to **give SAV files a different name to ALL routines** in GeoPIXE, to keep them distinct. Hence, “gimage.pro” main GeoPIXE routine is compiled to “GeoPIXE.sav” and so a reference to ‘gimage’ does not trigger loading of GeoPIXE.sav, but instead compiles “gimage.pro”, if it’s not already compiled. The file “gimage.pro” also has an entry point “GeoPIXE” if executed from the SAV file with name “GeoPIXE.sav” under runtime control.*

Tip: In the Eclipse IDLDE environment, use the “Search” function to look for all occurrences of a proposed SAV file name to check whether it has been used for a procedure or function already.

Because of these considerations, a separate runtime directory “geopixe” is used, separate from the main source directory “main”. Furthermore, make sure that the runtime project “geopixe” is never opened in IDLDE when compiling, and that the Preference properties for “geopixe” do NOT add it to the path if opened.

NOTE: The name “geopixe” of the runtime directory is assumed throughout. Please do not rename this.

Directory organization

The root directory is called “Workspace”. All projects are in directories under workspace. Each Project is built separately into a program or a plugin/library of some sort (plugin, library, wizard), which needs to be loaded into one of the GUI windows.

In Eclipse, the “**properties**” of each project define how it is compiled (for our Open Source download from GitHub, where these properties are not defined, it may be preferable to use the **Builder.pro** approach, as described below under “Building GeoPIXE”). Some simple examples illustrate the cases in GeoPIXE. Most compile everything within the directory of the project only (e.g. devices, plugins, wizards) and do not execute “resolve_all”, which will go looking for anything remaining and unreferenced and compile that too. Only for GeoPIXE itself, and some standalone utilities, do we do this.

GeoPIXE “main” programs

The main programs and windows and support files are located in project “main”. This includes Util routines used by many of the other projects and plugins. Once compiled as “GeoPIXE.sav” (stored in “geopixe” folder, see below) it often needs to be restored to provide these routines in many of the other project. In a sense, as well as containing the main GeoPIXE program (from “gimage.pro”), GeoPIXE.sav is also needed as the main library file for many projects.

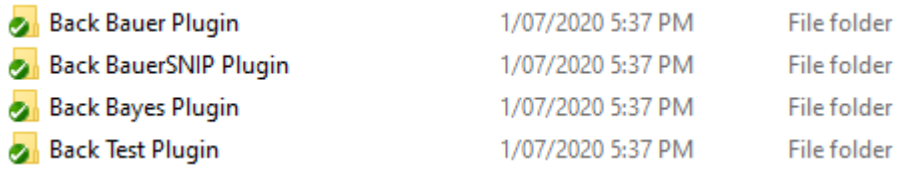
“geopixe” runtime files

A separate project “geopixe” contains runtime files, including GeoPIXE.sav and the compiled **Fortran libraries** used for speed up of intensive data operations and compiled for Windows (32 and 64 bit), Linux (32 and 64 bit) and Mac. As mentioned, “GeoPIXE.sav” often needs to be restored to provide these routines in many of the other project groups (e.g. foreground and background processes for the Maia, DAQ detectors and Maia Mapper, spectrum, background and image plugins and wizards). In a sense, as well as containing the main GeoPIXE program (from “gimage.pro”), GeoPIXE.sav is also needed as the main library file for all projects. But generally, the plugins and wizards are called from GeoPIXE and so GeoPIXE.sav and all its routines have already been loaded. The exceptions are the background processes (e.g. for Maia, DAQ), where the code starts by loading GeoPIXE.sav (and probably a relevant device object).

Perhaps crudely at present, all Fortran routines are compiled from a single source file (“image_lib.f” in project “Fortran”) into the various library files stored here. So all source, including for some of the Device Objects, are collected in a single Fortran source file. However, this makes loading the correct Fortran libraries simple from a single file, based on host hardware platform. The “geopixe” runtime directory also contains runtime data, such as detector layout, and the subdirs for the SAV files for plugins, Maia/DAQ processes, wizards and device objects, etc. The program also recognizes additional local filters and detectors stored on “config/filters” and “config/detectors” paths.

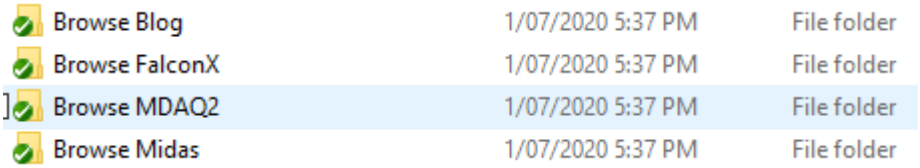
Background plugins

The Background algorithm plugins generate SAV file with names of the form “*_back_plugin.sav” are stored in the “geopixe/plugins” directory. They are read in by *X-ray Spectrum Fit* (“fit_setup.pro”) to provide different options for forming an approximation to underlying spectrum background. The project directories look like …



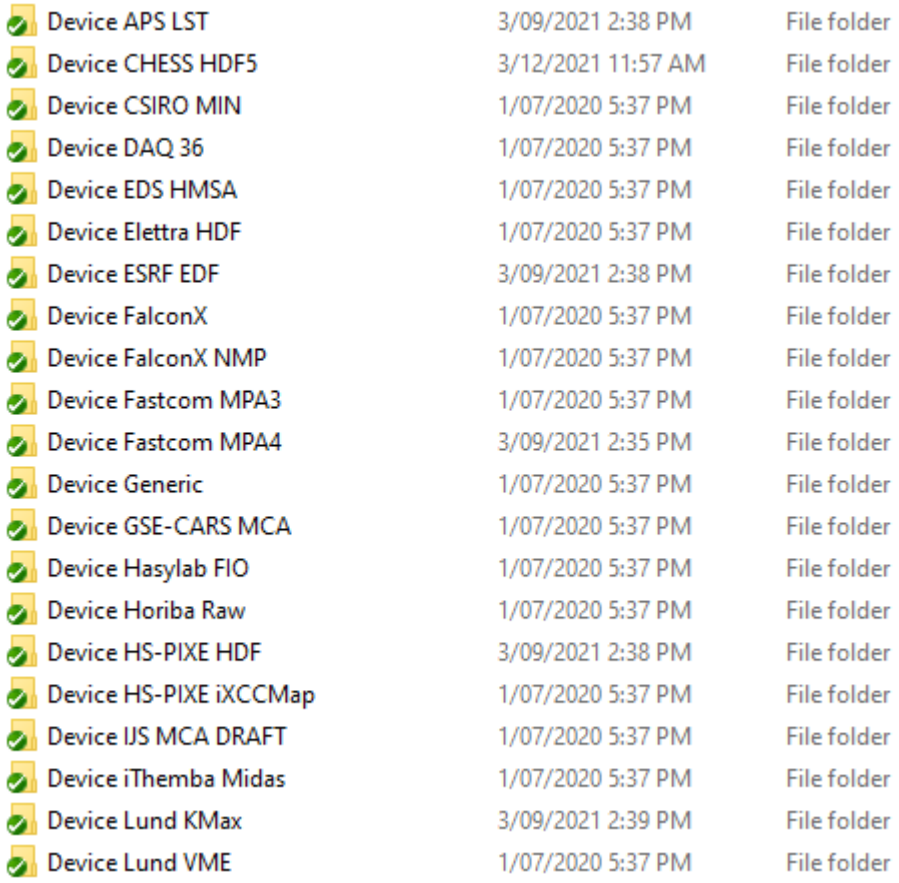
Raw data browsers

Modelled on the code and approach of *blog_browser* (Maia detector), some variants have been developed for other data (e.g. FalconX at Australian Synchrotron, MDAQ2 at UMelb, Midas format at iThemba Labs). These project directories look like …



Device Objects

Projects for the Device Objects used to provide import filters/processing for various raw data formats. The created device object SAV files ("*_device__define.sav") are stored in the runtime "geopixe/interface" directory and loaded using "define_devices" into common, which is typically called from GeoPIXE ("gimage.pro"). It will also run from various other routines, if not already loaded, enabling them to be tested standalone.



Device Maia	3/09/2021 2:39 PM	File folder
Device Maia NMP	1/07/2020 5:37 PM	File folder
Device Maia XYabs	1/07/2020 5:37 PM	File folder
Device MDAQ2	1/07/2020 5:37 PM	File folder
Device MicroDAQ	1/07/2020 5:37 PM	File folder
Device MicroDAQ HDF	1/07/2020 5:37 PM	File folder
Device MPsys	1/07/2020 5:37 PM	File folder
Device MPsys Unix	1/07/2020 5:37 PM	File folder
Device NAC XSYS	1/07/2020 5:37 PM	File folder
Device NSLS HDF	1/07/2020 5:37 PM	File folder
Device NSLS MARS Ge	1/07/2020 5:37 PM	File folder
Device NSLS MCA	1/07/2020 5:37 PM	File folder
Device NSLS NetCDF	1/07/2020 5:37 PM	File folder
Device NSLS XFM H5	1/07/2020 5:37 PM	File folder
Device OM DAQ	3/09/2021 2:40 PM	File folder
Device PNC-CAT HDF	1/07/2020 5:37 PM	File folder
Device PNC-CAT HDF devAW	1/07/2020 5:37 PM	File folder
Device Primecore U48	1/07/2020 5:37 PM	File folder
Device Sandia BD12	1/07/2020 5:37 PM	File folder
Device Sandia EVT	1/07/2020 5:37 PM	File folder
Device Sandia MPAWIN	1/07/2020 5:37 PM	File folder
Device SLS MCA	1/07/2020 5:37 PM	File folder
Device Spring8 37U	19/07/2021 12:38 PM	File folder
Device Tohoku Labo	1/07/2020 5:37 PM	File folder
Device UQ iXRF	27/09/2021 4:08 PM	File folder
Device Wakasa UniDAQ	1/07/2020 5:37 PM	File folder
Device Zagreb LST	1/07/2020 5:37 PM	File folder

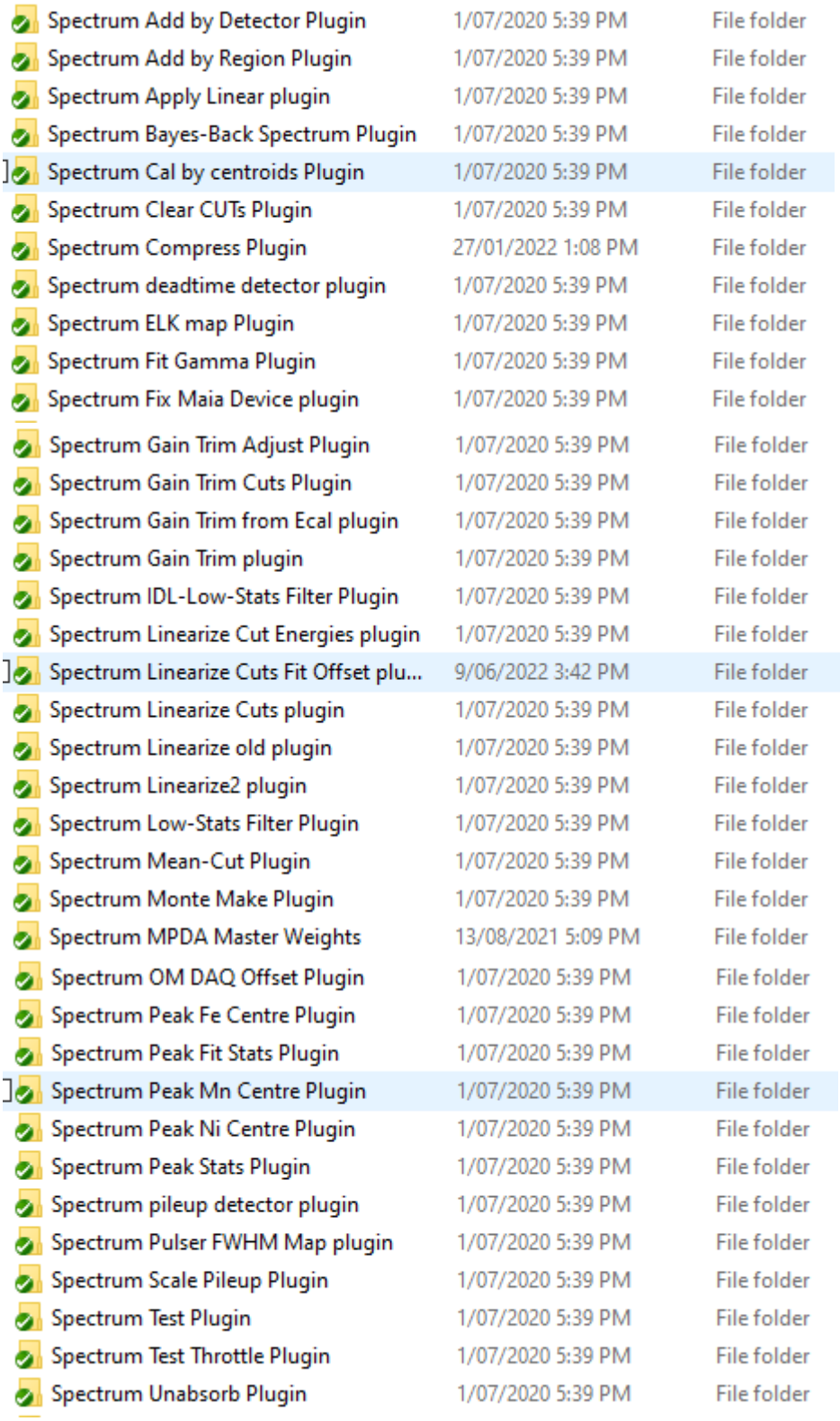
Image plugins

Projects for specific image processing plugins. The generated SAV files ("*_image_plugin.sav") are stored in the runtime "geopixe/plugins" directory and loaded into gimage.pro and accessible from menus.

Image Align Plugin	1/11/2021 7:57 PM	File folder
Image Blocky Plugin	1/07/2020 5:39 PM	File folder
Image BMP Read Plugin	1/07/2020 5:39 PM	File folder
Image Combine Align Plugin	1/07/2020 5:39 PM	File folder
Image Conc Offset Plugin	1/07/2020 5:39 PM	File folder
Image Conc Stats test Plugin	1/07/2020 5:39 PM	File folder
Image Correct Encoder Plugin	1/07/2020 5:39 PM	File folder
Image Cross Corr Plugin	1/07/2020 5:39 PM	File folder
Image Detector Histogram Plugin	1/07/2020 5:39 PM	File folder
Image Even Rows Plugin	1/07/2020 5:39 PM	File folder
Image EVT read template Plugin	1/07/2020 5:39 PM	File folder
Image Export Flat Plugin	1/07/2020 5:39 PM	File folder
Image FFT Play Plugin	27/07/2020 3:11 PM	File folder
Image FFT Plugin	1/07/2020 5:39 PM	File folder
Image Flatten to Compton Plugin	1/07/2020 5:39 PM	File folder
Image Flatten to Dwell Plugin	1/07/2020 5:39 PM	File folder
Image Flatten to Elastic Plugin	1/07/2020 5:39 PM	File folder
Image Flatten to Flux0 Plugin	1/07/2020 5:39 PM	File folder
Image Flatten to Flux1 Plugin	1/07/2020 5:39 PM	File folder
Image Flatten to Smoothed Elastic Plu...	1/07/2020 5:39 PM	File folder
Image flux Plugin	1/07/2020 5:39 PM	File folder
Image Move Blog Plugin	18/12/2020 1:41 PM	File folder
Image PCA Plugin	1/07/2020 5:39 PM	File folder
Image Phase Distance Plugin	7/12/2021 4:22 PM	File folder
Image Phase maps Plugin	1/07/2020 5:39 PM	File folder
Image raw count_rate Plugin	1/07/2020 5:39 PM	File folder
Image raw dead_fraction Plugin	1/07/2020 5:39 PM	File folder
Image raw dwell_map Plugin	1/07/2020 5:39 PM	File folder
Image raw flux Plugin	1/07/2020 5:39 PM	File folder
Image raw pileup_map Plugin	27/07/2022 2:10 PM	File folder
Image Reassemble XAN plugin	1/07/2020 5:39 PM	File folder
Image Shear Correct Plugin	1/07/2020 5:39 PM	File folder
Image subtract plugin	1/07/2020 5:39 PM	File folder
Image Sum REE plugin	1/07/2020 5:39 PM	File folder
Image table merge plugin	1/07/2020 5:39 PM	File folder
Image Video Play Plugin	18/03/2021 12:48 PM	File folder
Image Yield maps Plugin	1/07/2020 5:39 PM	File folder

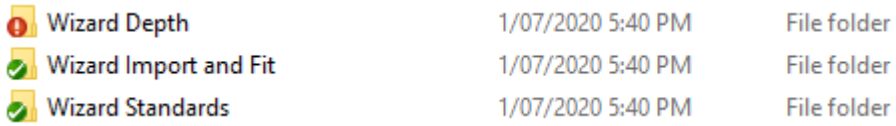
Spectrum plugins

Projects for specific spectrum processing plugins. The generated SAV files ("*_spectrum_plugin.sav") are stored in the runtime "geopixe/plugins" directory and loaded into spectrum_display.pro and accessible from menus.



Wizard projects

Special user interface program Wizards to guide users through complex tasks. The generated SAV files ("*_wizard.sav") are stored in the runtime "geopixe/wizards" directory and loaded into gimage.pro and accessible from menus.

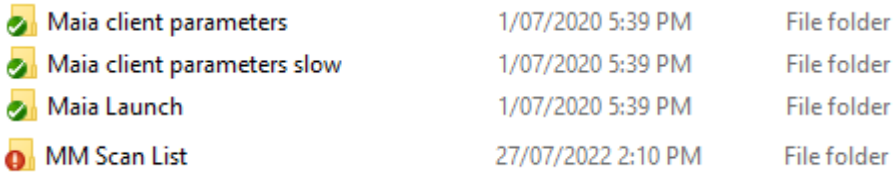


Maia background processes for Blog data

Projects for the Background processes that access raw blog data from the blog server used for the *Maia Control* program to monitor and control the Maia detectors are stored in the runtime "geopixe/maia" directory. These include processes to track detector count-rates across the Maia array (activity), accumulate Dynamic Analysis (DA) records to assemble real-time images (DA), collect Epics records for charge and flux (Epics), accumulate spectra for all detector channels (ET2 spectra), accumulate spectra for the 'groups' of detector channels (group spectra) and a version of DA, which can simulate the DA records by reading from a file (file DA).

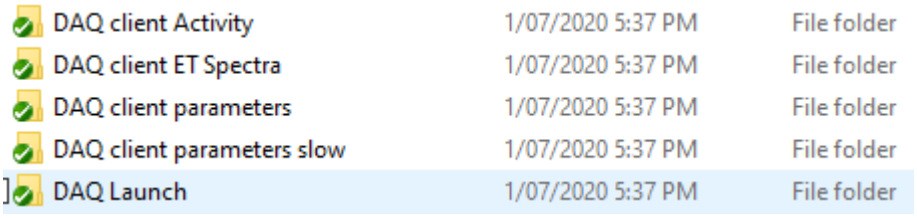
Maia Control and background processes for Kandinski data from Maia

Projects for *Maia Control, Scan List* and the Background processes that access Maia parameters via a TCP/IP port to Kandinski running on the Maia Hymod processor, which are used for the *Maia Control* program to monitor and control the Maia detectors, are stored in the runtime "geopixe/maia" directory. These include update of a copy of many Kandinski parameters (parameters), seldom changed Kandinski parameters (parameters slow), the *Maia Control* program GUI (Maia Launch), the *Scan List* scan sequence control GUI (mm_scan_list), stored in the runtime "geopixe" directory.



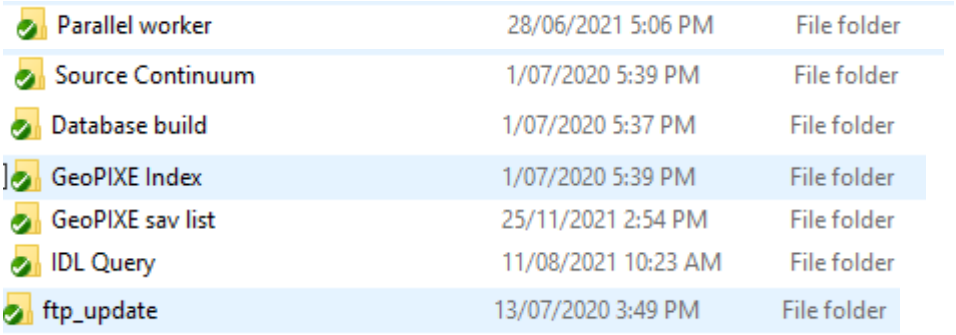
DAQ background processes for DAQ data

Projects for the Background processes used for the DAQ-Control program to monitor and control the Hymod SCEPTER interface used for general purpose detectors on the Nuclear Microprobe (NMP) at UMelb are stored in the runtime "geopixe/daq" directory. These include processes to track detector count-rates across a 32-channel array interfaced to SCEPTER plus 4 direct connect serialized ADC channels (activity), accumulate spectra for all detector channels (ET spectra), update of a copy of many Klee parameters (parameters), seldom changed Klee parameters (parameters slow), and the *DAQ Control* GUI (DAQ Launch), stored in the runtime "geopixe" directory.



Special purpose projects

Some miscellaneous projects include “parallel worker”, which is spawned by *Parallel Client*, using the IDL Bridge Object, for each parallel instance of the background processing, “Source Continuum”, the X-ray source modelling routines, “Database build”, which provides a project hook for building the GeoPIXE database from source data (data sourced from “main/database/*” and output data SAV file “geopixe2.sav” stored in runtime “geopixe” directory), “GeoPIXE Index”, which is a stand-alone tool to scan a directory tree for image DAI files and generate a HTML web page of images for each and then build a top-level table format summary web page (“index.html” file) for the directory, “sav list” to print the contents of all SAV files, “IDL Query”, which pops up information about the IDL and GeoPIXE environment, “geopixe_update”, which are the source routines for the GeoPIXE code FTP site retrieval GUIs (“geopixe_update” and “maia_update”).



Old projects

Some old projects, which may be out of date …

Handling multiple workspaces

IDL does not seem to handle the idea that we’d like to work in very separated workspaces at times with nothing in common. In GeoPIXE we can have quite separate versions being managed at various times. Now GeoPIXE-source2 (version 7.6) is aimed at MM compatibility using python 2.7 and IDL 8.5.1, while GeoPIXE-source3 (version 8.6) and GeoPIXE-open-source (version 8.6) uses py3.8 and IDL 8.8. And some features were staged and not in all together.

To do this we have organized 3 workspaces (“GeoPIXE-source2/workspace” and “GeoPIXE-source3/workspace” and “GeoPIXE-open-source/workspace”).). To ensure that we can launch IDL Eclipse for them separately, and select the correct IDL and python versions, we use a script located in the top level dirs: “GeoPIXE-source2”, “GeoPIXE-source3” and “GeoPIXE-source-open-source”. For GeoPIXE-source3 (“*IDLDE 88 Python38 GeoPIXE source3*.bat”), it looks like this under Windows (to set the correct python and a different ‘prefs’ file for IDL launch) … (note this is just 4 lines of code in the BAT file without the line breaks here)

```
call c:\Anaconda3\condabin\conda.bat activate base

PATH=C:\Program
Files\Harris\IDL88\bin\bin.x86_64;C:\Anaconda3;C:\Anaconda3\Scripts;
%PATH%

SET PYTHONPATH=C:\Program
Files\Harris\IDL88\bin\bin.x86_64;C:\Program
Files\Harris\IDL88\lib\bridges;C:\Anaconda3; …; %PYTHONPATH%

"C:\Program Files\Harris\IDL88\bin\bin.x86_64\idlde.exe" -data
"C:\Software\IDL\GeoPIXE-open-source\Workspace" -
pref="C:\Software\IDL\GeoPIXE-open-source\Workspace\idl88-py38.pref"
```

The file "idl88-py38.pref" contains the IDL environment setup for the instance, which sets the initial search PATH in IDLDE, root directory, startup script and command prompt (so you know which is running in the Eclipse window) … (note this is just 4 lines of code in the PREF file)

```
IDL_PATH : <IDL_DEFAULT>

IDL_STARTUP : C:\software\IDL\GeoPIXE-open-
source\workspace\main\startup.spro

IDL_START_DIR : C:\software\IDL\GeoPIXE-open-source\workspace\main

IDL_PROMPT : "python3.8 IDL 8.8 open> "
```

Dealing with SAV files at compile time.

As mentioned above, a complication in this approach is that during a build IDL will automatically load any SAV files that it finds in the specific project dir tree, which is mostly NOT what we want. The SAV files are now located in the runtime dir "geopixe" in the Open Source version of GeoPIXE. This avoids most of these issues, as "geopixe" is never added to the path (make sure that in the Preferences for project "geopixe" that the Path is never updated when it opens/closes).

IDL DE preferences

From the menu "Windows→Preferences", these need to be set correctly. Normally

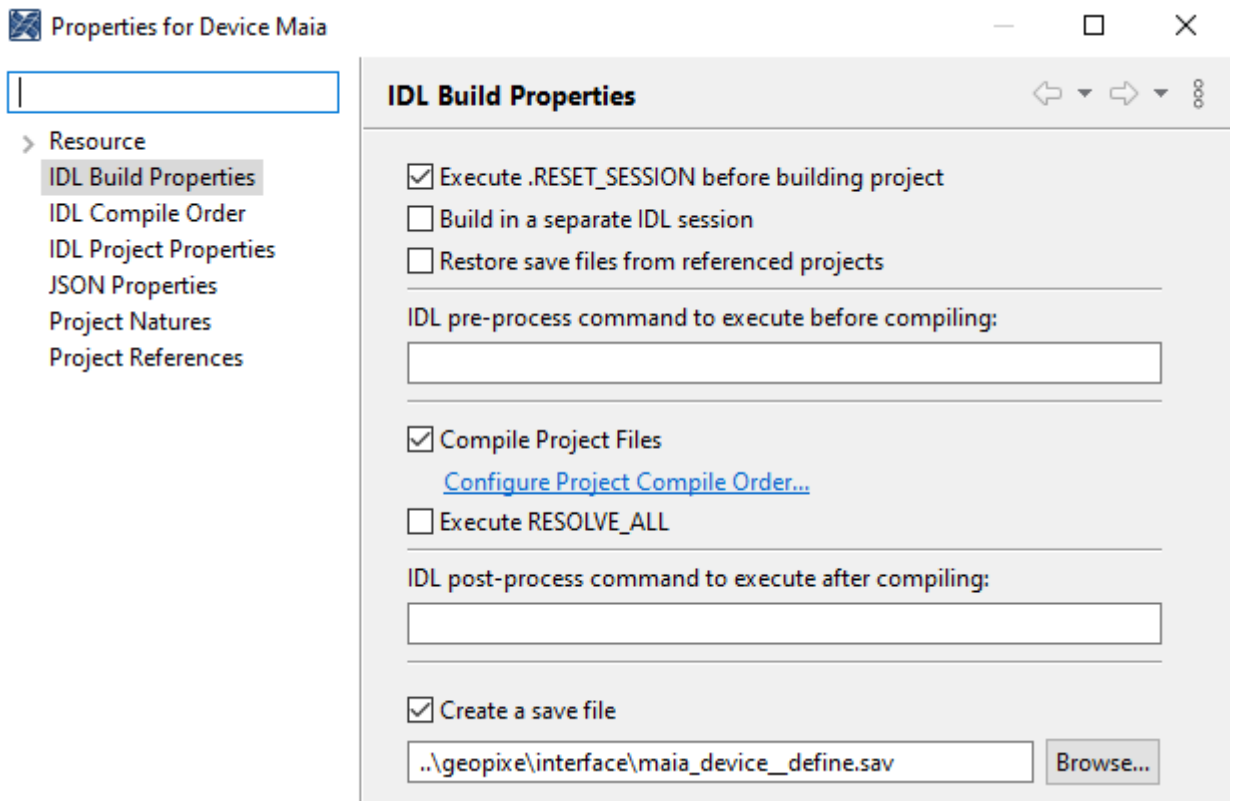
1. General
    a. Editors
        i. Text editors
            1. Display tab width = 4
            2. Do NOT use "insert spaces for tabs".
2. IDL
    a. Editor
        i. Display tab width = 4
        ii. Do NOT use "insert spaces for tabs".
    b. These will probably come from the "PREF" file, if started using the script.
        i. IDL prompt (e.g. "python3.8 IDL 8.8 open>")
        ii. Startup file (e.g. "**startup.spro**" in geopixe main dir)
        iii. Initial working directory (e.g. main workspace/GeoPIXE dir)
    c. Paths (these get set by the launch script PREF file)
        i. <IDL_DEFAULT>
3. Run/Debug

   a. Console
      i. Console buffer size (characters) = 10,000,000

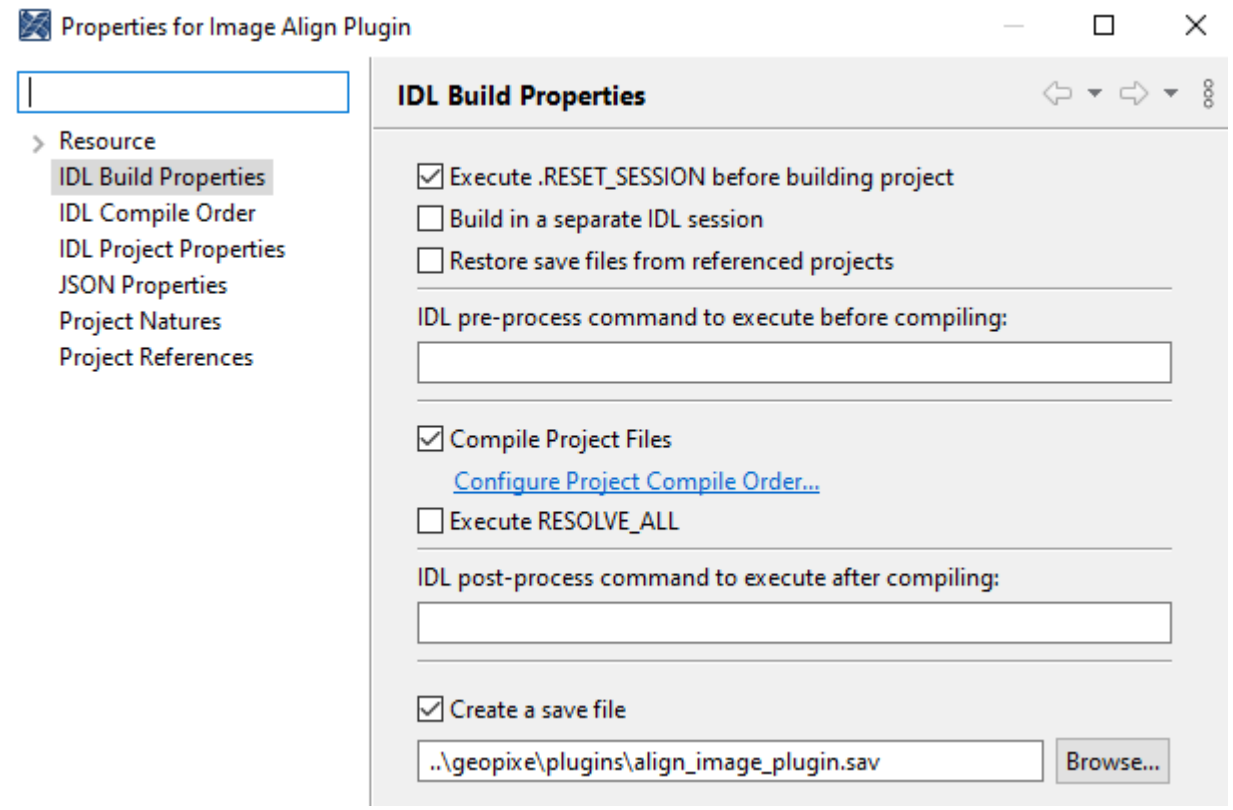
Project Properties

If you right-click on any Project title and select “Properties”, you can set the “IDL Build Properties”, which are essential to include the correct content and save a SAV file to the right place. Setting these properties is not essential, as you can use the “builder” PRO (or SAV file in runtime) to construct a script for the build. See the notes on compilation of GeoPIXE below. The critical elements of the IDL Build Properties are:

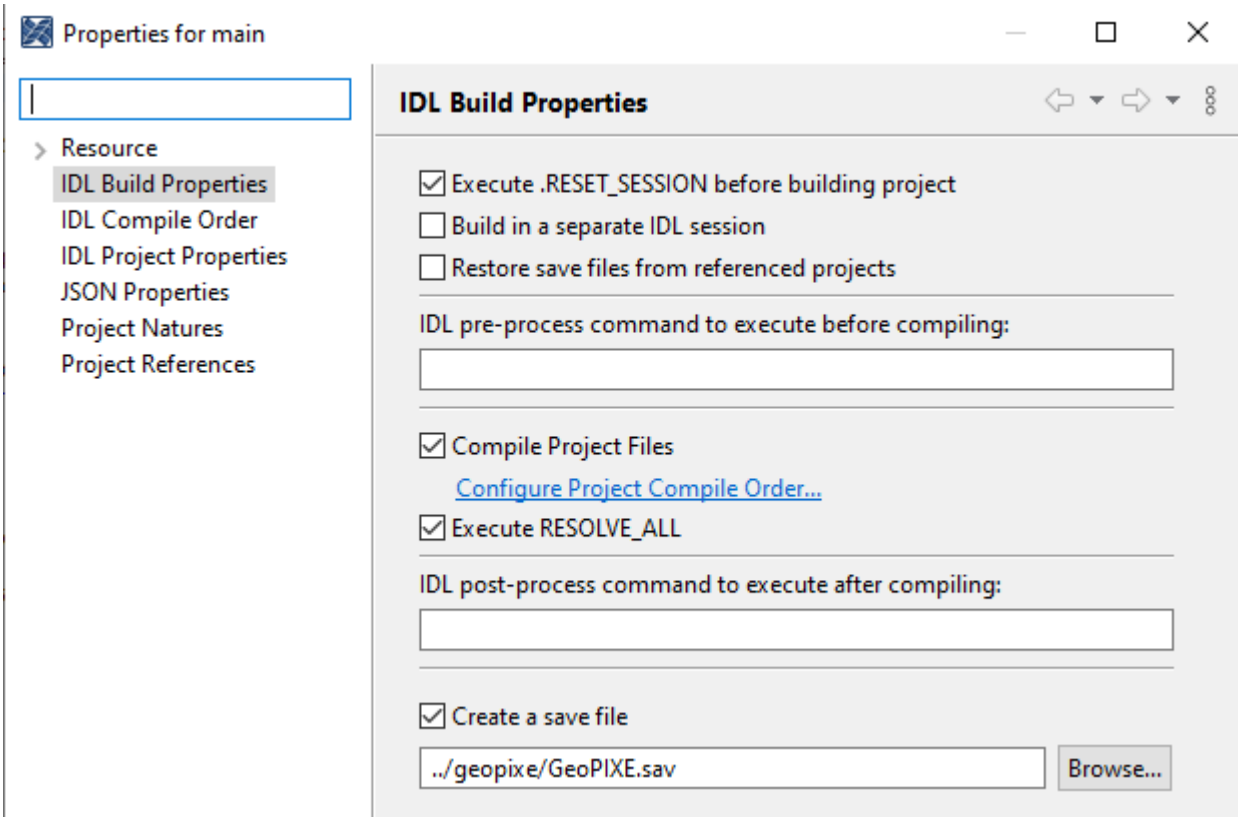
1. Execute .RESET_SESSION
   a. Is usually set to clear out all compiled routines and data.
2. IDL pre-process command
   a. Sometimes used to do something after a build
3. Execute RESOLVE_ALL
   a. This will search for any routine name referenced across the entire PATH and compile a routine PRO file of the same name, if it exists.
   b. This is required to build GeoPIXE itself.
   c. But we DO NOT do this for plugins (image, spectrum, background, wizard, device) and background processes (e.g. Maia, DAQ) because we don’t want GeoPIXE main routines to be compiled into these plugins – these are provided by restoring “GeoPIXE.sav” at run-time.
4. IDL post-process command
   a. Sometimes used to do something after a build
5. Create a save file
   a. The output SAV filename and full path.



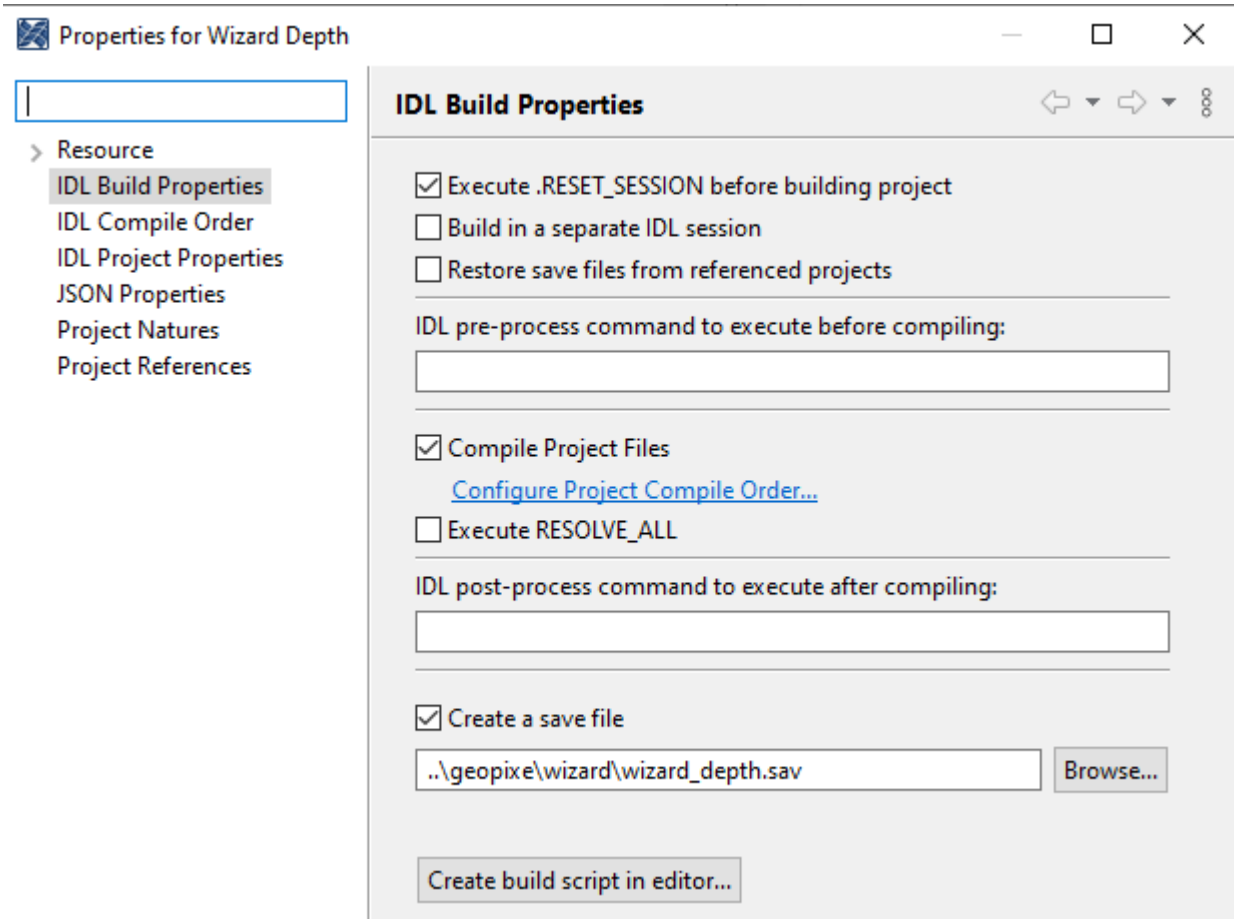
Properties for building the Maia device



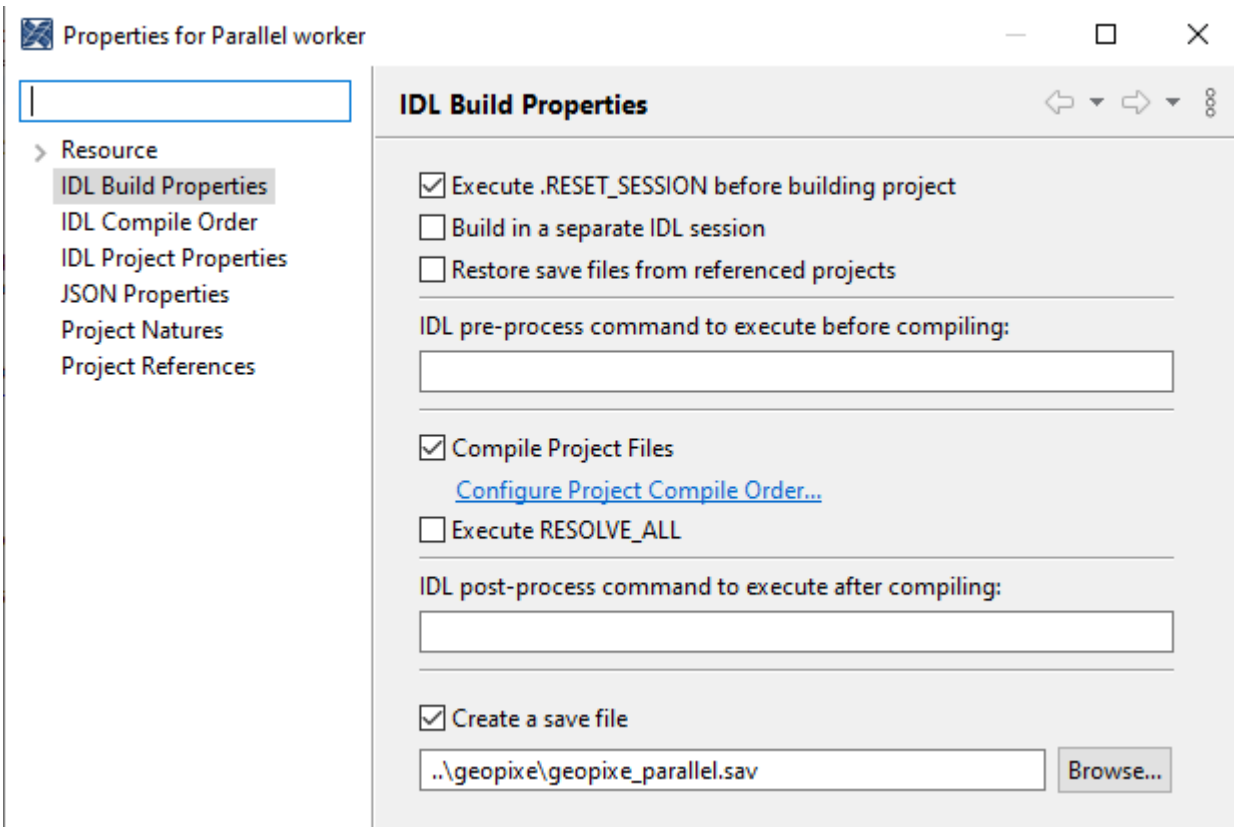
Properties for building the Image Align plugin



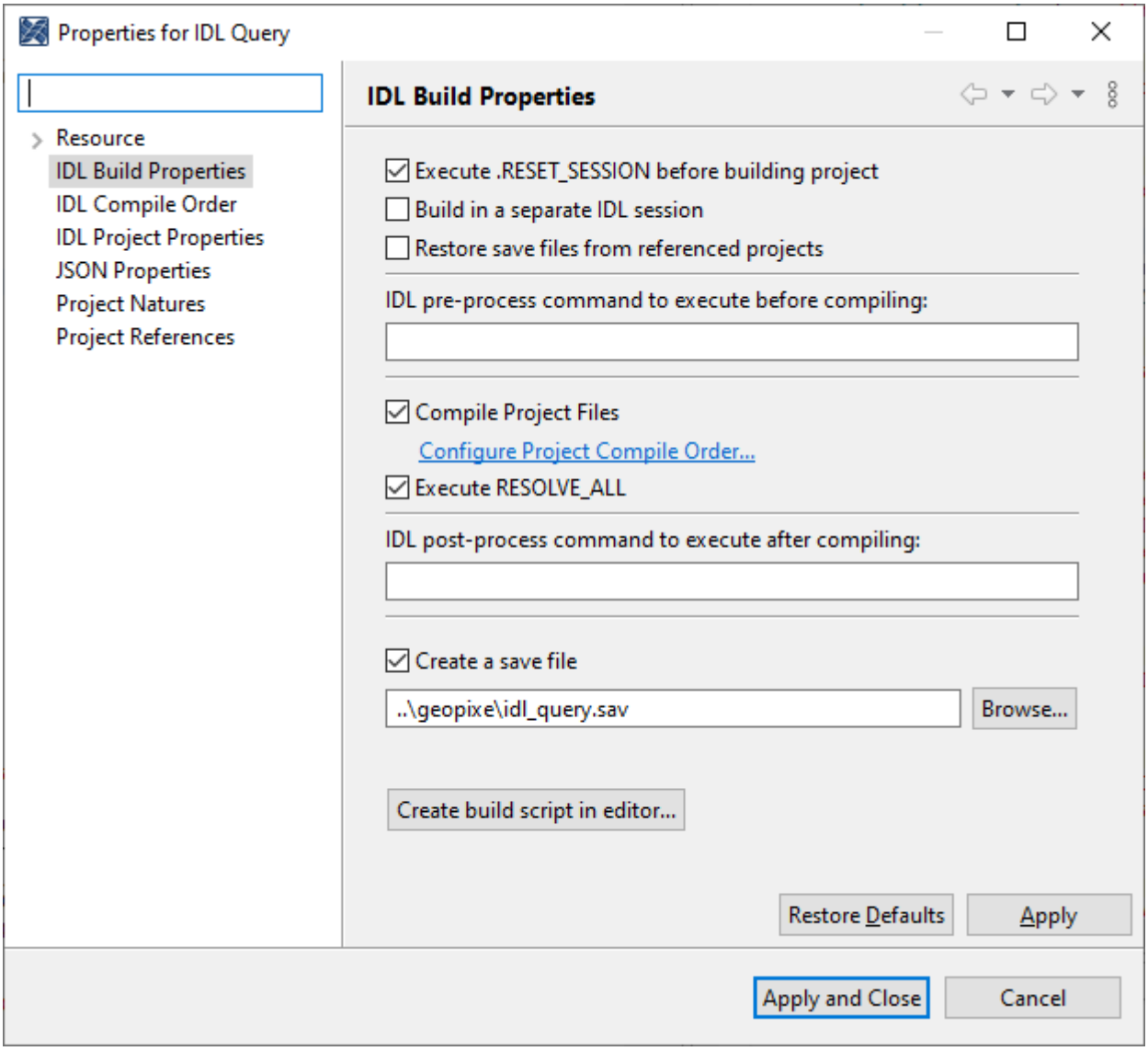
Properties for building the project "main" GeoPIXE.sav



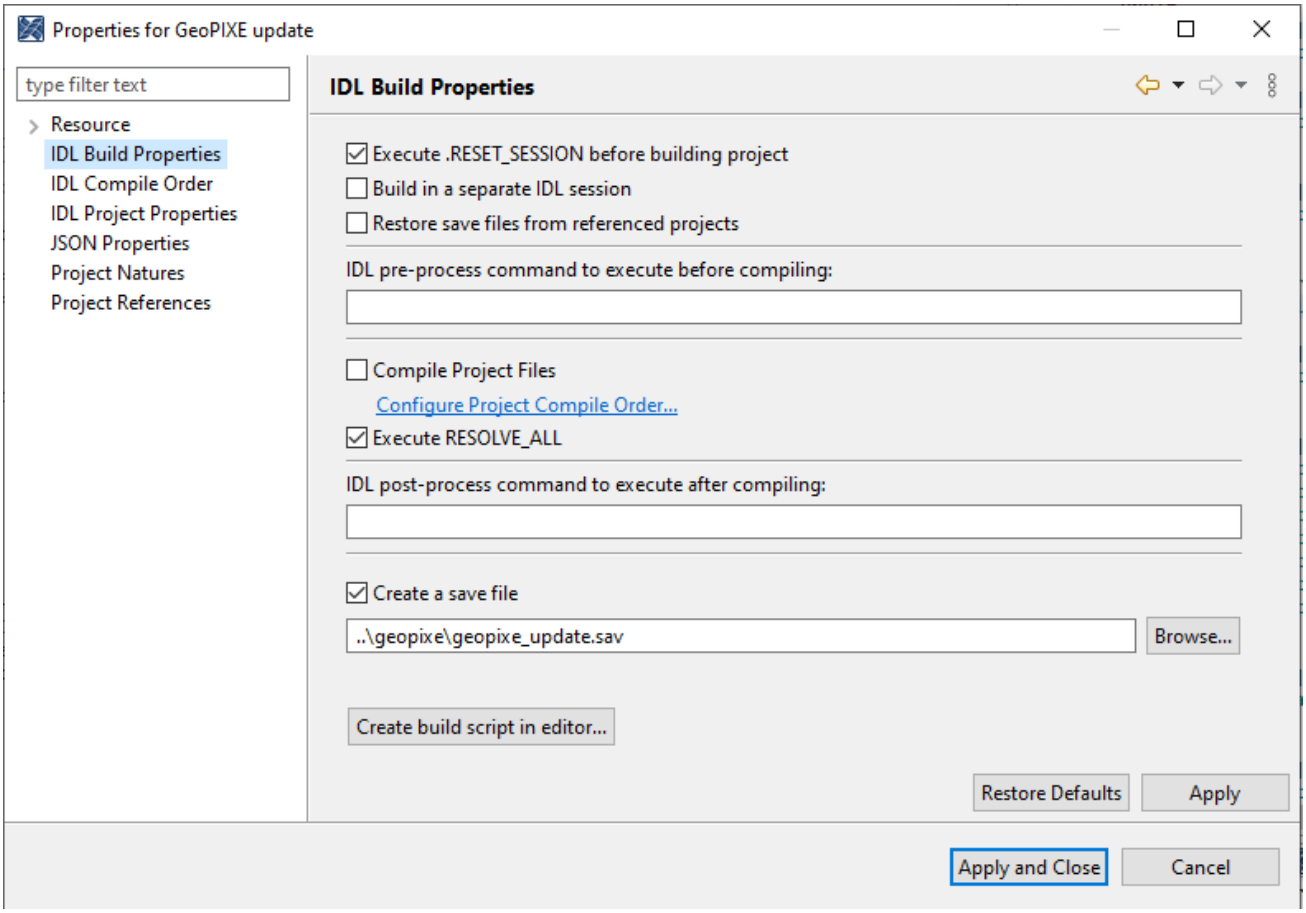
Properties for building the Depth Wizard



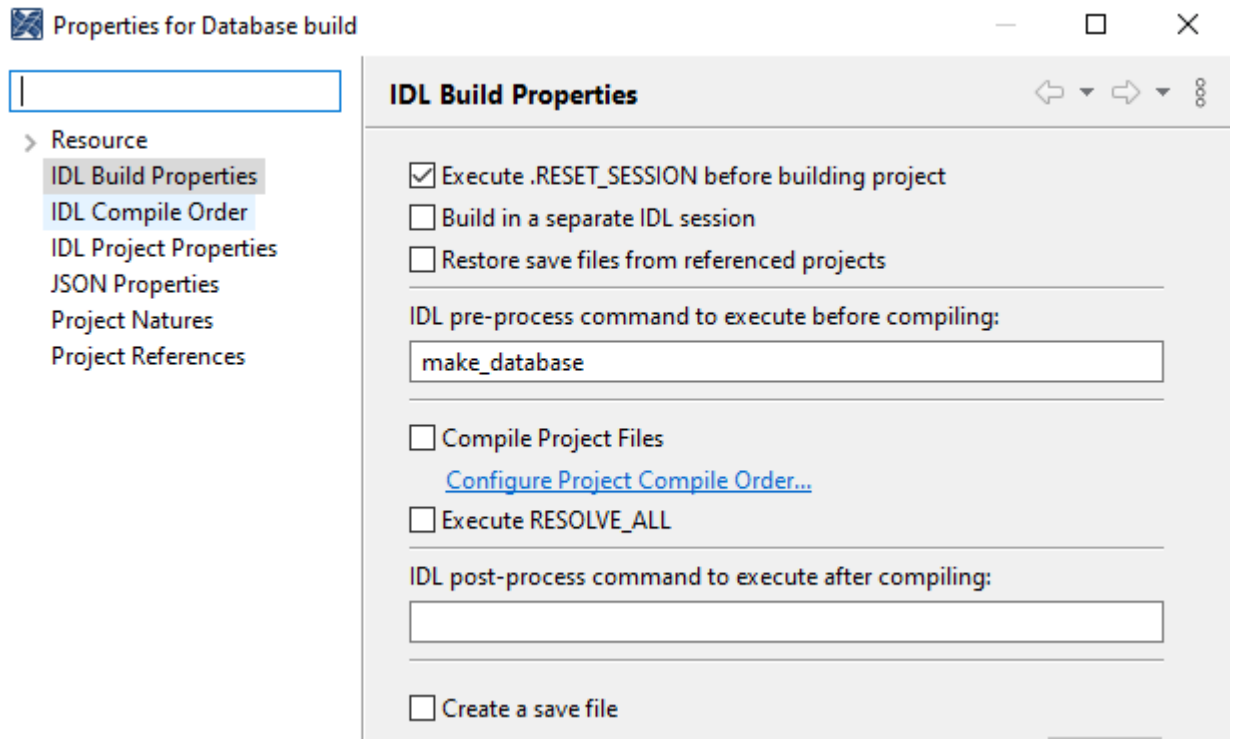
Properties for building the background processing Parallel worker



Properties for building the environment checker idl_query



Properties for building the GeoPIXE update tool (no non GUI version)



Properties for building the geopixe2.sav database file (no routines, just X-ray and other databases)

Building GeoPIXE

GeoPIXE and its modules can be built using IDLDE menus or using the "Builder" PRO script. The former requires the "properties" of every project to be setup, as described above. The *Builder* approach avoids this. But does require a two-stage approach: (i) run Builder.pro to construct a script ("build.spro") for building the selected (or all) components, and (ii) run "build.spro" at the IDL command line (after "cd" to the correct base path). This is described below.

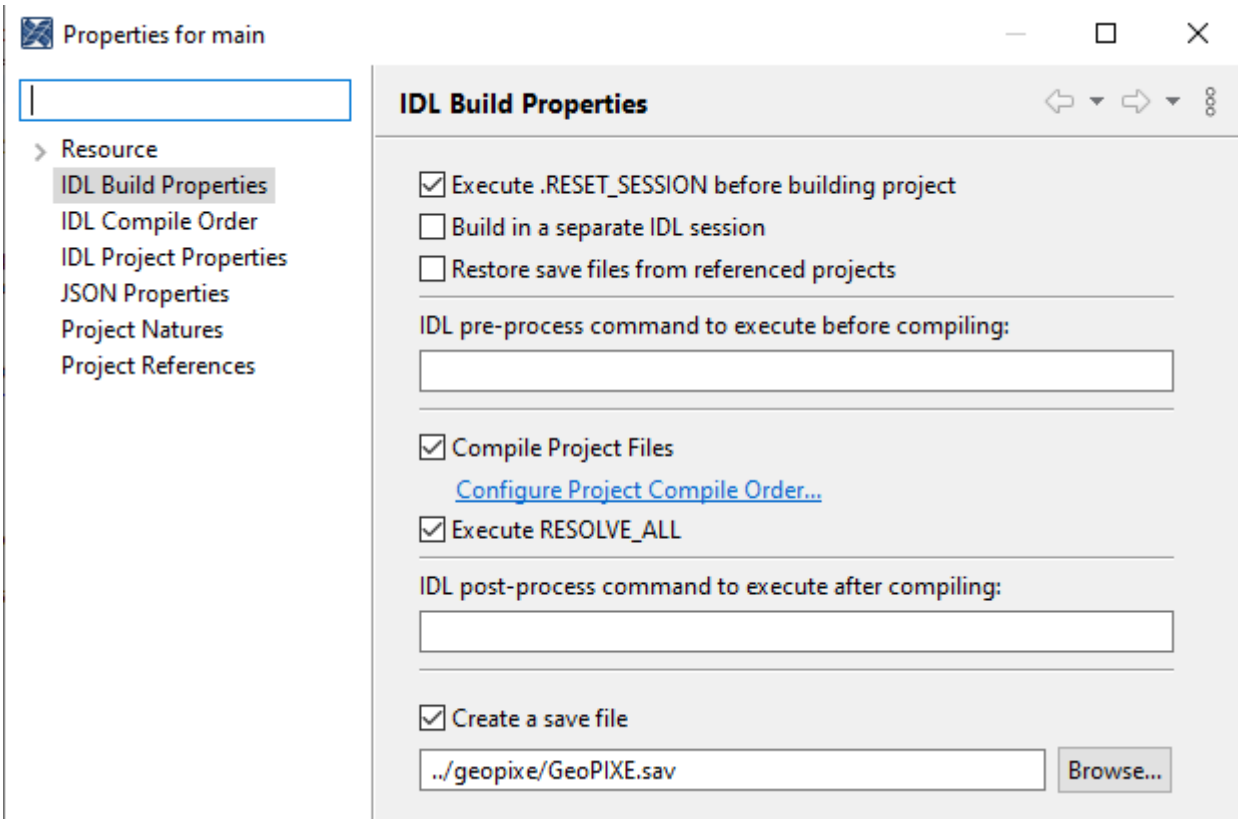
NOTE: The file "build.spro" will end up in the project "main" directory, if builder is run within IDLDE, but may end up in the "geopixe" directory if run from the builder SAV file at runtime.

Compilation of GeoPIXE main using Build menu

For the Build menu approach, the properties described above for the main GeoPIXE project "main" outline how GeoPIXE is compiled. In detail, this involves:

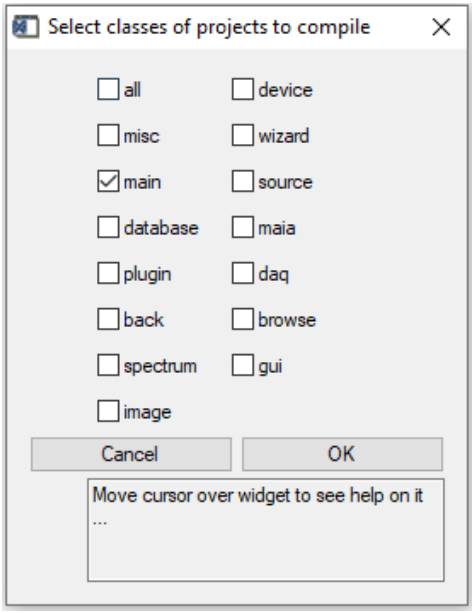
1. Reset of the session, so no compiled routines are in memory.
2. Compiles all routines in the GeoPIXE directory.
3. Executes "resolve_all" to resolve all references anywhere on the IDL search Path, which is across all open projects and IDL libraries. This is used for the stand-alone programs, like GeoPIXE, not for plugins and device objects.
4. Writes the compiled "GeoPIXE.sav" into the runtime "geopixe" directory.

This can be done in IDLDE by selecting the menu item "Build" for the project "main", if the properties are setup correctly (see above). The result of the GeoPIXE compilation is the "GeoPIXE.sav" file, which contains all referenced procedures and functions, but not those loaded at run-time (e.g. devices, plugins and wizards, which are not referenced yet, as well as "geopixe_parallel.sav" – see below).



Compilation of GeoPIXE main using the Builder PRO

In the absence of properties being setup, the simplest way to perform a full or partial build is to use the "builder" PRO (or its SAV file in runtime). This is the assumed approach for source downloaded from GitHub. Builder by default pops up a list of groups of projects to build. Select "all" to build everything. To build just the "main" dir tree of GeoPIXE program PRO and library routines, select "main" only. The result of running Builder is a script "build.spro", which needs to be executed at the **IDL command line** ("@build.spro"). Use "cd" to select the correct path first, to where build.spro is located. A pop-up will provide hints to this, including the paths.



Compilation of modules

Each project module for GeoPIXE can be built either using IDLDE and the "build" menu for the project or by using the "builder" PRO.

The result of the GeoPIXE "main" compilation (see above) is the GeoPIXE.sav file, which does not include those loaded at run-time (e.g. devices, plugins and wizards, "geopixe_parallel.sav"). These need to be compiled separately, by building the corresponding IDL project. Similarly, each device object project, wizard or image/spectrum plugin must be built individually and not include main GeoPIXE routines.

The build properties for plugins, wizards, devices and the Maia and DAQ modules do not do a resolve_all, and place the compiled SAV files in the "geopixe" runtime directory tree.

In the absence of properties being setup for each class of project, the simplest way to perform a full or partial build is to use the "builder" PRO (or its SAV file in runtime). Builder by default pops up a list of classes of project to build. Select "all" to build everything. To build just one class of project module (e.g. plugins) select the class from the pop-up list. The result of running Builder is a script "build.spro", which needs to be executed at the **IDL command line**. Use "cd" to elect the correct path first, where build.spro is located.

Compilation of Fortran libraries

A "Makefile" uses GNU Fortran (set to F90 mode) to compile the Fortran code ("image_lib.f") to libraries for Windows (32, 64bit) and Linux (32, 64bit). To compile image_lib.f use "make -f Makefile". The same Makefile supports compilation of the libraries under Darwin (Mac).

Makefile for Fortran compile

```
OS := $(shell uname -s)
NF := $(findstring n,$(MAKEFLAGS))
trace = $(if $(NF),,@echo "$(1)")

INSTDIR = /opt/geopixe/latest

#GFORTRAN=g77
#FFLAGS=-fno-pedantic -Wno-globals -ff90 -ffree-form

# fortran flags common to all platforms ...
FFLAGS = -O3 -cpp -ffree-form -fno-range-check

#FFLAGS = -O0 -cpp -ffree-form -fno-range-check

# Linux
LR = /opt/gcc-4.5.3
LT = x86_64-unknown-linux-gnu
LGF = $(LR)/bin/$(LT)-gfortran
LGF32 = $(LGF) -D__32BIT__ -m32
LGF64 = $(LGF) -D__64BIT__ -m64
LGFFLAGS = -fPIC -fsecond-underscore $(FFLAGS)
#LLD32 = $(LGF) -m32
#LLD64 = $(LGF) -m64
#LLD32 = ld --oformat elf32-i386
#LLD64 = ld --oformat elf64-x86-64
LLD32 = ld -m elf_i386
LLD64 = ld -m elf_x86_64
LLDFLAGS = -shared
LLDLIBS32 =
LLDLIBS64 =

# Windows
WR = /opt/mingw-w64
WT = x86_64-w64-mingw32
WGF = $(WR)/bin/$(WT)-gfortran
WGF32 = $(WGF) -D__32BIT__ -m32
WGF64 = $(WGF) -D__64BIT__ -m64
WGFFLAGS = -mdll $(FFLAGS)
WLD32 = $(WGF) -m32
WLD64 = $(WGF) -m64
WLDFLAGS = -shared $(FFLAGS)

# Darwin
DGF = gfortran-mp-4.5
DGFFLAGS = -fPIC $(FFLAGS)
DLD32 = gcc -m32
DLD64 = gcc -m64
DLDFLAGS = -bundle -flat_namespace -undefined suppress $(FFLAGS)

# these source files ...
SRCS = image_lib.f # ...
# produce this library (without suffix) ...
LIB = image_lib
```

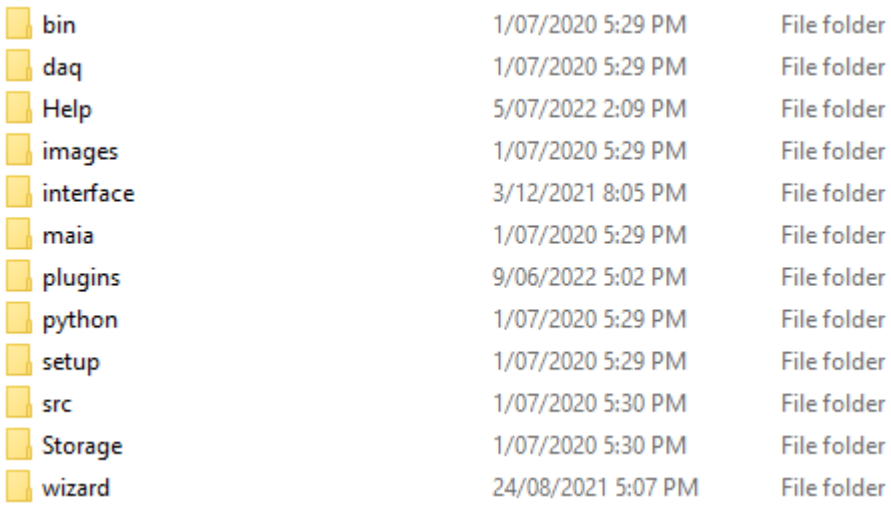
```
# there are three platforms (Linux, Windows and Darwin)
# and two word-sizes (32 and 64 bit)
word-sizes = 32 64
ifeq ($(OS),Darwin)
int-suffs = do
out-suffs = dylib
else
int-suffs = lo wo
out-suffs = so dll
endif

OBJS = $(foreach s,$(int-suffs),$(foreach w,$(word-
sizes),$(SRCS:.f=.$wbit.$s)))
LIBS = $(foreach s,$(out-suffs),$(foreach w,$(word-sizes),$(LIB).$wbit.$s))
ifeq ($(OS),Darwin)
LIBS += $(foreach s,$(out-suffs),$(LIB).$s)
endif

all: $(LIBS)

install: all
	$(call trace,Installing into $(INSTDIR) ...)
	@cp $(LIBS) $(INSTDIR)

clean:
	rm -f $(LIBS) $(OBJS)

%.32bit.lo: %.f
	$(call trace,[Linux 32bit] Compiling $^ ...)
	@$(LGF32) $(LGFFLAGS) -c -o $@ $^

%.64bit.lo: %.f
	$(call trace,[Linux 64bit] Compiling $^ ...)
	@$(LGF64) $(LGFFLAGS) -c -o $@ $^

%.32bit.wo: %.f
	$(call trace,[Windows 32bit] Compiling $^ ...)
	@$(WGF32) $(WGFFLAGS) -c -o $@ $^

%.64bit.wo: %.f
	$(call trace,[Windows 64bit] Compiling $^ ...)
	@$(WGF64) $(WGFFLAGS) -c -o $@ $^

%.32bit.do: %.f
	$(call trace,[Darwin 32bit] Compiling $^ ...)
	@$(DGF32) $(DGFFLAGS) -c -o $@ $^

%.64bit.do: %.f
	$(call trace,[Darwin 64bit] Compiling $^ ...)
	@$(DGF64) $(DGFFLAGS) -c -o $@ $^

$(LIB).32bit.so: $(SRCS:.f=.32bit.lo)
	$(call trace,[Linux 32bit] Linking $@ ...)
	@$(LLD32) $(LLDFLAGS) -o $@ $^ $(LLDLIBS32)
```

```
$(LIB).64bit.so: $(SRCS:.f=.64bit.lo)
	$(call trace,[Linux 64bit] Linking $@ ...)
	@$(LLD64) $(LLDFLAGS) -o $@ $^ $(LLDLIBS64)

$(LIB).32bit.dll: $(SRCS:.f=.32bit.wo)
	$(call trace,[Windows 32bit] Linking $@ ...)
	@$(WLD32) $(WLDFLAGS) -o $@ $^ $(WLDLIBS32)

$(LIB).64bit.dll: $(SRCS:.f=.64bit.wo)
	$(call trace,[Windows 64bit] Linking $@ ...)
	@$(WLD64) $(WLDFLAGS) -o $@ $^ $(WLDLIBS64)

$(LIB).32bit.dylib: $(SRCS:.f=.32bit.do)
	$(call trace,[Darwin 32bit] Linking $@ ...)
	@$(DLD32) $(DLDFLAGS) -o $@ $^ $(DLDLIBS32)

$(LIB).64bit.dylib: $(SRCS:.f=.64bit.do)
	$(call trace,[Darwin 64bit] Linking $@ ...)
	@$(DLD64) $(DLDFLAGS) -o $@ $^ $(DLDLIBS64)

$(LIB).dylib: $(foreach w,$(word-sizes),$(LIB).$wbit.dylib)
	$(call trace,[Darwin both] Linking $@ ...)
	@lipo -create -output $@ $^
```

Run-time directory

The run-time (distribution) directory “geopixe” organization has these sub-directories.

1. “**bin**” - contains scripts for Linux and Mac to run each *GeoPIXE*, *Maia Control*, *Scan List*, DAQ programs, as well as utilities such as blog_browse, idl_query, geopixe_update, maia_update, falconx_browse and geopixe_index.
2. “**interface**” – SAV files for Device Object
3. “**plugin**” – SAV files for spectrum, background and image processing plugins
4. “**wizard**” – SAV files for Wizards
5. “**maia**” - contain SAV files for Maia background processes as well as data (e.g. layout maps for various generations of Maia, Hymod data-flow map) and default/template *Maia Control* config file “template.Maia.conf”
6. “**daq**” - contain SAV files for DAQ background processes as well as data (e.g. layout maps, Hymod data-flow map) and default/ template *DAQ Control* config file “template.DAQ.conf”
7. “**image**” – button images
8. “**Help**” - contains help data and manuals for GeoPIXE and Maia.
9. “**src**” – sources for device objects and plugins (a convenience, so even a runtime distribution shows the source for the plugins and device objects often used as a start to make new ones).
10. “**setup**” – some example setup files for GeoPIXE
11. “**storage**” – a place to move setup files not in use (e.g. old detectors and filters)



NOTE: The name "geopixe" of the runtime directory is assumed throughout. Please do not rename this.

The runtime directory contains these classes of files (by file type):

1. dll – Windows libraries
2. so – Linux libraries
3. dylib – MAC libraries
4. detector – detector specification/parameter files (better to use "config/detector")
5. filter – filter specification/parameter files (better to use "config/filter")
6. source – source specification/parameter files
7. csv – detector array layout tables
8. sav – IDL SAV files

GeoPIXE Quantification

Theory: The Dynamic Analysis method

In GeoPIXE, the construction of the DA transform matrix builds on the results of a full non-linear least-squares fit of a model function ***F*** to a representative spectrum ***S*** that samples the elements of interest and under the same experimental conditions [Ryan 2000]. The representative spectrum is often the total spectrum from the first sample to be imaged. Once the iterative fitting procedure converges, a final linear iteration is used, with the non-linear terms fixed, to provide the terms needed to construct the DA transform matrix.

Also see added detail in the Maia detector section.

The fitting function comprises line-shape functions (Λ_{jk}) for each X-ray line *j* in each element *k*'s signature, together with pile-up (treated as an additional "sum" element) and a background term (B_i). The fitting model and background algorithms are described elsewhere, initially for PIXE [Ryan *et al*., 1990a] and [Ryan *et al*., 1988] and extended for SXRF. The function F_i takes the form

$$F_i = a_0 B_i + \sum_k a_k \sum_j \rho_{jk} \Lambda_{jk}\left(E_i, E_{jk}; p_1, p_2, \ldots\right) \qquad (1)$$

where ρ_{jk} are branching ratios for the individual X-ray lines. The line-shape functions (Λ_{jk}), centred on the line energies E_{jk}, are based on Gaussians with exponential tails and other terms containing non-linear parameters p_n, such as the parameterization of peak widths and the linear energy calibration of the spectrum.

X-ray relative intensities reflect sub-shell ionization cross-sections, Coster-Kronig transition rates between L sub-shells and fluorescence yields. PIXE yields are integrated, including secondary fluorescence effects using the approach of Reuter *et al.* (1975). In the GeoPIXE implementation for XRF, the sub-shell absorption cross-sections are based on the parameterizations of Ebel [Ebel *et al*., 2003], and the Coster-Kronig rates, fluorescence yields and branching ratios use the fundamental parameter database of Elam. [Elam *et al*., 2002]. Missing branching ratios for minor transitions (e.g. Au L transitions) are taken from the GeoPIXE database, which is based on PIXE measurements of pure elements and simple compounds for 52 elements [Ryan *et al*., 1990a], or from direct measurements.

The non-linear parameters (p_n) are determined in a full non-linear least-squares calibration fit to the representative spectrum acquired under the same experimental conditions as the image data to be analyzed. This fixes the shape of the line-shape functions Λ_{jk}. The remaining linear parameters a_k, representing major-line peak-areas and one representing background intensity (i.e. a scaling factor a_0 applied to a background model vector ***B***), can be found using a linear least squares approach (one extra linear fit iteration after the non-linear fit). The condition of χ^2 minimum in a linear least-squares fit to the spectrum, viz. $\partial\chi^2/\partial a_k = 0$ for all a_k, leads to values of the a_k at the minimum satisfying the equations (Bevington, 1969)

$$\sum_k \sum_i w_i \left(\frac{\partial F_i}{\partial a_j} \right) \left(\frac{\partial F_i}{\partial a_k} \right) a_k = \sum_i w_i \left(\frac{\partial F_i}{\partial a_j} \right) S_i \qquad (2)$$

These simultaneous linear equations can be expressed as the matrix equation (Ryan 2000)

$$\boldsymbol{\alpha a} = \boldsymbol{\beta S} \qquad (3)$$

where $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$ are matrices in terms of the partial derivatives $\partial F_i/\partial a_k$ and ***S*** is the spectrum vector (Ryan 2000):

$$\alpha_{jk} = \sum_i w_i^{-1} \beta_{ji} \beta_{ki} \qquad (4)$$

$$\beta_{ji} = w_i \left(\frac{\partial F_i}{\partial a_j} \right) \qquad (5)$$

The statistical weights w_i of Awaya ($w_i = F_i^{-1}$) have been adopted as they are based on Poisson counting statistics, which typify trace element signals and yield more accurate peak areas at low counting statistics [Awaya, 1978]. As discussed in detail elsewhere (Ryan and Jamieson, 1993; Clayton and Ryan, 1990), the convergence of the least-squares fit is only weakly affected by the form of the weights, provided the approximation $w_i = S_i^{-1}$ is <u>not</u> used as it introduces errors at low counting statistics of the order of χ_ν^2 per channel [Awaya 78, Clayton and Ryan, 1990].

The peak-areas a_k (excluding the background term) are related to element concentration C_k by the equation

$$a_k = Q\Omega\varepsilon_k T_k Y_k C_k \quad (6)$$

in terms of the integrated beam flux Q, detector solid-angle Ω and efficiency ε_k, X-ray absorber attenuation T_k and generic X-ray yield Y_k (counts per ppm.flux for an ideal detector); for the moment Y_k will be assumed to be constant for each element k across the entire image area. Hence, the solution of the linear least-squares problem can be cast as a matrix equation, which transforms directly from spectrum vector $\boldsymbol{S}$ to concentration vector $\boldsymbol{C}$ in terms of the matrix $\boldsymbol{\Gamma}$ (see imaging section for the extension to a detector array) (Ryan 2000):

$$\boldsymbol{C} = Q^{-1}\,\boldsymbol{\Gamma}\boldsymbol{S} \quad (7)$$

$$\Gamma_{ki} = \left(\Omega\varepsilon_k T_k Y_k\right)^{-1} . \sum_j \left[\alpha^{-1}\right]_{kj} \beta_{ji} \quad (8)$$

Ignoring the scalar Q^{-1} for now, each detected event in channel e selects the column Γ_{ke} from the right side of this equation. If the beam is positioned within pixel x,y in a raster scan, each event (e,x,y) makes a contribution $\delta C_k(x,y) = Q(x,y)^{-1}\,\Gamma_{ke}$ to the concentration distribution of element k. This suggests a convenient strategy to accumulate elemental images M_k in concentration-flux units by simply incrementing each image k (at x,y) by Γ_{ke} for each event at energy e and a variance image can be accumulated by incrementing by Γ_{ke}^2 (Ryan 2000):

$$\delta M_k(x,y) = \Gamma_{ke} \quad (9)$$

$$\delta V_k(x,y) = \Gamma_{ke}^2 . \quad (10)$$

These images can be directly related to concentration, both on-line during data collection or off-line, by reference to the live-flux distribution $Q(x,y)$. In other words, the average concentrations $\langle C_k \rangle$ in a region of the image for elements k and their uncertainties $\langle \delta C_k \rangle$ are given by (Ryan 2000):

$$\left\langle C_k \right\rangle = \frac{\left(\sum_{region} M_k(x,y)\right)}{\left(\sum_{region} Q(x,y)\right)} \quad (11)$$

$$\left\langle \delta C_k \right\rangle = \frac{\left(\sum_{region} V_k(x,y)\right)^{1/2}}{\left(\sum_{region} Q(x,y)\right)} \quad (12)$$

Hence, within the approximation of constant Y_k across the image area, these images are quantitative and can be directly interrogated for elemental concentrations, even online while data are collecting. Correction for the variation of Y_k due to sample composition spatial variation is considered below as a post image projection correction.

Spectrum fitting

The fitting routine is based on the book by Bevington (1969) with some changes/additions: (i) It uses the weighting scheme of Awaya (1978) to produce better peak areas at low counting statistics without any systematic bias. (ii) It parametrizes the peak functions in terms of peak area, width and centroid. The use of area and width provide more “orthogonal” fitting terms than height and width, which increases fit stability.

(iii) It parametrizes energy calibration around a pivot typically 1/3 the way along the fitted energy range, which also improves the "orthogonality" of the two-energy calibration fitting terms. (iv) It parametrizes peak width around a pivot typically at the start of the fitted energy range, which also improves the "orthogonality" of the two width fitting terms, but also guarantees that no peak width can go negative. (v) Originally, the structure of the fitting loop was designed to utilize vector processor libraries, with all terms expressed as vectors and matrices. Since IDL supports vectors and matrices natively, this now appears as simple equations in terms of IDL vector and matrix variables (see "pixe_fit.pro").

References

T. Awaya, (1978), ***Nucl. Instr. Meth***. 165, 449.

P.R. Bevington, (1969), "***Data reduction and error analysis for the physical sciences***", McGraw-Hill, New York.

J.L. Campbell, (2003), ***Atomic Data and Nuclear Data Tables*** 85, 291–315.

E. Clayton and C.G. Ryan, (1990), ***Nucl. Instr. Meth***. B49, 161-165.

H. Ebel, R. Svagera, M.F. Ebel, A. Shaltout and J.H. Hubbell, (2003), ***X-Ray Spectrometry*** 32, 442–451.

W.T. Elam, B.D. Ravel, J.R. Sieber, (2002), ***Radiation Physics and Chemistry*** 63, 121–128.

W. Reuter, A. Luria, F. Cardone, and J.F. Ziegler, (1975) , "Quantitative analysis of complex targets by proton-induced X-rays", ***J. App. Phys.*** 46, 3194-3202.

C.G. Ryan, E. Clayton, W.L. Griffin, S.H. Sie and D.R. Cousens, (1988), ***Nucl. Instr. Meth.*** B34, 396-402.

C.G. Ryan, D.R. Cousens, S.H. Sie, W.L. Griffin, G.F. Suter and E. Clayton, (1990a), ***Nucl. Instr. Meth***. B47, 55-71.

C.G. Ryan and D.N. Jamieson, (1993), ***Nucl. Instr. Meth***. B77, 203-214.

C.G. Ryan, (2000), ***International Journal of Imaging Systems and Technology*** 11, 219-230.

GeoPIXE Operations

Also needs notes on:

1. spectrum overlays, extra for MPDA
2. Extension of DA for multiple phases - MPDA

Spectrum Fitting and Yield modelling

Element lines are fitted with the "Area" of the major line as the main variable. Other lines are expressed relative to this using the line relative intensities. The parametrization of the peak area fit is in terms of variables for peak area, centroid and FWHM. Together with a careful choice of weighting (see Ryan,1990a, and Clayton and Ryan, 1990) converges on accurate peak area, even as counting statistics decline. The challenge for quantification is to deal with matrix effects on the model X-rays yields from elements, integrated with depth in a finite thickness sample, and to also deal with variation across large detector arrays, such as Maia (see Maia detector section). This walk through the routines, hopefully, conveys these two aspects.

Model yields will be calculated for all X-rays lines for all elements in the periodic table, with lines between two energy limits (e.g. 2 to 48 keV), for each layer in a multilayer sample. For a detector array, these will be

calculated for the central detector in the array (including secondary fluorescence), and then relative sensitivity factors will be calculated to relate the other detectors to the central detector. X-ray relative intensities for the X-ray lines for each element will also be calculated across the detector array (see "geo_array_yield" below).

When a specific spectrum is to be fitted, the actual detector elements that contribute to the spectrum will be selected as a subset of the array and yields that combine these will be determined as well as X-ray relative intensities that reflect the relative sensitivities across these selected detectors, to arrive at representative yields and relative intensities for each element.

For a multilayered sample, you specify the 'unknown' layer, where you believe the remaining elements reside. It is assumed that the composition of all other layers has included any elements there, and the signal from these will be subtracted from the signal in the spectrum to arrive at what remains in the 'unknown' layer and the result will be the concentration of the elements in the unknown layer. This model implicitly assumes a simple layered structure. Extensions to handle fluid inclusions model departures from this model to cater for ellipsoidal volumes, which approximate fluid inclusion cavities.

PIXE_fit

Performs a least square fit to a spectrum using model yields, a selection of elements and an empirical background (see "pixe_fit.pro"). The method is based on the algorithms of Bevington (69) written to make use of fast vector and matrix operations in IDL, but with modified weights of Awaya (78). Yields are modelled for a layered sample structure and need to be corrected to reflect averages across the selected detectors in a detector array (done in 'array_yield').

See also the doc "**PIXE-fit equations**" PDF for details of the fitting function equations, the book by Bevington and the description above of the DA method. For details see the method paper: Ryan *et al.*, 1990a.

It starts by deriving some initial starting values for the fitting parameters 'A', which are the element peaks areas (for major line for each element), background weighting (background is estimates elsewhere, see "strip_clip" routine and the SNIP algorithm), 2 parameters for energy calibration, 2 parameters for peak width versus energy, X-ray line tailing parameters and something similar for Compton tails. This includes constructing a "sum" element for pileup based on the starting intensities of all elements (based on their 'A' parameters).

The 'A' vector is organized like this (limited by IDL vector indices, which always start at zero*):

'A' index	**Function**
0	Noise – constant width term
1	Fano – energy variation of peak widths
2	constant channel offset (see below)
3	spectrum channels/bins per energy (keV^{-1})
4	Pileup weighting
5	Tail Amp – amplitude of exponential tails
6	Tail length – length of tails
7	Background 1 – background weight (or of low energy part if "split backgrounds" are used)
10	Background 2 – weight for high energy part of background (if "split backgrounds" are used)
8	Compton tail Amp – amplitude of the Compton tail distribution
9	Compton tail length – Compton tail length
11-20	Tweek lines – when "Adjust" is used to fit groups of line strengths for a selected element
21-*	Element intensity – Area of "major line" for each element.

* The original Fortran permitted negative indices for parameters and 0 and positive for elements.

The parametrization of energy calibration takes this form. The centroid ‘c’ of a peak at energy ‘E’ is given by:

c = A[2] + A[3] * (E-eoc))

where ‘eoc’ is the energy origin or “pivot”.

The parametrization of peak width squared (w2 = $FWHM^2$) takes this form:

w2 = A[0]*A[0] + A[1]*A[1] * (E-eow))

where ‘eow’ is the peak width origin or “pivot”.

It gathers X-ray line intensity data and corrects it for the effects across a detector array in ‘array_yield’.

Calling ‘array_yield’ passes the following:

```
Y = (*peaks).yield[*,(*peaks).unknown-1]        ; generic yields in unknown layer
rY = (*peaks).ratio_yield[*,*]                  ; ratio_yields
```

where (*peaks).unknown selects the “unknown” layer (i.e. where we believe our analyzed element reside) from the full set of layer model yields to form our model yields ‘Y’. ‘array_yield’ calculates the effective yields for an array detector, given the selection of detectors for this analysis, from the full set of yields calculated for each detector in the array.

*Note: Notation (*peaks) means the structure pointed to by pointer ‘peaks’.*

Note that rY (and rGamma returned by ‘array_yield’) must be in normal detector number order (as distinct for detector table CSV file index order).

The function “pixe” calculates the contributions to the spectrum for the current fitting parameters ‘A’, including supplied background, to a vector of spectrum channels ‘x’. A ‘mask’ vector enables elements of the ‘A’ parameter vector to be varied in the fit or not. The output partial derivatives (‘dfa’), of the fitting function with respect to the fitting parameters ‘A’, are output only for the parameters masked ON. A vector ‘q’ provides the index of the varied parameters back into the ‘A’ vector. “pixe” calls “line”, which calculates the contributions of each X-ray line to the fitting function ‘f’, weighted by the relative intensities of the lines. It also calculates contributions to the partial derivatives ‘dfa’.

PIXE_fit then determines the ‘beta’ vector and ‘alpha’ matrix as per Bevington (see equations 4 and 5 above) for these masked ON parameters. The diagonal terms of ‘alpha’ are weighted by (1+grad) to achieve a transition from a gradient fit search (high ‘grad’) to a function linearization fit (‘grad’ << 1), as discussed by Bevington. The idea is to start with some value for ‘grad’ to follow Chi-squared gradients and eventually reduce ‘grad’ to use function linearization in the vicinity of the Chi-squared minimum, which converges more quickly where the Ch-squared surface becomes more paraboloidal. Depending on the initial reduced Chi-squared, the initial value of ‘grad’ is 0.1 (small reduced Ch-squared) or 0.5 for larger Chi.

It forms the ‘gamma_matrix’ from the inverse of ‘alpha’ using (“##” denotes the matrix multiply in IDL) …

gamma_matrix = inverse(alpha) ## beta

For each iteration of the fit, increments ‘da’ to the fitting parameters ‘A’ (including the peak Area) are determined from this matrix equation (“##” is matrix multiply – rows of first arg * columns of second):

da = gamma_matrix ## transpose(y-f)

where ‘y’ is the spectrum values vector, ‘f’ is the fitting function for the latest iteration and ‘gamma_matrix’ is the inverse of the covariance matrix for the least-squares fit (e.g. see Bevington), which uses the modified weights of Awaya given by ‘f’. Chi-squared is checked for fitting convergence after each iteration.

The fit may proceed in several phases, which control the fitting ‘mask’. This is done in stages enabling certain sensitive parameters (e.g. widths or tailing) later in order to keep the fit stable and converging, at the slight expense of an extra iteration or two.

At each iteration, a “sum” peak set of intensities is updated, an updated ‘grad’ is modified and Chi-squared is determined to see if it is converging. Until a convergence is obtained at a low enough ‘grad’, the fit loops for another iteration, unless a maximum iteration count is achieved (typically 20).

In the special case of the multiphase loop, the fit is repeated after the yields are modified to reflect the calculated phase fractions determined from the current fit concentrations (related to ‘A’ for the elements).

The resulting element concentrations are determined from the peak areas (‘A’ for elements) using …

Conc = area / yield

Area = major line peak area fitted
Yield = major line yield, calculated in ‘array_yield’

A clean-up routine (“pixe_cleanup”) converts the A parameters back to Cal a, b for energy calibration (E = aX + b) and peak width w0, w1 (FWHM = sqrt(w1*E + w0)) and determines error estimates, reduced Chi-squared contributions for each element (reported in old Fortran GeoPIXE, but not output at present in the IDL version), overlap errors and detection limits (99% confidence MDL) for all fitted parameters, and derived parameters (a,b, w0,w1).

Array_yield

X-ray yields are usually calculated for all X-ray lines in the range (e.g. 2-48 keV), for all elements (Z up to 93) and in the case of a detector array, for all detectors on the array (see “array_yield.pro” and below). For a particular data-set and a spectrum being fitted, some subset of detectors will be selected. ‘array_yield’ calculates the effective yields for an array detector, given the selection of detectors for this analysis, from the full set of yields calculated for each detector in the array (see below). It is called prior to fitting.

Inputs provide details of the full detector array (pdetector, playout), which detectors are active (active), elemental X-ray line info (e_line, n_lines, n_els, E), a full table of model yields for all elements, in all layers.

Inputs:

pdetector	pointer to detector struct
playout	pointer to detector array layout
pfilter	pointer to external filter
array	bool flags an array detector
active	list of active detector channels
n_det	total number of detector channels (includes inactive)
n_els	# elements
n_lines	# lines per element
e_line	line energies per element per line index
E	major lines energies per element

Y	yields per element
theta	detector centre theta angle
phi	detector centre phi angle
charge	beam charge
rY	relative yields per element and detector (not input if /refit or /use_last)
rIntensity	relative intensities relative to 'central' detector for all lines, elements
multiplicity	fall-back (some old data), replaced by sum of 'rGamma' to form 'totGamma'
pressure	indicates fitting a spectrum with ambient conditions specified, with pressure (mbar)
temp	and temperature (C). Pass these onto to 'transmit' in filters and 'det-eff'.
/refit	flags doing a refit of an existing fit results
/use_last	use previous calculation of array corrections to save time

Input (if refit=1):

rGamma	relative sensitivity factors per element and detector (ignore 'rY')

Output (if refit=0):

rGamma	relative sensitivity factors per element and detector

Output:

counts_per_ppm_uc	effective total yields per element
cIntensity	correction factor across array to line intensities per line per element (these are applied to (*peaks).intensity on return)
totGamma	sum of rGamma for used detectors (effective multiplicity factor)
error	flags an error

where ‘active’ selects which detector pads are used for this spectrum, ‘Y’ are the yields for the selected elements in the ‘unknown’ layer, ‘rY’ are the relative yields for these elements across the full detector array, and ‘rIntensity’ are the relative intensities of all lines for each element, relative to 'central' detector for all lines, for all detector pads across the array. The output 'rGamma’ are relative sensitivity factors per element and detector including yield, efficiency, filters and solid-angle.

Note that ‘rY’ (calculated in ‘geo_array_yields’) and ‘rGamma’ must be in normal detector number order.

For a single detector, or the first, reference detector in an array, calculate solid-angle and efficiency using the normal 'diameter' and 'distance' parameters. For a valid array, calculate ratios relative to this for each detector element below in the multiplicity factors, returned as ‘rGamma’.

Generic or ‘central’ detector element as reference ...

Solid angle:

```
omega = solid_angle( (*pdetector).distance, (*pdetector).diameter, tilt=(*pdetector).tilt, $
                                                    array=0, shape=(*pdetector).shape)
```

Intrinsic efficiency:

```
eff = det_eff( pdetector, e, external_filters=pfilter, gamma=(*pdetector).pige)
```

Filters (now part of ‘det_eff’):

```
T = 1.
```

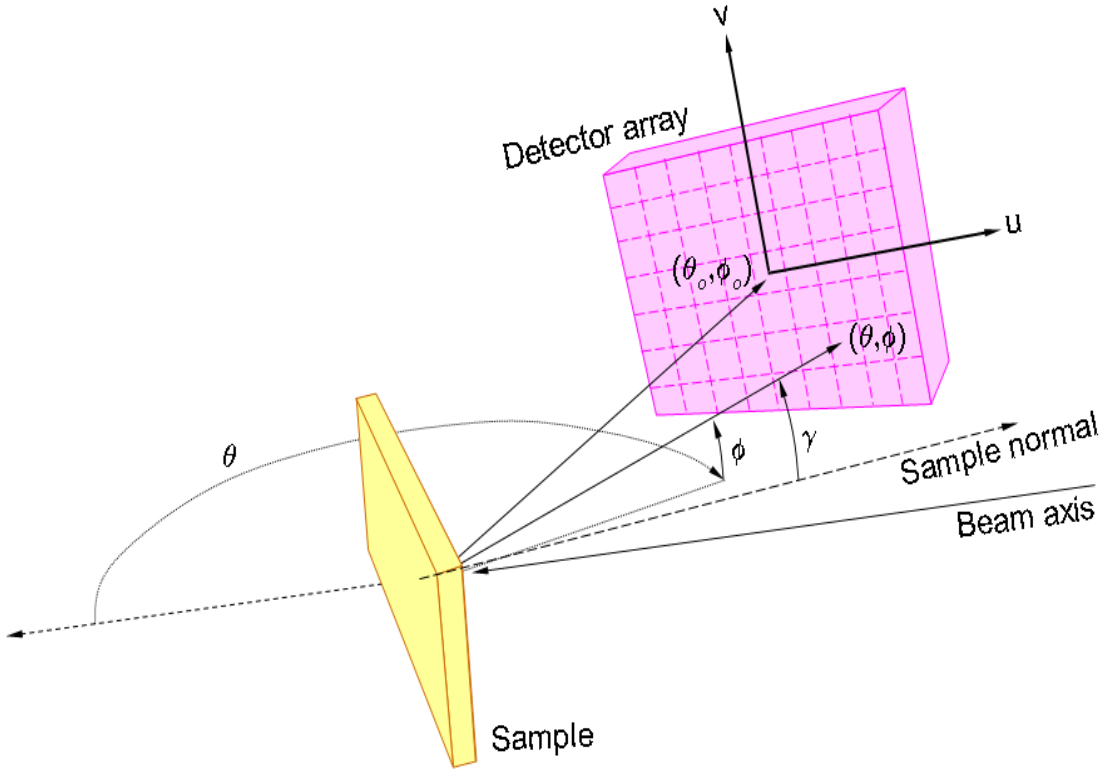
Effective yields:
counts_per_ppm_uc = float(omega * eff * T * Y)

Total yields, for given flux:
yield = counts_per_ppm_uc * charge (initially for central detector)

If /refit or /use_last, then just scale this 'yield' by total 'rGamma' for selected detectors. For /refit, 'rGamma' is input and for /use_last, 'rGamma' comes from 'wGamma' in common from previous loop in fit-all. If user_last=0 and refit=0, then need to calculate 'rGamma'.

Note that some filter transmission is now built into det_eff as 'absorbers', passed as 'pfilter' (in order to handle pin-holes properly?). For filters that DO follow the contours of the detector array, put the filter in the Detector 'absorber' specification, not here in 'filters'. The filters here are assumed to be straight across the front of a non-flat detector, and not following the contours of the detector array.

For an array, need to calculate relative efficiency of each relative to a 'central' generic detector, which is the one referred to in the *pdetector struct. In the Maia case, a square 1x1 mm detector on axis. This uses the call 'detector_geometry', which takes the detector layout, any global tilt and the central angles to the array (theta, phi) and determines the effective detector geometry for each detector. 'array_yield' calculates the ratio to the central detector for solid angle (rOmega), filter attenuation (rFilt), intrinsic efficiency (rEff), and finally combining rY, rOmega, rEff, rFilt to give the relative sensitivities 'rGamma' across the array. It also calculates correction factors 'cIntensity' across array to the line intensities per line per element (these are applied to line relative intensities (*peaks).intensity on return).



Detector relative 'u,v' coordinates across face of detector, located at global angles $\vartheta_o\varphi_o$. Take-off angle from sample normal is γ.

Definition of angles

At theta=0, phi=0 detector is down-stream in positive Z direction. Theta rotates detector away from Z axis (positive theta is to left, opposite of this diagram). Phi rotates detector, azimuthally about the Z axis (above diagram not clear). See “detector_geometry.pro”.

Initially ‘rGamma’ is calculated in CSV index order (order in the CSV file table of detectors within an array), the same as rOmega, rFilt and rEff. rY is not in this order. It is in detector number order, and is accessed in CSV order using: m = (*playout).data[i].index - (*playout).start

rGamma[i,*] = rY[m,*] * rOmega[i] * rEff[i,*] * rFilt[i,*]

rEff[i,*] = det_eff(det, e) / Eff0	use R and tilt for CSV ‘i’
rOmega = 1000. * (darea*cos(g.tilt/!radeg)/(g.R*g.R)) / omega	relative solid-angles
rFilt[i,*] = transmit(pfilter, e, tilt=g2[i].tilt) / Filt0	relative filter transmission
rY[j,*] = relative yields, considering take-off angles	relative yields for det ‘j’

Eff0 = ‘central’ or generic detector efficiency
omega = generic or ‘central’ detector solid-angle
darea = area of each detector element
Filt0 = normal (central) transmission through filters
g = struct array; each has effective angles *ϑ, φ*, *R* and *tilt* of each detector calculated using ‘detector_geometry’. ‘g’ is in CSV order, which is why rOmega, rFilt and rEff are too.

rGamma is finally re-ordered back to detector number order like this ...

rGamma[*,i] = rGamma[(*playout).ref, i]

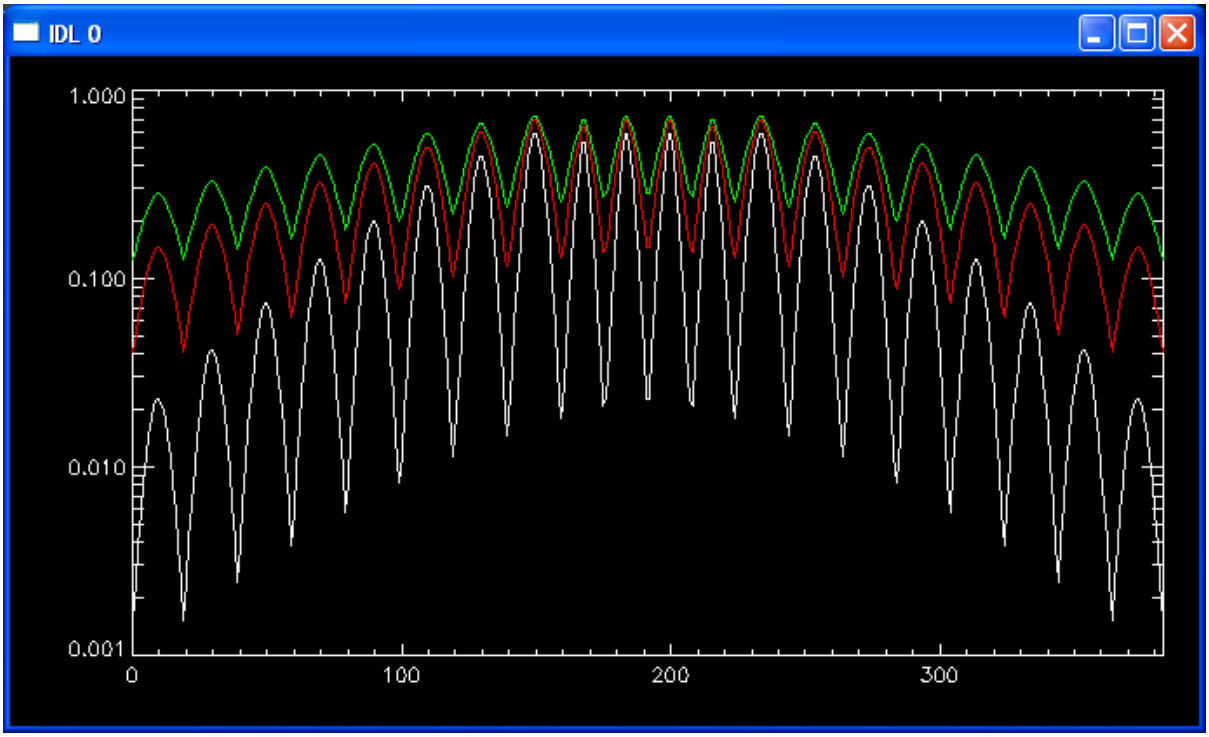
Finally, yield of central detector (counts_per_ppm_uc * charge) is modified by the sum of the rGamma:

totGamma = totGamma + rGamma[i,*]	only for active channels
yield = yield * totGamma	

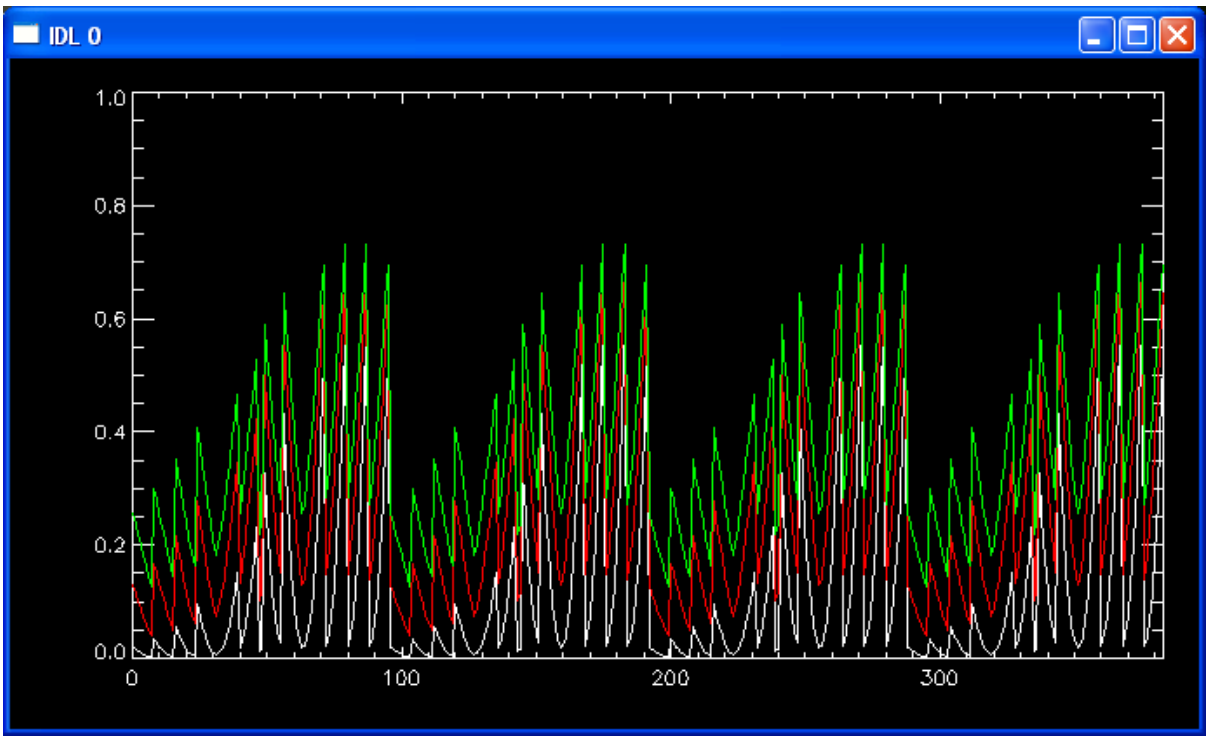
In fitting, effectively, we use ‘yield = counts_per_ppm_uc * charge * totGamma’.

Relative Intensities
Also calculate corrections ‘cIntensity[k,i]’ to the relative intensities, depending on which detectors are active, to be used in the fit. This combines the following terms:

rIntensity	changes to rel-ints calculated in the geo_array_yield model across array
rFilt_all	corrections to rel-ints for filter absorption
rEff_all	corrections to rel-ints for detector efficiency



rGamma plots (CSV order prior to re-ordering, across rows for Maia) for Cr (white), Zn (red) and scattering (green) in "array_yield", for Ni foil using 154 µm Al filter and Maia 384 detector.



After re-ordering to detector number order.

Detector_geometry

Used in 'array_yield' to calculate the effective angles *ϑ, φ*, *R* and *tilt* of each detector element.

Calc_DA_matrix

If the generation of a DA matrix is flagged, after the fit a single further linear iteration will be performed to calculate the DA matrix, with data passed to 'calc_da_matrix'. Calculate DA matrix, for the 'central' detector, where 'counts_per_ppm_uc' is defined for it.

matrix[x,i] = gamma_matrix[*,i] / counts_per_ppm_uc

Optionally, test matrix against parent spectrum:

```
t = reform( matrix ## (*(*pspec).data)[0:size-1] )
t = t / (rG * charge)
```

rG = sum of rGamma for active detectors (same as 'totGamma' above)
charge = charge for spectrum

Similarly, an image (ppm.uC units) is built by accumulating 'matrix' column values for each event (see "da_evt" routine). Then the final image is normalized by 1/rG, where rG is the sum of rGamma above.

Find element names using: print, name[q]
Plot the following: gamma_matrix[*,2] * y

DA Matrix struct:

```
** Structure <3208a00>, 21 tags, length=652320, data length=652316, refs=1:
   LABEL           STRING    'Godel_6618-June-2013/blog/62337/6'...
   FILE            STRING    ''
   CAL_ORIG        STRUCT    -> <Anonymous> Array[1]
   CAL             STRUCT    -> <Anonymous> Array[1]
   STATION         INT             50
   CHARGE          FLOAT      3.64294e-006
   N_EL            LONG               19
   EL              STRING    Array[19]
   MU_ZERO         FLOAT     Array[19]
   DENSITY0        FLOAT           3.20000
   ECOMPRESS       LONG                 1
   MDL             FLOAT     Array[19]
   SIZE            LONG             4096
   MATRIX          FLOAT     Array[4096, 19]
   YIELD           FLOAT     Array[19]
   THICK           FLOAT           2.14400
   ARRAY           STRUCT    -> <Anonymous> Array[1]
   N_PURE          LONG               19
   PURE            FLOAT     Array[4096, 19]
   E_BEAM          FLOAT           18.5000
   PMORE           POINTER   <NullPointer>
```

where 'yield' is 'counts_per_ppm_uC' for the central detector, and 'array' contains the 'rGamma' factors. There is no relative intensity data stored here.

Geo_array_yield

Before fitting is done, X-ray yields are modelled in 'geo_array_yield' for all X-rays lines for all elements in the periodic table, with lines between two energy limits (e.g. 2 to 48 keV), for each layer in a multilayer sample.

The hierarchy of routines for yield calculation follows:

geo_array_yield	Accumulate yields across detector array.

geo_yield2	Initially for the central detector, including secondary fluorescence.
geo_yield2	In a loop over all detectors in an array, without sec. fluorescence (sec. fluor. enhancement provided by central detector model).
experiment_angles	Determine direction cosines for beam and detectors.
calc_slices	Construct incremental slices of sample from input layer specs.
slow_beam	Model slowing (or attenuation) of input beam.
calc_abs	Build array of mass-absorption coefficients for all lines.
calc_cross	Calculate cross-sections as beam slows for all elements.
calc_yield	Integrate X-ray yields over all slices.

In ‘geo_array_yield’, the notation below uses ‘l’ for layer index, ‘k’ for line index, ‘i’ for element index and ‘d’ for detector index.

Input:

formula[l]	array for chemical formulae for layers 'l'
thick[l]	thickness in mg/cm^2, or optionally in microns
weight[l]	formula multipliers in weight-% (default to atomic fraction)
microns[l]	single switch, or array (one for each layer), selecting microns thickness (default is mg/cm^2)
density[l]	density of each layer (if microns is selected)
energy	energy of ion beam (MeV), or photon beam (keV), or use 'beam'
beam	general continuum beam struct, will replace 'energy'
z1	Z of beam particles
A1	A of beam
state	charge state of beam
theta	angle of X-ray detector to the beam in horizontal plane
phi	azimuthal angle of detector
alpha	rotation of target about Y axis
beta	tile of the target about the target centre line
unknown	number of the "UNKNOWN" layer, to weight relative intensities.
sec_fl=0	disables secondary fluorescence calculation.
select	specify a list of Z to calculate yields for (default 2-92).
/gamma	for PIGE yield calculation
array	array=1 denotes a detector array, for which detector and layout should be supplied
detector	pointer to detector struct
layout	pointer to layout struct

Output:

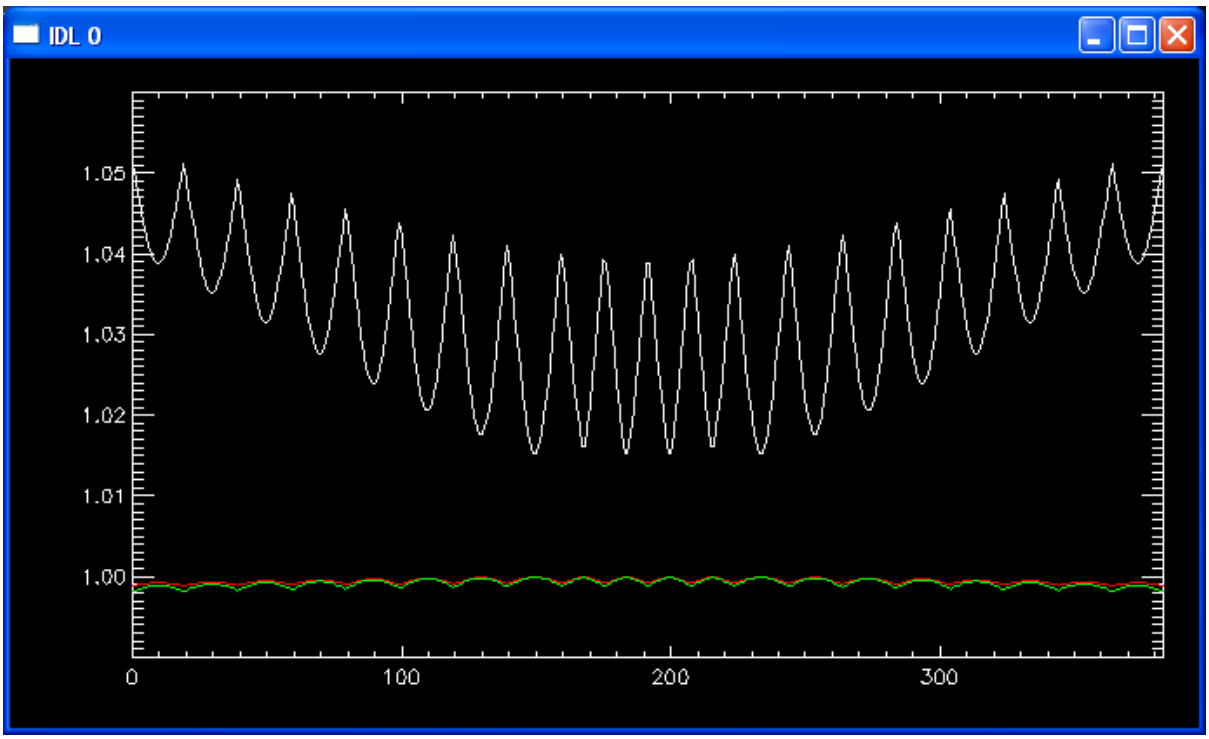
intensity[k,i]	Relative intensities for X-ray line 'k', element 'i', 'central detector, averaged over all layers, weighted by # atoms in each (LATER).
ratio_intensity[d,k,i]	Relative intensities for X-ray line 'k', element 'i', detector 'd'
lines[k,i]	Line index for each line 'k', element 'i'
e_lines[k,i]	Energy of lines
n_lines[i]	Number of lines for each element 'i'
z2[i]	Atomic number for element 'i'
shell[i]	Shell used for element 'i' (note K, L, M done as separate "elements")
layers[l]	array of layer structs for sample layers
sec_yield[i,l]	Yield component due to secondary fluorescence only.
mu_zero[k,i]	mass absorption coefficients for slice 0.
ratio_yield[d,k]	ratio of yield for detector 'd' to generic yield, element 'k' must be in normal detector number order.

Return:

yield[i,l]	Yield total for element 'i', layer 'l' Generally in units of counts/ppm/uC, unless continuum beam, then counts/ppm/s

'geo_array_yield' determines cosine factors for incoming beam and detected outgoing X-rays using 'experiment_angles' and calls 'geo_yield2' to model yields for the central detector, including secondary fluorescence. Then it calls 'geo_yield2' in a loop over all other detectors in a detector array, not including secondary fluorescence. It simply adopts the secondary fluorescence enhancement factor from the central detector result (as this is a sample property, not the detector array). It uses 'detector_geometry' to provide the effective geometry (theta, phi) of every detector in the array, given the global angles and tilt of the array as a whole. It forms the array 'rIntensity' as the ratio of the intensities across the array relative to the central detector.

'Geo_yield2' returns rel_int[k,i] for element 'i', line 'k' for 'central' detector. Then for a detector array, 'Geo_yield2' is called again to return rel_inti[k,i] for each detector. Need to form weighted average across the array using rY[d,i] for element 'i', detector 'd' to weight rel_inti[*,i] and sum over all 'd'. But, like yields, this will need to be done just prior to the fit using the *current detector selections* (done in 'array_yields'). Here we need to just pass back the individual rel_int's for each detector to be saved in the yields file.



Cr Kb (white), Zn Kb (red) and Br (Kb (green) rIntensity across Maia array in 30 µm pyrite yield calculation (CSV order), which is Kb relative to 'central' detector rel-int.

Geo_array2

This builds the X-ray yields for the beam (particle, X-ray) hitting a layered sample structure, and optionally builds the X-ray relative intensities. The notation below uses 'l' for layer index, 'k' for line index, 'i' for element index and 'd' for detector index.

Input:

formula[l]	array for chemical formulae for layers 'l'
thick[l]	thickness in mg/cm^2, or optionally in microns
weight[l]	formula multipliers in weight-% (default to atomic fraction)

microns[l]	single switch, or array (one for each layer), selecting microns thickness (default is mg/cm^2)
density[l]	density of each layer (if microns is selected)
energy	energy of ion beam (MeV), or photon beam (keV), or use 'beam'
beam	general continuum beam struct, will replace 'energy'
z1	Z of beam particles
A1	A of beam
state	charge state of beam
theta	angle of X-ray detector to the beam in horizontal plane
phi	azimuthal angle of detector
alpha	rotation of target about Y axis
beta	tile of the target about the target centre line
unknown	number of the "UNKNOWN" layer, to weight relative intensities.
sec_fl=0	disables secondary fluorescence calculation.
select	specify a list of Z to calculate yields for (default 2-92).
/gamma	for PIGE yield calculation
/force_trans	force the inclusion of slices for a transmission target
/reenter	2nd pass into Geo_yield2 for yield only of more detector elements

Carry over for '/reenter' only. They are used to bring in results (e.g. 'slices' and 'fluoro') from initial 'geo_yield2' (called from 'geo_array_yield') using central detector and including secondary fluorescence.

slices	slices structures
lid	layer id for a slice index
relmux	relative mux
branch_ratio	branching ratios
e_bind	binding energies
fluoro	fluorescence yields
jump	jump ratios
spec	beam struct as a function of half-slice

Output:

intensity[k,i]	Relative intensities for X-ray line 'k', element 'i' averaged over all layers, weighted by # atoms in each (LATER).
lines[k,i]	Line index for each line 'k', element 'i'
e_lines[k,i]	Energy of lines
n_lines[i]	Number of lines for each element 'i'
z2[i]	Atomic number for element 'i'
shell[i]	Shell used for element 'i' (note K, L, M done as separate "elements")
layers[l]	array of layer structs for sample layers
sec_yield[i,l]	PIXE yield component due to secondary fluorescence only.
mu_zero[k,i]	mass absorption coefficients for slice 0.

Return:

yield[i,l]	PIXE/SRXF yield total for element 'i', layer 'l' Generally in units of counts/ppm/uC, unless continuum beam, then counts/ppm/s

Yields struct:

```
** Structure <2d09010>, 33 tags, length=176432, data length=176428, refs=4:
   N_ELS           LONG                 96
   Z               INT       Array[96]
   SHELL           INT       Array[96]
   N_LINES         INT       Array[96]
```

```
LINES           INT       Array[20, 96]
E               FLOAT     Array[20, 96]
INTENSITY       FLOAT     Array[20, 96]            ; 'central' detector
N_LAYERS        LONG                2
YIELD           FLOAT     Array[96, 2]
LAYERS          STRUCT    -> LAYER Array[2]
UNKNOWN         INT             1
E_BEAM          FLOAT         18.5000
THETA           FLOAT         180.000
PHI             FLOAT        0.000000
ALPHA           FLOAT        0.000000
BETA            FLOAT        0.000000
TITLE           STRING    'Ni Micromatter (XFM thin)'
FILE            STRING    ''
FREE            INT       Array[96]
FORMULA         STRING    Array[2]
WEIGHT          INT       Array[2]
THICK           FLOAT     Array[2]
MICRONS         INT       Array[2]
DENSITY         FLOAT     Array[2]
Z1              INT             0
A1              INT             0
STATE           FLOAT         1.00000
EMIN            FLOAT         3.00000
EMAX            FLOAT         48.0000
MU_ZERO         FLOAT     Array[20, 96]
RATIO_YIELD     FLOAT     Array[384, 96]
ARRAY           INT             1
DETECTOR_FILE   STRING    'C:\...\AS-Maia-384-array.detector'

RATIO_INTENSITY FLOAT     Array[384, 20, 96] new rel-ints per detector
```

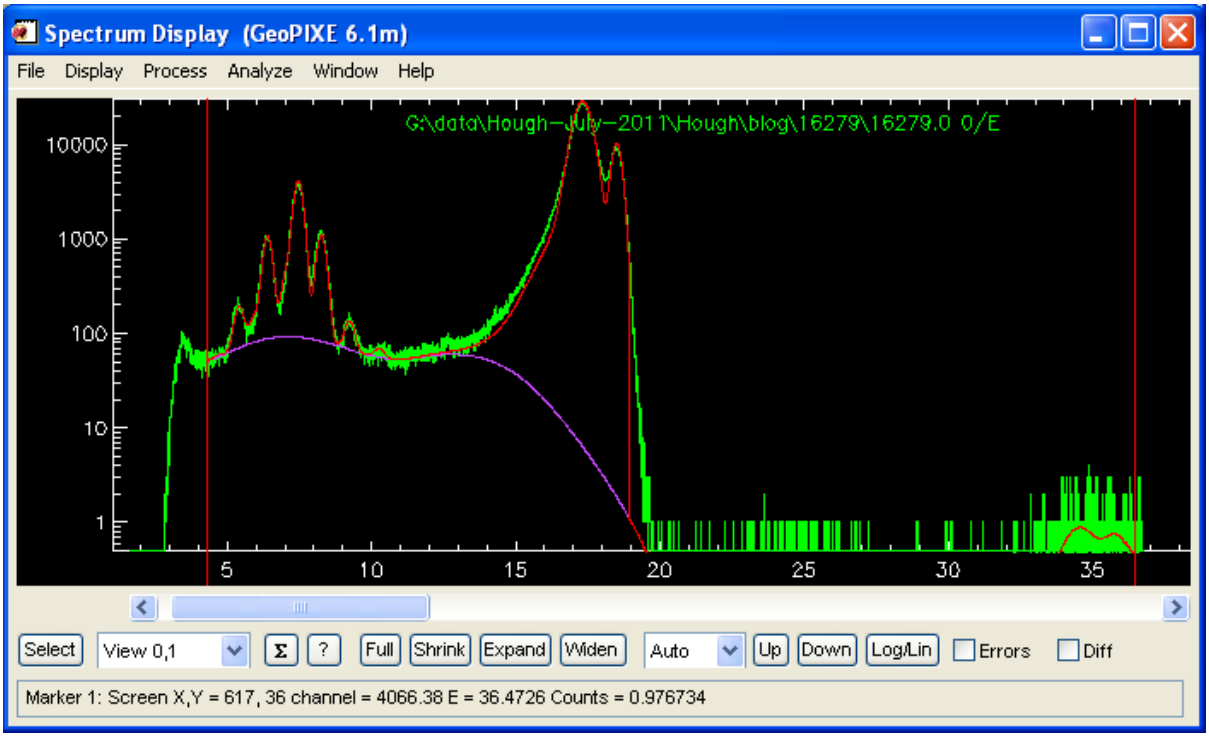
'Geo_yield2' first constructs an array of 'layer' structs, to define the layered sample structure, using the input data for chemical formulae, thickness and density. It then calls 'calc_slices2' to sub-divide the coarse layers into smaller 'slices' that will be used for modelling energy loss (for ion beams), X-ray absorption and production yields. These 'slices' start small near layer boundaries and the sample surface, so that detail is not lost in the integration, and then grow away from boundaries for more efficient integration. The aim is the minimum number of slices that maintains energy loss and integration accuracy, while making the process as fast as possible. For an ion beam, it determines the slowing down of the beam through the slices (actually it uses half slices for integration steps) in 'slow_beam'. In the case of a continuum source, the beam spectrum is modified for any beam filters in 'harden_beam'.

After all X-ray lines and relative intensities are retrieved using 'get_lines', in the case of a continuum beam, the XRF relative intensities get modified integrating over all energies in the beam. Then an array of mass absorption coefficients 'cmux' is generated for half slices to be used to model X-ray absorption through the sample slices. And X-ray production cross-sections are calculated for the beam energy (or range or energies). Next 'calc_yield' is called to integrate PIXE/SXRF yields using a 4th order Runge Kutta integration through half slices, optionally including an inner loop to include secondary fluorescence effects. Finally, for a multi-layer sample structure, the overall X-ray relative intensities are determined by including contributions from all layers. These relative intensities will be used in X-ray spectrum fitting.

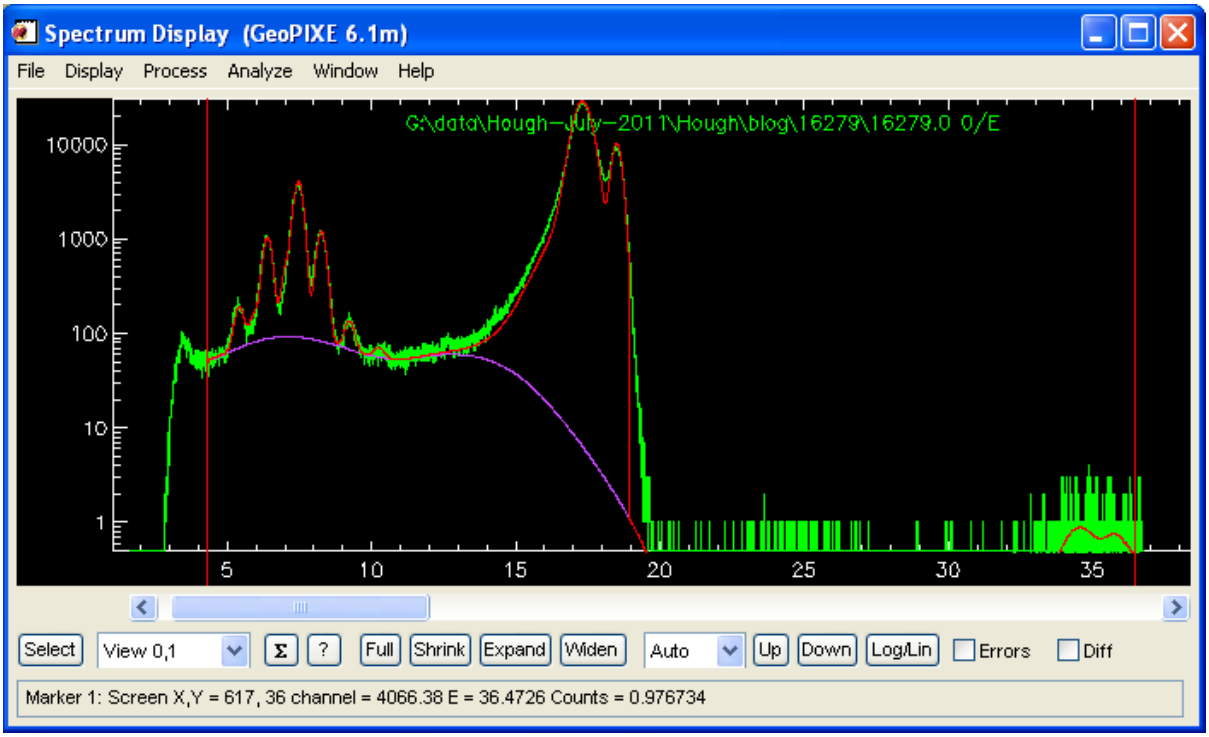
Example

See example in AS\July-2011\Hough\analysis\16279.

Sum spectrum



Fit before rel-int corrections across array applied.



After rel-int correction across array. Note improved fit to Ni Kb. Also changes fit value for Ni by 3% (see debug.pfr).

Calc_yield

Calculate PIXE/SXRF (or PIGE) yield values for all lines/elements at each slice. Lines are in descending intensity order, so major line is always index 0.

Note that 1 µC of charge is assumed. For photons this means 6.242 x 10^{12} photons.

Input:

state	charge state
cos_beam	cosine for beam incidence (made by 'experimental_angles')
cos_detector	cosine for outgoing X-rays ('experimental_angles', -ve for transmission target)
slices[j]	struct array giving details of each target slice ('calc_slices')
z2[i]	atomic numbers of target elements (from 'get_lines')
shell[i]	shell index (K=1, L=2, M=3) of target elements (from 'get_lines')
e_bind[i]	binding energy / edge
xsect[i,n]	cross-section for each element, at half-slice steps 'n' ('calc_cross')
xsect[i,e]	or across continuum spectrum 'e' if continuum mode ('calc_cross')
cmux[k,i,n]	mass-atten coeff. for line 'k', element 'i', half-step 'n' ('calc_abs')
e_lines[k,i]	X-ray line energies, line 'k', element 'i' (only used in sec fl.)
branch_ratio[k,i]	branching ratios (from 'get_lines')
branch_ratio[k,i,n]	over [k,i,n] for half-slices 'n', for continuum spectrum source
fluoro[i]	fluorescent yields (from 'get_lines')
n_lines[i]	number of X-ray lines for each element 'i'
n_layers	number of layers
lid[j]	layer ID of each slice (from 'calc_slices')
relmux[k,i,i2,l]	relative mux for source line 'k', element 'i', fluorescing destination element 'i2' in layer 'l'
/sec_fl	enables secondary fluorescence (requires relmux too).
/photo	photon induced ionization.
beam	continuum beam spec, if present
flux	relative flux decay from surface.

Note that 'cmux' and returned 'xsect' are both in half-slice steps (start at surface).

Output:

intensity[k,i,l]	relative intensities (rel to major) for line 'k', element 'i', for layer 'l'
yieldx[h,i]	yield of major line, element 'i', half-slice steps 'h'
sec_yield[]	yield due to secondary-fluorescence
zero[i]	flags zero xsect for this element

Returns:

major_yield[i,l]	yield of major line for element 'i', layer 'l'

Event mode sorting and processing

GeoPIXE relies on raw data in the form of X-ray events, which typically contain E (ADC value related to detected X-ray energy), n (detector index or array channel number), X (stage X position, which may not be spatial, e.g. energy or angle) and Y (stage Y position). Maia also has a T (time-over-threshold, which is related to dead-time). For flexibility, the returned “event” also includes a “multiplicity”, which is usually 1. However, this multiplicity simplifies converting saved spectra (for data types/devices that store whole spectra at each X,Y pixel position) into a virtual event steam by setting multiplicity to the count in a channel/bin of a spectrum and E to the channel number. The DA image sorting routine is ‘da_evt’, which calls ‘read_buffer’ to get the next buffer of events, contained in the vectors X,Y, E, n, T and multiplicity.

The device objects (see below for details) handle presenting input raw data into this event form. The routine ‘read_buffer’ takes its first argument ‘obj’ to be the device object. Each device object implements a method ‘read_buffer’ to return this vector data plus a count of returned events and a method ‘read_setup’, which is called for each opened input raw data-file before calling ‘read_buffer’ in a loop until end-of-file.

da_evt

The 'da_evt' routine reads event buffers using 'read_buffer' and accumulates DA images for the selected detector channels. It also accumulates other information and has other features:

1. Accumulates:
    a. DA images
        i. Element images based on DA matrix method, as well as 'error' images of statistical variance (related to 1-sigma error estimates for each element map pixel).
        ii. If an energy array DA matrix is used (e.g. for XANES image processing), then the energy closest to the current 'beam_energy' (returned as metadata from the 'read_setup'/'get_header_info' header read) is used.
        iii. If a 'proxy_axis' is used, then the DA matrix energy index comes from the value of the proxy axis in the event stream (e.g. Z axis).
    b. flux
        i. images of flux or beam charge (possibly for a number of ion chambers)
    c. dead_fraction
        i. image of fraction of events lost due to dead-time
    d. pileup_loss
        i. image of fraction of events lost due to pileup rejection method
    e. dwell
        i. dwell time in each pixel
    f. statistics
        i. A number of parameters counting total events processed, valid events, pileup losses, bad XY, etc.
        ii. Used to form the count-rate image at the end of processing.
2. Can process just one "stripe" of an image in cluster mode
    a. Used in parallel processing, it tackles just a part of the image based on the 'cluster_index' passed to 'da_evt'. All input files are spread over the total number of processes ('cluster_total') and each process uses 'cluster_index' to know which files it should process.
    b. A "Y lookup table" (or YLUT file) is used to know what Y scan values are in each raw data file. This is used to decide which raw data files go in each "stripe" of an image divided amongst the parallel processes.
3. Can process a "windowed sort", where just a subset of the image area is acquired.
    a. This still maintains good X,Y offsets so that absolute image coordinates remain good within the windowed image output.
4. Can use the new MPDA algorithm
    a. which uses the 'phase' maps and maps of inverse yields.
5. Can use a 'linearize' file
    a. To provide a mapping from non-linear onto a linear energy calibration, which is performed event-by-event, and may be different for each detector channel.
6. Can use a 'throttle file
    a. To provide a correction for throttling applied during data acquisition in the device.
    b. Maia can selectively "throttle" events to sub-sample strong/high-rate events (e.g. save only every 10^{th} occurrence), such as scattering or a major element peak.
    c. The 'throttle' file enables this sub-sampling to be corrected so the spectra look normal.
7. Can use a 'pileup' file
    a. To provide a selection of good events based on the expected E-T relationship.
    b. Maia uses T based on time-over-threshold, which shows a E-T curve which can discriminate pileup events.
8. Save in output DAI file (many are accessed using image plugins for display):
    a. DA images
    b. error images (variance maps)

c. flux maps (normalized after “flattening”)
d. raw flux map
e. dead fraction map
f. pileup loss map
g. count rate map
h. dwell time map
i. Optionally for Multi-Phase DA
    i. phase maps
    ii. inverse yields maps

For each raw data file, the object ‘read_setup’ method is called, which reads the file header (or associated info file), sets up the read buffer size and data type and initializes parameters tracking processing between buffer reads. Then for each file, the method ‘read_buffer’ is called until end-of-file. It returns the event vectors, the total number of events and a vector ‘veto’ which is ‘0’ for all normal X-ray events and ‘1’ for special events used to convey extra data, such as flux counts and dead-time. These are processed in the device object ‘read_buffer’ method, and so ‘da_evt’ just skips these. The ‘dead_fraction’ and ‘flux’ arrays are accumulated within the ‘read_buffer’ method in a device specific manner (some devices have flux data in the event stream, others have a map stored, etc.). ‘da_evt’ calls ‘da_accumulate11’ (‘da_accumulate10’ in MPDA mode) to accumulate element image values based on the event stream vectors and the DA matrix (this routine is in compiled Fortran and optimized for speed, as are many device routines, such as the dead-fraction and flux accumulation routines in the Maia device object ‘read_buffer’ method).

Once event processing is completed, a number of normalization steps are performed on ‘da_evt’:

1. Image finite
    a. Checks for any non-finite pixels (NaN values) and sets them to zero (uses ‘finite_image’ routine).
2. Correct for multiplicity
    a. The element images and variance maps are scaled down by the effective ‘multiplicity’, which is defined as the sum of the gamma values (relative sensitivity factors) (uses ‘multiplicity_scale’ routine).
3. Smoothing of flux and dead_fraction
    a. For only some devices, the dead_fraction, flux and dwell maps may be noisy. The dwell and weight get smoothed in object method calls triggered by ‘da_evt’. The smoothing width is controlled in the Device panels. If the device does not support the smoothing of flux or dead-time, nothing is changed (falls back to a null method in the base device).
4. Form count-rates
    a. The total number of events per pixel map is normalized to dwell to form a count-rate map.
5. Form pileup losses map
    a. The total number of pileup events per pixel map is normalized to total count per pixel.
6. Normalize dead_fraction
    a. The dead_fraction array may need to be normalized by dwell per pixel or some other weight.
    b. If the method ‘get_dead_weight’ returns a map, then the dead_fraction must be normalized by dividing by this weight.
        i. The way the weights work varies according to the method ‘get_dead_weight_mode’ (0 = weight is incoming count estimate, or sum dwell and 1 = weight is outgoing raw count weight). Note: in practice a test for finite denominator is done first and only these pixels are normalized):
            1. dead_fraction = dead_fraction / weight
            2. if mode = 1 then need to correct for using the outgoing raw count instead of incoming (correct from OCR to ICR weights):

        a. dead_fraction = dead_fraction / (1 + dead_fraction)
      ii. The FalconX device object uses this, as it only counts outgoing events.
   c. If the method 'get_dwell' returns a map, then the dead_fraction must be normalized by dividing by this dwell map.
      i. dead_fraction = dead_fraction / dwell
      ii. The Maia device object uses this, as it accumulated total time-over-threshold for all events in a pixel, which then needs to be normalized to total time in each pixel.
   d. Test dead_fraction bounds
      i. If any pixel has dead_fraction more than 95% it is clipped to 95%.
   e. Correct dead_fraction also for pileup losses (effective dead fraction)
      i. dead_fraction = 1. - (1. - dead_fraction)*(1. - pileup_loss_map)
7. Correct flux for dead fraction
   a. Flux is assumed to represent "live flux" and so total flux per pixel is corrected for the cumulative effects of dead-time and pileup.
   b. flux[*,*,0] = flux[*,*,0] * (1. - dead_fraction)
8. Add "attribute" planes to image array
   a. The flux map(s) are added as extra planes to the element image array.
   b. A variance map(s) is also added to 'error'.
9. "Flatten" by flux
   a. Normalization to flux (to make it "flat") is performed (if /flatten option is used) in 'image_correct_flux', which corrects all planes.

Image regions and element Associations

Users can place and manipulate various shapes on images (in *Image* window, "gimage.pro") in order to determine average concentrations for features, track variation along lines or curves as a traverse or examine element-element correlations or in *Associations* ("corr.pro") as a probe of the phase makeup of a sample.

The main GeoPIXE image program allows selection of various closed shapes and spline curves for selection of areas. These have control points for moving, shaping or rotating the shapes. The selection can be done in "*Include*" mode, which means select the enclosed pixels as a region or "*Exclude*" mode, where any previously selected pixels that fall within the *Exclude* shape are removed from the selection. This provides a way to edit pixel selection. See the *GeoPIXE Users Guide* for details ("Image Region Selection Shapes" and the "Linear Traverses and Line Projections" section).

The *pstate struct in the main GeoPIXE image program (see "gimage.pro", "image_eventcb.pro") has an entry 'analyze_mode' for include (0)/exclude (1) and 'analyze_type' which is a vector of two indices into a list of possible shapes, one for *Include* and one for *Exclude* mode. A further pointer, pmark, in *pstate, is a 2 element pointer array. The first points to a pointer array for *Include* shapes and the second to a pointer array for *Exclude* shapes. Each points to a struct giving the X,Y pixel coordinates of the shape control points, with some having extra parameters: 'theta' (rotation), 'shear' (shear), 'curvature' (with "Curve 8" shape)

```
0 distance          2 ends
1 box               4 corners      centre handle   rotate handle
2 circle            2 diameters    centre handle
3 curve 8           9 spline pts   2 width handles (right mouse curvature)
4 traverse          4 handles      centre handle, 2 length, 2 width (right mouse shear)
5 ellipse           4 diameters    centre handle
6 spline 10         10 control points centre handle
7 spline 32         32 control points centre handle
8 projectX          4 handles      centre handle (no shear)
9 projectY          4 handles      centre handle (no shear)
10 spline 100       100 control points    centre handle
11 S pixel          1 point
```

When the "Σ" button is pressed (may shows as "S" if symbols not found), all pixels within the current shape (or those remaining after editing using *Exclude* shapes) are determined and saved as a vector of indices (in pointer 'q') in the *Image* *pstate struct. 'q' points to a vector of indices into the image 2D array. See the "analyze_image" routine, which determines average concentration within the selected pixels (when (*pstate).corr_mode = 0).

Element-element correlations or associations are displayed in the *Association* window. It plots the image concentrations of two selected elements as a scatter plot (see "Associations" in the users guide). Another way to do select pixels is to lasso features in the *Association* scatter plot. This selects these pixels, which are saved as another vector of indices in the pointer 'qc' in *pstate (see "analyze_image" when (*pstate).corr_mode = 1).

Regions are saved to pointer 'pregion' in *pstate for the *image* window. 'pregion' is a pointer array, each pointing to a region struct (see "define.pro") used by the *Image Regions* window:

Spectra extraction and model spectrum overlay

Spectra extraction is launched from the *Image Regions* window (or via callable GeoPIXE command file approach) on the "Extract" tab. It uses routine 'image_table_evt' (see file "image_table_eventcb.pro") to extract a spectrum for each image region in the table. This routine follows this method:

1. Use the search feature of 'file_requester' to make sure to locate all necessary files referenced from the region (files may have been moved since the regions were defined), especially the raw data files (also Linear, Pileup, Throttle and YLUT where needed, e.g. Maia device). The search looks locally in the dir tree (potentially 3 levels up and down from there).
2. Use the Device Object method 'select_evt_files' to determine the filtered raw data files list.
3. Determines memory needs and possibly breaks down extract into multiple passes.
4. Use the Device Object method 'trim_evt_files' to trim raw data file list (based on the X,Y sampled by all regions and reference to the YLUT, Y lookup table for cluster stripe).
5. Calls 'spec_evt' to do the extraction, either directly, or if "Cluster=1' is enabled, via 'cluster_client' to launch a number of background processes to work on subsets of the region list each. Once completed 'cluster_client' assembles the resultant temporary spectra file list.
6. Call 'cluster_merge_spectra' to merge all the temporary spectra from the sub-processes into the final spectra.
7. Save spectra to disk.
8. Pass (via Notify) the spectra pointer 'pp' to the *Spectrum Display* window for display.

spec_evt

The 'spec_evt' routine reads event buffers using the Device Object 'read_buffer' method and accumulates spectra for the selected regions. It runs in one of two modes: (i) full spectra extraction (e.g. called from the *Import* spectra menu of the *Spectrum Display* window), and (ii) region extraction (when provided with a 'pmask' pointer to region masks, which may get very large, or alternatively a 'mfile' argument providing an alternate method for passing regions mask data from a regions file). It also accumulates other information and has other features:

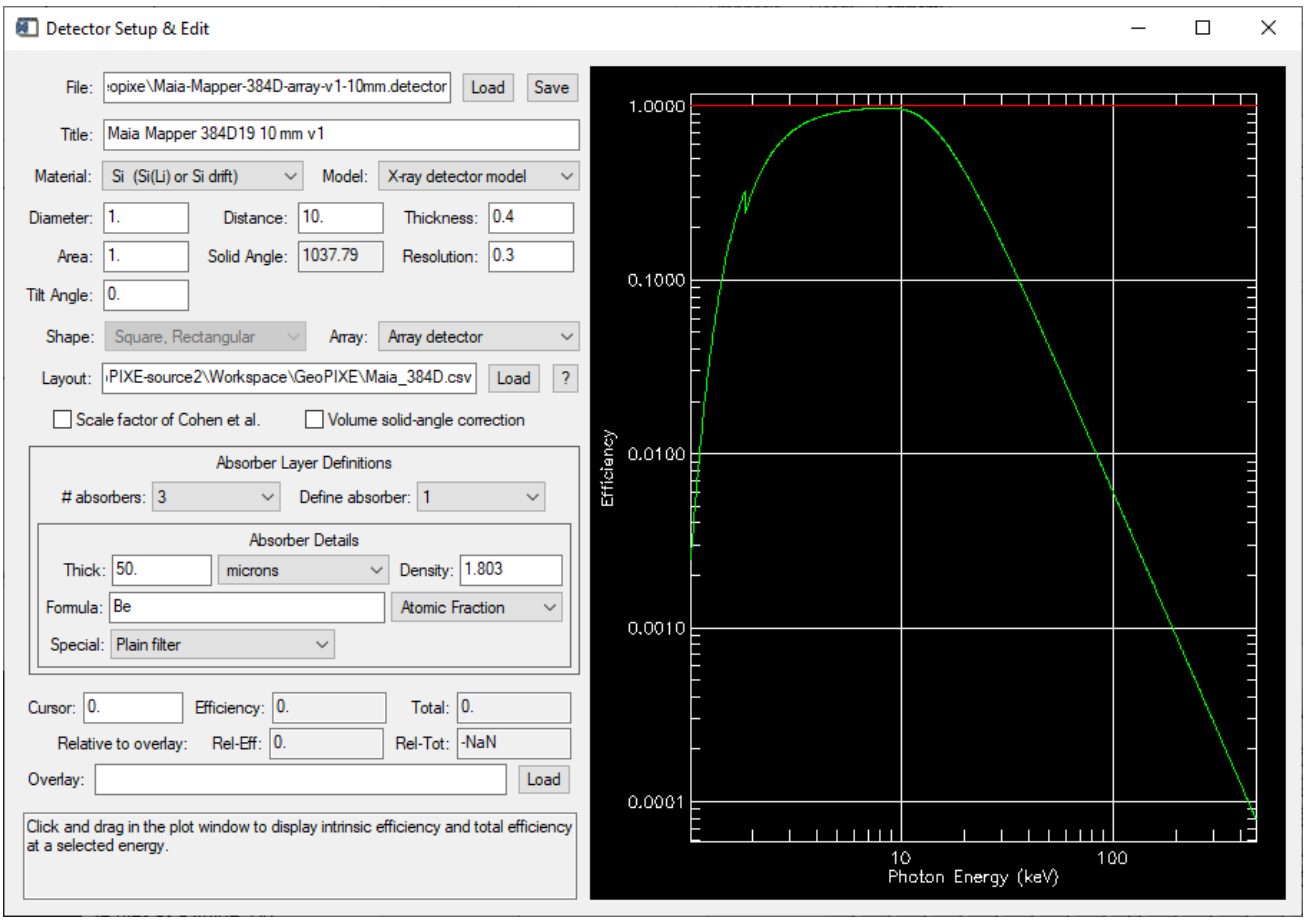
1. Input data
    a. Reads regions from a file, if 'mfile' method used.
    b. Reads in linearization, pileup field and throttle vector data if needed (e.g. Maia).
2. Modes
    a. If no 'pmask' or 'mfile' provided then in total spectra mode (import all spectra).

   b. If ‘pmask’ or ‘mfile’ provided and ‘by_detector’ = 0, then do all regions (detector array data combined across all detectors, after remapping for linearization and energy calibration).
   c. If ‘pmask’ or ‘mfile’ provided and ‘by_detector’ = 1, then do all detector array spectra separately, but only for one selected region (given by ‘by_detector_select’).
3. Accumulates
   a. Use Device Object ‘range_ylut’ method to determine

Detector specifications

The .detector files (transportable XRD binary) contain all parameters for a detector to enable modelling its efficiency. In the case of an array detector, an additional file (.csv referenced from the .detector file) contains a table providing the layout of detectors in the array.

Detectors are setup in the ‘detector_setup’. It permits the size, shape and thickness of a single detector to be specified, as well as any absorber layers over the sensitive volume, which may include windows, contacts and dead layers. Si and Ge detectors are supported. The detector may have a tilt relative to the incoming X-ray (i.e. direction to beam spot on sample). For thick detectors, can use a volume correction to the effective solid-angle. This is not used if the detector solid-angle is defined by a thick mask.



In the case of a detector array, the position of each detector in the array is given in a table relative to the “central detector”, which is the single detector above. It provides the thickness and built-in absorbers assumed for all detectors in the array. The layout permits grids of detectors of various shapes, in a plane or on a general surface with individual tilt and axial offset to follow some other surface, such as a hemisphere. The table also permits each detector to be masked to a different size (solid-angle), have individual tilt

(relative to the axis of the array) and have individual offset Z along the detector axis. The “Data” column provides the detector number identifier for each detector in the array.

# This file produced by program: "write_detector_layout.pro"													
title	Maia 384 Si array												
N	384	Number of detectors											
Start	0	Start at this # (default: detector numbers start at 0)											
Shape	1	Shape of c	1=square)										
Symmetry	4	Detector array symmetry steps for 360 degree (default is 4)											
Reorient	0	Re-orient or rotate by "reorient*360/symmetry" degrees											
MirrorX	0	Mirror the layout in X											
MirrorY	0	Mirror the layout in Y											
Veto	0	Veto channels with excessive FWHM											
Threshold	200	FWHM threshold (eV)											
# ID	X	Y offsets f	Z offset fi	Width	Height (mm)	Tilt (degrees)	bad (0=good	FWHM (eV)	Hermes	Quadrant	Radial	Column	Row
Data	X	Y	Z	width	height	tilt	bad	FWHM	Hermes	Quadrant	Radial	Column	Row
298	-9.252	9.252	0	0.548	0.548	0	0	0	9	3	12	0	0
290	-8.277	9.252	0	0.583	0.548	0	0	0	9	3	11	1	0
297	-7.303	9.252	0	0.619	0.548	0	0	0	9	3	11	2	0
289	-6.328	9.252	0	0.654	0.548	0	0	0	9	3	10	3	0
296	-5.354	9.252	0	0.689	0.548	0	0	0	9	3	10	4	0
288	-4.379	9.252	0	0.724	0.548	0	0	0	9	3	9	5	0
295	-3.405	9.252	0	0.759	0.548	0	0	0	9	3	9	6	0
294	-2.43	9.252	0	0.794	0.548	0	0	0	9	3	9	7	0
85	-1.456	9.252	0	0.829	0.548	0	0	0	2	0	8	8	0
80	-0.481	9.252	0	0.864	0.548	0	0	0	2	0	8	9	0
67	0.481	9.252	0	0.864	0.548	0	0	0	2	0	8	10	0
70	1.456	9.252	0	0.829	0.548	0	0	0	2	0	8	11	0
57	2.43	9.252	0	0.794	0.548	0	0	0	1	0	9	12	0
48	3.405	9.252	0	0.759	0.548	0	0	0	1	0	9	13	0
47	4.379	9.252	0	0.724	0.548	0	0	0	1	0	9	14	0
38	5.354	9.252	0	0.689	0.548	0	0	0	1	0	10	15	0
25	6.328	9.252	0	0.654	0.548	0	0	0	0	0	10	16	0

Maia detector layout file (only shows 17 out of 384 detectors in the array)

The ‘read_detector’ function reads .detector files. The Maia detector array (384 detectors in a square grid, with 16 missing in the centre, to form an annular array) has a data structure of the form (it references the layout file shown above):

```
** Structure <253c1940>, 28 tags, length=1240, data length=1198, refs=1:
   ABSORBERS       STRUCT    -> FILTER3 Array[3]
   CRYSTAL         STRUCT    -> LAYER Array[1]
   DIAMETER        FLOAT           1.00000
   DENSITY         FLOAT           2.32200
   DISTANCE        FLOAT           10.0000
   SOURCE          FLOAT         0.0100000
   AEFF            FLOAT           0.00000
   BEFF            FLOAT           0.00000
   GAMMA           FLOAT         0.0220000
   W0              FLOAT         0.0776853
   W1              FLOAT        0.00208900
   TAIL            STRUCT    -> <Anonymous> Array[1]
   RESOLUTION      FLOAT           0.300000
   TILT            FLOAT           0.00000
   USE_SPECIAL1    INT               0
   USE_SPECIAL2    INT               0
   CORRECT_SOLID_ANGLE
                   INT               0
   SPECIAL1        STRUCT    -> <Anonymous> Array[1]
   SPECIAL2        STRUCT    -> <Anonymous> Array[1]
   COHEN           INT               0
   PIGE            INT               0
   A               FLOAT     Array[6]
   E_LOW           FLOAT          0.100000
   E_HIGH          FLOAT           4.00000
   FILE            STRING    'C:\Software\IDL\GeoPIXE-open-source\Workspace\geopixe\Maia-Mapper-384D-array-v1-10mm.detector'
   LAYOUT          STRING    'C:\Software\IDL\GeoPIXE-source2\Workspace\GeoPIXE\Maia_384D.csv'
   ARRAY           INT               1
   SHAPE           INT               1
```

Each absorber uses the Filter structure (e.g. define(/.filter)).

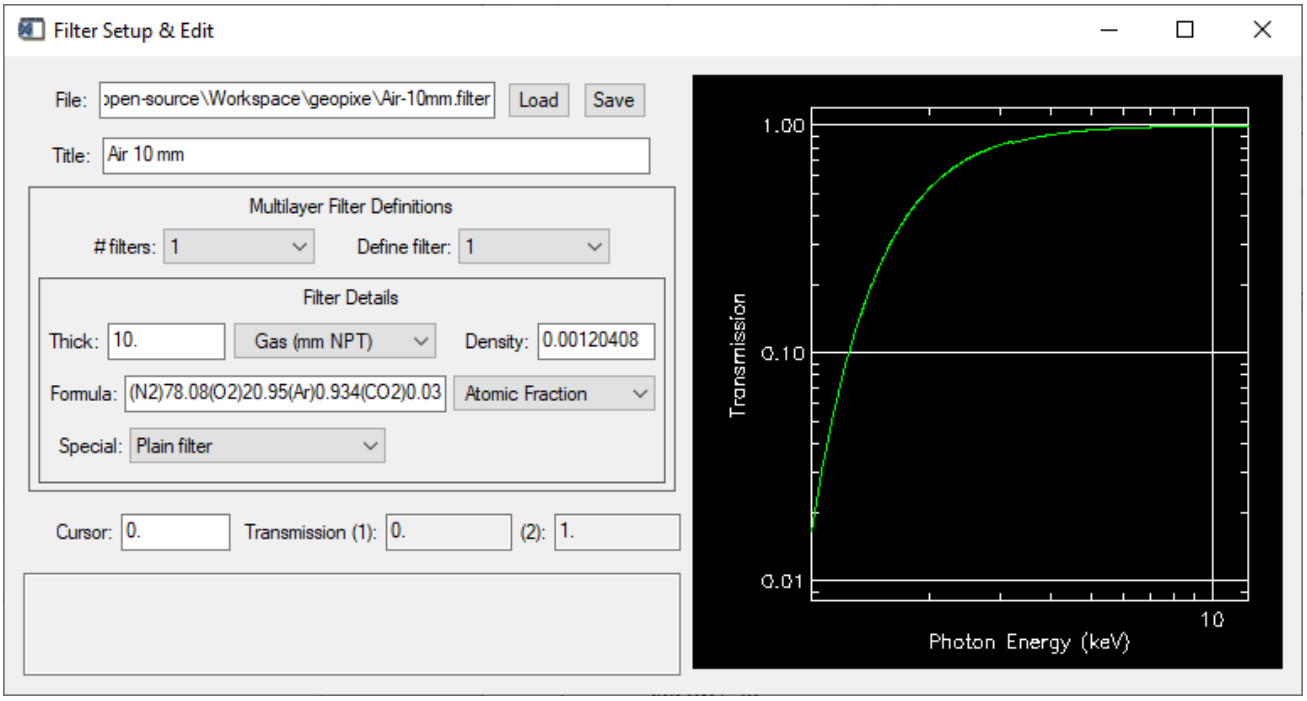
```
python3.8 IDL 8.8 open> help,(*d).absorbers[0]
** Structure FILTER3, 12 tags, length=280, data length=268:
   N               INT             1
   Z               INT       Array[32]
   F               FLOAT     Array[32]
   THICK           FLOAT           9.01500
   PINHOLE         INT              0
   PINRATIO        FLOAT           0.00000
   NAME            STRING    'Be window'
   MICRONS         INT              1
   DENSITY         FLOAT           1.80300
   FORMULA         STRING    'Be'
   WEIGHT          INT              0
   BRAGG           STRUCT    -> BRAGG Array[1]
```

The active volume ‘crystal’ is given by the simpler layer struct:

```
python3.8 IDL 8.8 open> help,(*d).crystal
** Structure LAYER, 5 tags, length=216, data length=216:
   N               LONG                 1
   Z               INT       Array[32]
   F               FLOAT     Array[32]
   THICK           FLOAT           92.8800
   NAME            STRING    'Maia Mapper 384D19 10 mm v1'
```

Filter specifications

The .filter files (transportable XRD binary) contain details of filters, which may have compound composition, layers, pin-holes or a gas composition. They are setup using ‘filter_setup’. The ‘Formula’ follows the general GeoPIXE practice of element mnemonic with an atomic fraction weight or molecules in brackets with a weighting outside that can be atomic fraction or weight% if the mode is switched to “Weight %”. Normally, thickness is in mg/cm^2 or microns (a density must be supplied). For gases, thickness is in mm at NPT.



A “special” option is used to select pin-hole filters, specified as the solid-angle ratio of the whole detector relative to the pin-hole

A filter specification is read in using ‘read_filters’ (plural because the file can contain an array of filters). This defines a layer of material (given by N, Z, F to provide ‘N’ components with the atomic fraction ‘F’ and element of atomic number ‘Z’). A logical ‘pinhole’ can indicate an optional pin-hole specified as the solid-angle ratio of the whole detector relative to the pin-hole (‘pinratio’). Microns=1 would indicate a filter setup using thickness in microns and a supplied density. However, by convention, Thickness (‘thick’) is always in mg/cm^2. ‘Formula’ is the chemical formula of the filter and the logical ‘weight’ would indicate a formula specified using weight% (outside of any radicals in “()” brackets); the default is atomic fraction.

An example is a pin-hole filter (100 micron Al with a 53.8 pinhole ratio combined with a 250 micron (39.4 mg/cm^2) Be filter (“Be-250-Al-100-pinhole.filter”):

```
python3.8 IDL 8.8 open> help,(*f)[0]
** Structure FILTER3, 12 tags, length=280, data length=268:
   N               INT              1
   Z               INT       Array[32]
   F               FLOAT     Array[32]
   THICK           FLOAT           26.9900
   PINHOLE         INT              1
   PINRATIO        FLOAT           53.8000
   NAME            STRING    '250 Be + 100 Al pinhole '
   MICRONS         INT              1
   DENSITY         FLOAT           2.69900
   FORMULA         STRING    'Al'
   WEIGHT          INT              0
   BRAGG           STRUCT    -> BRAGG Array[1]
python3.8 IDL 8.8 open> help,(*f)[1]
** Structure FILTER3, 12 tags, length=280, data length=268:
   N               INT              1
   Z               INT       Array[32]
   F               FLOAT     Array[32]
   THICK           FLOAT           39.4000
   PINHOLE         INT              0
   PINRATIO        FLOAT           0.00000
   NAME            STRING    ''
   MICRONS         INT              0
   DENSITY         FLOAT           1.80300
   FORMULA         STRING    'Be'
   WEIGHT          INT              0
   BRAGG           STRUCT    -> BRAGG Array[1]
```

Object Oriented Device Driver Modules

GeoPIXE now has plugins that contain all the device dependent IDL code to handle specifics of list-mode data-streams, local metadata and miscellaneous local spectrum file formats. They are implemented using IDL Object techniques, with one new Object class per device handler, plus a Generic_device class, to handle common formats such as ASCII, and a Base_device, which is the super-class with methods that handle many common tasks for all devices.

All device driver source files have file-names of the form “XXX_device__define.pro” for device “XXX” (note the double underscore “__” before the “define”) and when compiled they have names of the form “XXX_device__define.sav”. These files should reside in the “interface” sub-directory to the runtime “geopixe” directory. From here they will be detected and loaded automatically by GeoPIXE and used to populate all device droplists (e.g. in the “Sort EVT” and “Import Spectra” windows).

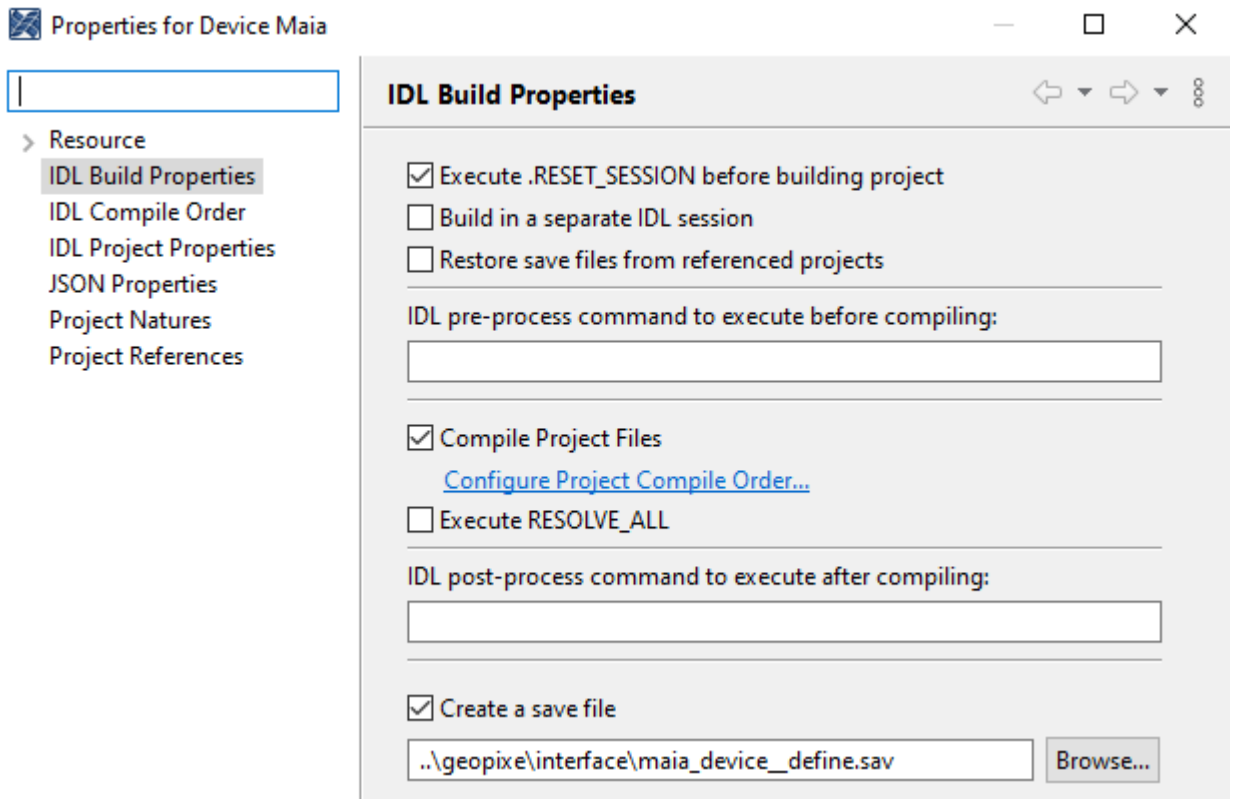
Device methods

Writing and using device methods, start by using features of existing device object define files as a guide. Do not base a new device object on the Base or Generic device code. Clone one of the existing "XXX_device__define.pro" files (in a new Project) and rename all occurrences of "XXX" to the new device name. Make sure that this new name is unique. Store the files in the "interface" sub-directory to the main GeoPIXE directory.

Building Device Object SAV files

When building SAV files for a new Device Object class, as with spectrum and image plugins, compile the main "XXX_device__define.pro" file and only resolve calls and routines that are local to your device (i.e. all files in Device Project dir) and **NOT part of GeoPIXE** (i.e. your local routines and standard IDL). Never use "resolve_all". Then save the compiled routines to "XXX_device__define.sav" in the "interface" sub-directory to the main GeoPIXE directory. When the device object is used, all object routines will already be loaded as part of GeoPIXE.

The simplest approach is to use "builder.pro", as detailed above in "Building GeoPIXE", and select "device" to build all Device Objects into their respective SAV files in "geopixe/interface".



Example of Project properties for the Maia device project. All Maia specific files are compiled but "resolve_all" is NOT used. The output goes in the "interface" directory.

Initialization

When all device classes are detected and loaded into GeoPIXE (all files of the form "xxx_device__define.sav" or "xxx_device__define.pro" in the "interface" sub-directory of the GeoPIXE runtime directory, excluding "BASE_DEVICE" and "GENERIC_DEVICE"), an Object reference is created for each using the IDL call "Obj_New('xxx_device')". This executes the "Init" method in the class.

init()	initialize the class definition and also defines the main object parameter struct in the xxx_device__define" routine. DO NOT RUN THIS METHOD (it is done by IDL when GeoPIXE creates each device object).
cleanup	this is run when an object is destroyed (DO NOT RUN THIS METHOD).

The init method defines the 'self' struct which contains the key parameters that define the properties of the device. They are set in a call to the base::init method from your device::init method. The parameters are:

Name	the unique name for this device (must end in "_device"
Title	device title (used in device droplists in GeoPIXE)
Ext	the file extension for raw list-mode data (if it is fixed)
Multi_files	1=flags that raw list-mode data occurs in multiple files
Multi_char	defines the character sequence that separates the raw data-file stub from the part that indicates the sequence number or XY coordinates (if used)
Big_endian	1=flags that the raw list-mode data has Big Endian byte order
Vax_float	1=flags the use of VAX D floating format in the raw data
Start_adc	defines the number of the first detector ADC (0 or 1)
Synchrotron	1=flags the use of this device for synchrotron data
Ionbeam	1=flags the use of this device for ion-beam data

The values for the following options can also be set-up in the device::init method.

self.options.scan.on = 1	scan sort options available for this class to be displayed as widgets in the "Sort" tab of the Sort EVT window (see below).
self.options.scan.ysize = 75	Y size of sort options box, when open

Similarly, the use of pileup, throttle and linearization can be flagged this way:

```
self.pileup.use = 1
self.throttle.use = 1
self.linear.use = 1
```

And, if the device can support the use of a Y lookup table to reduce processing times when extracting spectra from regions (only process multifile input files that are needed for the regions selected), this can be flagged this way:

```
self.header.scan.use_ylut = 1
self.header.scan.pYlut = ptr_new(/allocate_heap)
```

and a method 'trim_evt_files' will be needed. The Y lookup table is used with the extraction of spectra from regions in an image using the EVT button on the *Image Region* window. The 'trim_evt_files' method checks the Y lookup table against the regions selected and returns only the data file-names needed to provide data for these regions. In the Maia device, the Y LUT is just the first Y for each blog data segment file. Hence, only the Y values spanned by all regions is checked. It is possible to write more general lookup tables and methods that also check X.

For examples see supplied device IDL pro files to use as templates, such as:

Ion-beam:

FASTCOM_MPA3_DEVICE	for list-mode data in a single file

MPSYS_DEVICE	for a single list-mode file and example of the 'get_header_info' method to read header information from an associated metadata file.
LUND_KMAX_DEVICE	for multi-file list-mode data that includes import of some local spectra file formats

Synchrotron:

NSLS_HDF_DEVICE	for list-mode data in a single file.
NSLS_MCA_DEVICE	for multi-file synchrotron list-mode data that includes the 'flux_scan' method to look for ion chamber PVs and for import of some local spectra formats.
MAIA_DEVICE	for a multi-file list-mode format stream that includes 'flux_scan' as well as setting some extra variables local to the device through custom device widgets added to the Sort EVT window ("Device" tab) by the device and the use of Y lookup tables and the 'trim_evt_files' method.

Many of the devices use calls to Fortran library routines to speed up processing within the 'read_buffer' method, others just use IDL code. This choice will depend on how large the data files are and on whether the list-mode data stream has a simple recurring format or has a more complex structure that demands fiddly processing. The former can make use of IDL vector processing, while the latter will most often be done more efficiently using compiled languages (Fortran or C code).

Base-device methods available

Some general-purpose methods used by GeoPIXE are available in the BASE_DEVICE master-class to avoid writing these for each device class. These are:

name()	return name of device (e.g. "MAIA_DEVICE").
title()	return title string for this device (used in Sort EVT droplist)
extension()	return raw data file extension (if fixed, else '')
multi_files()	1=data organized in multiple files, else=0
multi_char()	character used to separate run name/number from numeric data-file series
big_endian()	1=flags data stored in raw data in Big Endian byte order, else=0
vax_float()	1=flags VAX D-floating variables as part of data header
start_adc()	# of the first detector ADC.
pileup()	1=flags the use of a pileup rejection file for this device
throttle()	1=flags the use of a Throttle mechanism for this device
linear()	1=flags a linearization correction table used for this device
ylut()	1=flags that this device can use and generate a lookup table of first Y for each member of a multi-file data series to speed up certain operations (e.g. spectra extract using EVT button on Image Regions window).
get_ylut()	retrieve the Y lookup table data from file.
write_ylut	write a Y lookup table file (called from the DA image sort).
use_bounds()	1=flags that this device may have a border of pixels that contain no data and no beam charge/flux that should be excluded.
smooth_flux	smooth the flux map using the smooth parameter in the options panel.
smooth_weight	smooth the weight map (for DT) using the smooth parameter in the options panel.

Reading list-mode data to produce images and extract full spectra

All device object classes need to implement these methods, with a full parameter list (even if some parameters are not used). The first 3 are not called as methods (i.e. "obj->method") but instead via a wrapper routine of the same name, which ensures good value typing (e.g. see "da_evt.pro").

read_setup()	will be called after each data file that is opened to setup internal device parameters (often as variables in common[1] blocks) needed for reading data buffers, such as buffer size and device-specific buffer organization.
read_buffer()	called repeatedly to read buffers from the data file, process these to extract X,Y,E triplet data, tagged by detector channel STE, compress X,Y,E if needed, and optionally detect other information (e.g. flux/charge, energy tokens).
get_header_info()	interrogate the data files (usually prior to starting processing) for various details, such as scan size (physical size and/or pixel count), title, energy cal for detectors, etc. Called, for example, when new data files are selected in the Sort EVT window. Needs good metadata in the data to really work well.

The above 3 methods (plus the init and cleanup methods) are the essential minimum set needed to be written for a new device class.

flux_scan()	scan the raw data-files for details of available ion chamber specifications (e.g. EPICS PVs) to provide for user selection, and select one to use, and the pre-amp sensitivity value and units.
trim_evt_files()	trim the list of files to only include files needed for the Y range seen in the region mask arrays (uses the information in the Y lookup table; used with EVT button on *Image Regions* window).

Device specific Import of spectra

Import of spectra is achieved in two modes: (i) using user-written IDL code to read local format spectrum files, and (ii) using the GeoPIXE "spec_evt" routine to scan through list mode file(s) set to extract all data from all detectors, together with the X and Y projections of all events for each detector. The latter creates a new file with the same name as the input data file (or first file in a set) and the suffix "i.spec'.

Local spectrum formats are handled by local IDL routines. They are defined in the 'get_import_list()' method by setting up a parameter structure called 'opt' for each import format or mode (a template for 'opt' is given in the example device code). Just the following parameters in opt are needed for local spectrum import:

Name	a unique name for the import (usually created as the device name plus a few extra characters).
Title	A string to identify this import mode in the "Import Spectra" window.
In_ext	The file extension (if it's fixed) for the import spectral data.
Request	A string to use as a title for the *File-Requester* pop-up window.
Device_name	The unique name for the device ('self.name')

The latter mode for extracting spectra from list-mode data-streams requires more parameters:

Spec_evt	1= flags this import as an extract from list-mode file mode
Multifile	1= flags this as a multi-file set (defaults to not multifile)

[1] Common blocks are often considered crude but have the advantage over fields within the object's 'self' structure of being able to be passed by reference (by name) into subroutines that need to return updated values for these variables. Passing 'self.var' is only passed by value in IDL and cannot return a new value into 'self.var'.

Separate	the characters that separate file-name from XY tags or run number (if used)
Raw	1= flags that the raw list-mode data can reside in a different file path to the output spectrum files created by this import.
Use_IC	1= flags the need to pop-up the flux selection window to allow IC PVs to be selected (or scanned in the data using the 'flux_scan' method).
Xr	default X range
Yr	default Y range

Some parameters are probably only needed for the Maia device, unless needed for your device too:

Use_linear	1= request linearization file
Use_pileup	1= request pileup file
Use_throttle	1= request throttle file
Use_tot	1= collect time-over-threshold (ToT) spectral data too

The method 'get_import_list()' is called by the *Import Spectra* window to build the selection lists and tables of devices for Synchrotron and Ion-beam local data formats. When one of the local spectrum file imports is selected, the method 'import_spec()' is called on the selected device object class to execute some local routine to read the spectrum data. If an extraction from list-mode import is selected, GeoPIXE handles this internally using 'spec_evt'. The spectrum reading routines should create a GeoPIXE spectrum struct using a call to "define(/spectrum)" and return a pointer to this struct (or an array of these structs for multiple spectra import).

Device specific parameters

If the device has some device specific parameters that need to be set-up for processing and read/written along with image and region data, etc. to/from disk, then use this facility to define widgets to gather info about parameter options and manage them. If you don't need them do not define these methods, then a default (no action) method will be used in the "BASE_DEVICE" superclass. These options are set-up in widgets that appear in the *Sort EVT* window options box on the "Sort" tab (also on the *Import Spectra* dialog). The parameters live in the class 'self' struct, defined in the device class 'init' method, and are handled using these methods:

render_options	draws device option widgets needed in supplied parent base (on "Sort" tab)
read_options	called when images and regions are read from disk to read the device specific options into the object self struct.
write_options	called when images and regions are written to disk to write the device specific options from the object self struct.
options_legend()	Returns a formatted string array to be added to the image history list in *the Image History* window to show device specific parameters.
set_options	explicitly pass these options parameters into the object. This should only be used internally to your object code to set option parameters.
get_options()	Explicitly get options parameters from Maia object. This should only be used internally to your object code to get option parameters.

The following two are used with Options widgets, called by GeoPIXE to the device object, but are handled by the Base super-class:

show_sort_options()	flags that this device has sort options to display.
get_sort_ysize()	returns number of Y pixels needed for device options fields.

To define and use additional device specific Options parameters, set these parameters in your 'init' method code. The size needed is set-up in the device Init method using:

self.options.scan.on = 1	scan sort options available for this class to be displayed as widgets in the "Sort" tab of the Sort EVT window .
self.options.scan.ysize = 75	Y size of sort options box, when open

The code in the *render_options* for creating options widgets for the Maia device, which also calls some OnRealize… routines and an event handler, can be used as a model for new device options fields.

Device Object programming notes

List-mode File organization

List-mode files can be organized in a number of ways:

1. **Single data stream** containing detector energy (E) data as well as pixel coordinates (XY). May also contain flux data. For single long files, the method 'read_setup' sets up initial parameters (perhaps reading a header or some associated file) and then sequential large buffers are read and processed by method 'read_buffer'.
2. **Multiple files** holding a single logical data stream. In this case, the method 'read_setup' opens each file and sets up initial parameters (perhaps reading a header) and then for each file sequential large buffers are read and processed by method 'read_buffer'.
3. **Pixel spectra files**, one for each pixel, that contain E spectra, and perhaps other information (e.g. energy calibration, dead-time, flux IC count, IC preamp sensitivity and range, ...). The initial opening of each pixel file is done by method 'read_setup' and then 'read_buffer' fills in the variables E, X, Y to return the pixel data to be processed, effectively like a list of events with multiplicity.
4. **Line of pixels** (e.g. NSLS_NETCDF device, with data for each line stored). In this case, the method 'read_setup' opens the file and sets up initial parameters (perhaps reading header attributes) and then for each Y line, sequential large buffers are read and processed by method 'read_buffer'.

The main job of the 'read_buffer' method is to fill in the vectors E for energy of each event and X,Y, which are the scan coordinates for the corresponding event and STE, which contains the index of the detector # for each event. The variable N contains the length of these vectors. If a particular E,X,Y,STE occurs many times, either list them (in any order) or set MULTIPLE to the count for the repeat. This is useful for presenting a spectrum in a list-mode form, by using Multiple to record the counts in a channel of the spectrum for a detector (e.g. "NSLS_MCA_DEVICE", "HASYLAB_FIO_DEVICE").

Header Info

Devices can provide a 'get_header_info' method to return details read in from a data file header, or some other associated metadata file or database. Often this method provides details of scan size (pixel count), energy calibration, scan size (mm) and beam energy. The full range of parameters that can be returned and usefully applied by GeoPIXE are given in the 'header' part of the device's 'self' variable (defined in the BASE device). The header part of the 'self' struct contains the following variables and default values:

```
header: {header_devicespec, $
        charge:         0.0, $          ; beam charge (uC)
        energy:         0.0, $          ; beam energy (MeV: ion-beam, keV: SXRF)
        title:          '', $           ; comment/title string
        sample:         '', $           ; sample name
        grain:          '', $           ; sub-sample/grain ID
```

```
metadata: {metadata_devicespec, $
                        sample_type:    '', $       ; sample type ("standard", "user", ...)
                        sample_serial:  '', $       ; sample serial/ID code string
                        facility:       '', $       ; facility string
                        endstation:     '', $       ; end-station string
                        detector_identity: ''}, $ ; detector identity code string

scan: {scan_devicespec, $
        on:             0, $                    ; scan active
        x_pixels:       0, $                    ; # X pixels
        y_pixels:       0, $                    ; # Y pixels
        z_pixels:       0L, $                   ; # Z pixels
        x:              0.0, $                  ; X absolute origin (mm)
        y:              0.0, $                  ; Y absolute origin (mm)
        z:              0.0, $                  ; Z absolute origin (mm)
        x_mm:           0.0, $                  ; X size of scan (mm)
        y_mm:           0.0, $                  ; Y size of scan (mm)
        z_mm:           0.0, $                  ; Z size of scan (mm)
        x_on:           0, $                    ; X axis in use
        y_on:           0, $                    ; Y axis in use
        z_on:           0, $                    ; Z axis in use
        x_name:         '', $                   ; X axis name
        y_name:         '', $                   ; Y axis name
        z_name:         '', $                   ; Z axis name
        mode:           0, $                    ; scan mode (0 XY, 1 Xstep, 3 Ystep)
        n_steps:        0, $                    ; number of steps per pixel
        step_size:      0.0, $                  ; step size in step mode
        sort_mode:      0, $                    ; sort mode (0 XY image, 1 Traverse)
        dwell:          0.0, $                  ; dwell time (ms)
        use_ylut:       0, $                    ; able to filter file list based on Y LUT
        pYlut:          ptr_new(/allocate_heap)}, $     ; Y lookup table

deadtime_cal: {deadtime_cal_devicespec, $
                        a:0.0, b:0.0}, $            ; deadtime cal parameters

pileup: {pileup_header_devicespec, $
                        on:0, file:'', found:0 }, $ ; pileup used and file-name
throttle: {throttle_header_devicespec, $
                        on:0, file:'', found:0 }, $ ; throttle used and file-name

sensitivity:            0.0, $                  ; if using indirect 'charge' via a counter,
                                                ; then charge = conv * flux * sensitivity
IC_name:                '', $                   ; name of IC channel/PV

px_coords:              ptr_new(), $            ; X coords array
py_coords:              ptr_new(), $            ; Y coords array
pz_coords:              ptr_new(), $            ; Z coords array
x_coord_units:          '', $                   ; X coord units
y_coord_units:          '', $                   ; Y coord units
z_coord_units:          '', $                   ; Z coord units
```

```
detector:    intarr(geopixe_max_adcs), $     ; detector_types **
cal:         replicate( {cal_devicespec, $
                  on:0, a:0.0, b:0.0, units:''}, geopixe_max_adcs), $
error:       0 } $                           ; error flag
old_mp:      ptr_new() $                     ; store old local device raw header data
```

** Detector types (starting at 0) in order: 'PIXE', 'PIGE', 'RBS', 'ERDA', 'STIM', 'CHARGE', 'DEADTIME', 'SXRF', 'SEM' (e.g. SXRF = 7).

Flux set-up

The “Sort EVT” window in GeoPIXE has a tab for setting up flux parameters. If an indirect flux/charge measurement is used, via an EPICS PV for example, then a button appears to scan the data for PVs, which are labels associated with ion chamber data used for flux measurement. This button calls the method ‘flux_scan’ in many devices to read in PV or other Tag strings that can be used to select the correct parameter for IC counter, and often PVs for preamp sensitivity. If sensitivity is not found, then some numerical sensitivity droplists are provided.

This panel sets the contents of the ‘flux_ic” variable that is passed into the ‘read_setup’ method. Typically, this sets the sensitivity of the IC count, often using the variable ‘nsls_flux_scale’ (= flux_ic.val * flux_ic.unit). Internally, the flux in a pixel becomes NSLS_IC, which is IC count times this factor (e.g. see NSLS_NetCDF_device__define.pro).

The flux_ic structure:

```
{ mode:0, pv:'', val:0.0, unit:0.0, conversion:1., use_dwell:0, dwell:1.0}
```

where	mode	0	no flux IC (ion beam, no IC data)
		1	use IC PV
		2	use conversion only (no PV)
	pv		user selected EPICS PV string to be used for flux IC
	val		pre-amp sensitivity value
	unit		pre-amp sensitivity unit (scale)
	conversion		conversion from flux count to charge (uC)
	use_dwell		use the dwell time with flux count-rate to build flux count per pixel
	dwell		dwell time in a pixel (ms)

Flux and Dead-time

When building images, zero ‘flux’ and ‘dead_fraction’ variables are passed to the imaging routine and on to the ‘read_setup’ and ‘read_buffer’ device object methods as arrays of the same size as the image area. These arrays need to be filled by the device, either:

1. Read in whole flux (and dead_fraction) arrays in ‘read_setup’ (e.g. APS and SLS are done this way).
2. Read in flux for each pixel as it is processed (e.g. in NSLS and Hasylab devices this sets variable NSLS_IC), which is then used to set flux array value for current pixel in ‘read_buffer’.
3. Typically, the data stream may supply an IC count. This is converted to flux count by scaling by the product “flux_ic.val * flux_ic.unit”, which comes from the flux PV selection pop-up window (the product is stored as common variable ‘nsls_flux_scale’ in many devices).
4. Dead-fraction may be determined for some devices from live and real time for each pixel (e.g. for ‘NSLS_MCA’ device in ‘read_setup’ for each pixel file and then assigned in ‘read_buffer’).

Maia detector system

The Maia design philosophy has been to integrate a large solid-angle detector array with event-by-event, real-time processing and stage control. It builds on the high throughput XAS detector concept of Siddons *et al.* and adds real-time full spectra data capture and an embedded real-time processor to orchestrate spectral acquisition and correction and real-time capture of stage position and incoming and transmitted beam flux signals locked to stage position. It uses an annular detector geometry, in which the beam passing through a hole in the centre of the array, to maximize solid-angle and to keep the sample plane completely free. This approach does increase the capture of scattering signals over a commensurate large solid-angle array located at 90° to the horizontally polarized synchrotron beam. However, in many applications the scattered signal is useful in its own right (e.g. inelastic scattering for imaging light elements in biological samples) and this geometry can accommodate both samples of any size and permit unimpeded sample movement for imaging. These points are explored further in the discussion.

Detector hardware description

The detector comprises 384 planar 1x1 mm^2 elements in an annular configuration developed in low-leakage, 0.4 mm thick silicon at BNL [De Geronimo 2003], mounted on an aluminium nitride frame and Peltier cooled to about −20 °C. This assembly and the analogue board are mounted on a water-cooled copper block, typically maintained at -5 °C. Beam passes through the detector via a molybdenum collimator which extends to a beryllium window on the front face of the detector. The tip of the Mo tube is tapered to avoid shadowing the inner detector elements. Further hardware details are provided by Kirkham *et al*. [Kirkham 2010] and Siddons *et al.* [Siddons 2006].

A molybdenum mask, designed to provide collimation of X-rays into the sensitive volume of each detector element, is glued to the front face of the detector wafer. Its purpose is to prevent charge generation in the vicinity of the gap between detector pads which may lead to charge-sharing between detectors aided by lateral diffusion of carriers. The mask comprises three layers, each 100 µm thick, with collimation apertures arranged to collimate X-rays originating from a source position 10 mm above the detector plane [Ryan 2010a] (Fig. mask).

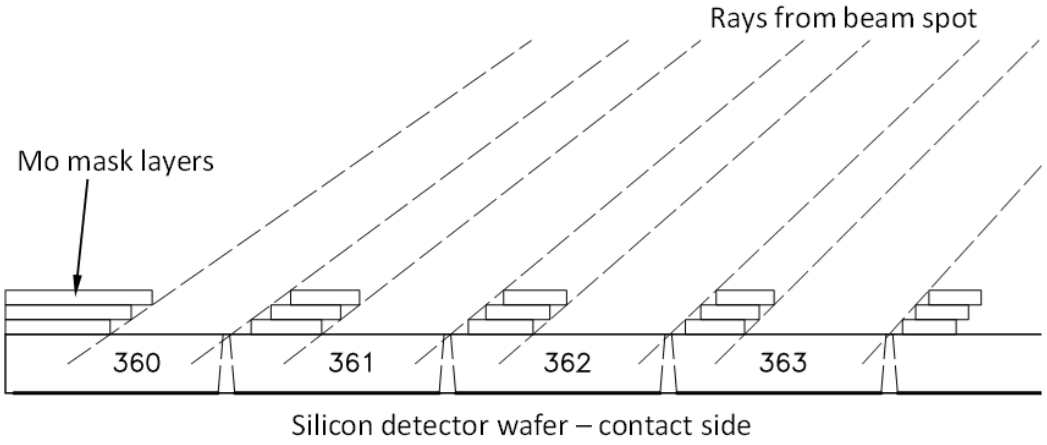


Fig. M1 Cross-section of detector array for 4 extreme off-axis detectors (see layout, Fig. M4) showing mask layers (3 x 100 µm Mo) that absorb X-rays that may otherwise deposit charge in the inter-pixel gaps to prevent charge-sharing between detectors. Rays from the beam-spot at the design distance of 10 mm from the detector wafer surface are indicated.

Pads on the detector wafer are directly wire-bonded to 12 CMOS application specific integrated circuits (ASICs) called HERMES, each providing 32 low-noise preamplifiers with high order quasi-Gaussian shapers and baseline stabilizers [De Geronimo 2003]. Shaped pulses go to 12 32-channel peak-detector derandomizer ASICs called SCEPTER, which capture pulse height, representing energy (E), and time-over-

threshold (T), which can be used for dead-time measurement and pile-up detection and rejection [Dragone 2005]. E and T analogue pulse outputs from SCEPTER are digitized by dual 14-bit fast synchronous ADCs, with 3 SCEPTERs multiplexed into each ADC pair. A field-programmable gate-array (FPGA) processor in the detector head packages event data for transmission out of the beam-line hutch over a dedicated 8 Gbit/s quad-channel fibre optic link to the processing subsystem, based on the parallel processing engine HYMOD, developed at CSIRO for high-speed instrumentation.

HYMOD consists of a 166 MHz Xilinx 4 Pro FPGA connected to 6 large static RAMs, a PowerPC 8555 processor and fast serial (12 x 3.125 Gb/s) and Ethernet ports (2 x 1Gb/s). FPGA code is written using a CSIRO-developed pipelined, parallel high-level processing language called 3PL [3pl], which generates a FPGA netlist for Xilinx place-and-route tools. The PowerPC handles network I/O, buffering for display, logging to disk and external control and parameter requests through a TCP socket. Additional parallel digital connections to the HYMOD FPGA provide pulse counting inputs and four channels of encoder readout. External switching under program control provides 3 banks of encoders.

Real-time processing pipeline

A new event can be accepted by the HYMOD pipeline on each clock cycle (Fig. M2). A clock frequency of 50 MHz permits event rates approaching this value. The processing pipeline comprises: (i) ±0.5 dither applied to E to reduce aliasing effects in subsequent stages, (ii) linearization correction, (iii) gain trim linear mapping to match both E and T across detector channels, (iv) energy calibration mapping, (v) pile-up rejection, (vi) ‘Groups Spectra’ accumulation, which provides real-time sum spectra of selected groups of detector channels, (vii) detector activity monitoring providing count rates across the array, (viii) dead-time and pileup monitoring across the array, (ix) E-T scatter plots, which help track pile-up behaviour, (x) real-time image accumulation, and optionally (xi) sub-sampling called ‘throttle’ to selectively reduce logged data rates to disk. Additional parallel pipelines (i) capture encoder position data, which are used to infer a real-time pixel address *x,y* for each event, and (ii) sample two pulse input channels FC0 and FC1, which are used to monitor flux on a pixel-by-pixel basis.

Each event *{n,E,T}* is tagged by detector number *n*. They are buffered in an ET event accumulator and output on pixel advance, buffer full or timeout. Each output ET record is tagged by *x,y* position, dwell time, FC0 and FC1 counts, total T for all events, total event count and total pileup rejection count. The ET records, and other accumulator records, are output over a dedicated 1 Gbit/s Ethernet connection to a file-server and logged to disk by the *blogd* logger.

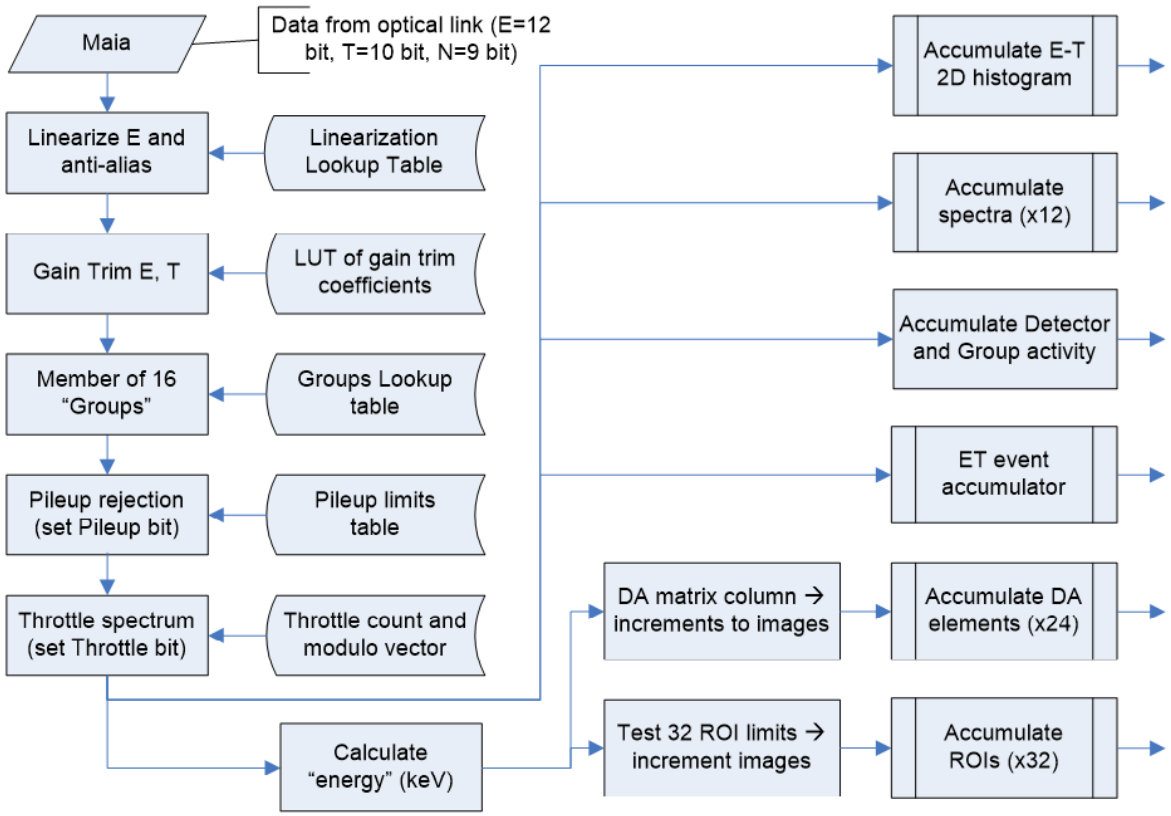


Fig. M2 Main event processing pipeline executed in the HYMOD FPGA. Corrected events are tagged with pileup and throttle bits and selectively accumulated for output over Ethernet.

Gain trimming

Gain trimming is designed to correct for the small gain and offset differences between detector channels. Once corrected, E and T spectra from all detectors should have similar gain and offset and can be overlaid and added together. This is necessary for simple application of the pileup rejection module and is essential for the Groups Spectra accumulator to provide a useful sum of selected detector spectra.

Pileup rejection

In the absence of pileup, the width of a quasi-Gaussian pulse in time at a fixed threshold T (time-over-threshold) follows a well-defined curve versus amplitude E. Coincident pileup falls on this curve at an energy corresponding to the sum of the pulses. Non-coincident pileup tends to display a longer T reflecting the delay between pulses at a lower E and can be discriminated [Dragone 2005] (see Fig. M3). Pileup rejection in Maia uses a table of low and high T limits for each E to define accepted events. The pileup module in the processing pipeline flags each event with a pileup status bit that can be used by subsequent modules to selectively reject pileup events.

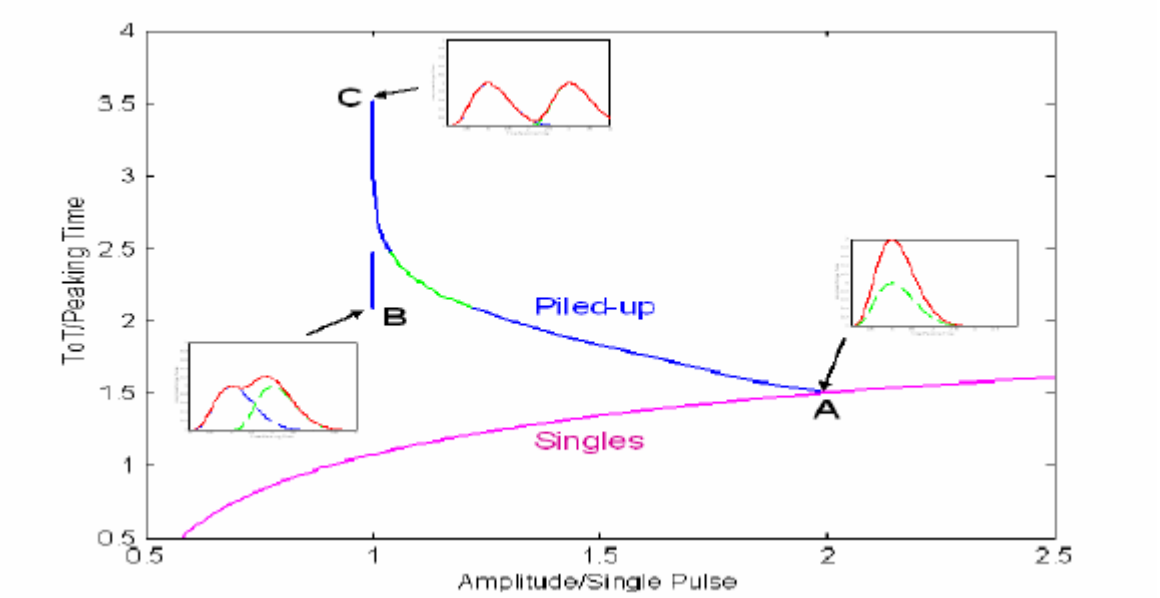


Fig. M3 E-T- trend for singles and pileup [Dragone 2005].

Throttling

Data rates are often dominated by a major element or the scatter peaks (e.g. for biological material). The data for these energies is often excessive and superfluous. It can be reduced prior to storage in the blog files without loss of significant image detail from these spectral features using the 'Throttle' mechanism. The 'throttle' module in the Maia FPGA pipeline uses a recycling vector to count all events versus E and compares the count to a 'throttle modulo' vector. Until the modulo count is reached the event is flagged with a set 'throttle' status bit that can be used by subsequent modules to selectively reject these events. When the modulo count is reached the throttle status bit is cleared. Commonly, the ET event accumulator discards events with 'throttle' set to reduce subsequent disk storage. By default, the modulo values are unity to accept all events. The 'throttle modulo' vector can accept values up to 15 for specific E values to accept only 1 in 15 events. Generally, the real-time imaging module ignores 'throttle' as it can easily manage all detector events.

Real-time imaging

Real-time imaging uses the *Dynamic Analysis* (DA) method, which is outlined below. Implementation of DA in Maia uses a lookup table in the FPGA to hold the DA matrix $\boldsymbol{\Gamma}$. Each event *{n,E,T;x,y}* is mapped onto the appropriate column in $\boldsymbol{\Gamma}$ using the linear energy calibration for each detector channel *n*. All members of this column are applied as increments to a DA accumulator vector, which gathers event contributions to each element image (and background, Compton and elastic scattering) at the current pixel '*x,y*'. When '*x,y*' steps out of the current pixel, this accumulator is read-out as a DA record over Ethernet to the *blogd* server and the accumulator is cleared. Each DA record also includes (i) dwell or transit time in that pixel, (ii) accumulated counts in the FC0 and FC1 counters, (iii) total event count, (iv) pileup lost count, and the (v) total time-over-threshold. These enable correction in down-stream processing for flux variation across an image and pileup and dead-time losses. The *Maia Control* GUI process listens for DA records and displays the DA pixel data as real-time element images.

Beamline integration and sample scanning

Maia control software is configured so that Maia functions as a slave to the beamline control system, which generally orchestrates sample scanning. Critical aspects are taken care of directly by Maia, to avoid latency issues, such as stage position monitoring and beam flux integration per pixel. Flux integration uses a voltage-to-frequency converter on each ion chamber preamplifier output and counting of pulses using the FC0 and FC1 inputs to Maia. At the XFM beamline, parameters for a scan are set via EPICS using a beamline user interface, which also initiates data collection and logs scan details to the *blogd* server. Maia listens for the

scan parameters to capture details needed for the display of real-time images. This flexible approach can be nested within other scan loops to embed the acquisition of an image within a XANES energy scan or a tomography axis rotation to yield 3D data-sets. The connections to Maia/Hymod use a simple socket interface (at XFM the socket is handled via a small EPICS IOC), which also logs slowly varying parameters of the beamline and detector to the *blogd* data-stream (such as photon beam energy and detector temperature and leakage current).

Maia imaging method

The method developed for Maia and other detector arrays, and implemented in the GeoPIXE software, strives for practical solutions to four aspects of elemental imaging: (i) spectral deconvolution of overlapping components aiming to faithfully portray their spatial distribution, (ii) accumulation of images in concentration-fluence units to allow direct interrogation in concentration units, even online during data collection, (iii) tools for verification to allow the accuracy of images to be probed and refined (for example the identification and addition of a missing element) and (iv) high speed processing to enable extension to high count-rate detector arrays such as Maia, scanning with large addressable pixel areas and real-time application. It is based on the calculation of model fluorescence X-ray yields relating detected counts to concentration using a fundamental parameters approach and casting the least-squares fitting problem as a matrix transformation of an X-ray spectrum (or its constituent events) that enables event by event accumulation of concentration images.

Fundamental parameter approach

The method represents an evolution from a simple fundamental parameters approach that assumes a small solid-angle detector where take-off angle variation across the detector is negligible and 'generic' theoretical yields can be pre-calculated into a library based on sample composition and structure and beam energy for an ideal, unit solid-angle detector and combined with specific experimental details for detector solid-angle, intrinsic efficiency and external absorbers [Ryan 1990, 2005b]. Detected counts N_k for element k are related to element concentration C_k by the equation

$$N_k = Q\Omega\varepsilon_k T_k Y_k C_k \qquad (1)$$

in terms of the integrated beam fluence Q, detector solid-angle Ω and intrinsic efficiency ε_k, X-ray absorber attenuation T_k and 'generic' X-ray yield Y_k . It has been extended to handle a large solid-angle detector array, such as Maia, through the definition of 'relative sensitivity factors' ψ_{kn} for each detector n relative to a 'central' detector element ("0" subscript), which account for the effects of variation of take-off angle, absorption path, solid-angle and intrinsic efficiency for detector elements across an array [Ryan 2010a]:

$$\psi_{kn} = \frac{\left(T_{kn}\Omega_n\varepsilon_{kn}Y_{kn}\right)}{\left(T_{k0}\Omega_0\varepsilon_{k0}Y_{k0}\right)} = \frac{N_{kn}}{N_{k0}} \qquad (2)$$

Model yield calculation

Thick target yields of fluorescence X-ray lines are integrated through a layered sample structure using the approach of Reuter *et al*. [Reuter 1975], including secondary fluorescence contributions, and using the sub-shell absorption cross-sections of Ebel *et al*. [Ebel 2003], and the Coster-Kronig transition rates for the L shell, fluorescence yields and branching ratios from the fundamental parameter database of Elam *et al*. [Elam 2002] with refinements recommended by Campbell [Campbell 2003]. Missing branching ratios for weak transitions are taken from the GeoPIXE database, which is based on PIXE measurements of pure elements and simple compounds for 52 elements [Ryan 1990] and from direct SXRF measurements using Micromatter foils (~40 μg/cm^2 foils for 60 elements on Mylar supports) [Ryan 2005b, Micromatter].

Yields are calculated in two passes: (i) calculate yields for the 'central' detector, including integration of secondary fluorescence enhancement, integrated through the layered sample, and (ii) calculate yields for all other detector array elements, applying the secondary fluorescence enhancement from the 'central' detector. For each element, the ratio of the yield for each detector to the 'central' detector value (Y_{kn}/Y_{k0}) contributes to the 'relative sensitivity' factors for each detector.

Yields are calculated for all X-ray lines individually so that the derived relative line intensities reflect the integration through the sample layered structure, detector intrinsic efficiency, solid-angle and absorption effects (e.g. external filters and detector windows). Finally, the ratios of line relative intensities for each detector relative to the central detector are calculated too; these are drawn on for spectrum fitting to derive line relative intensities based on a weighted average of the sub-set of detector elements selected for a particular analysis.

Detector model

In the GeoPIXE detector model, members of an array are arranged in a grid using local coordinates *u* and *v*. By default, the detectors are assumed to fall in a plane. An optional *w* parameter can place them out of the plane and a local tilt angle can be applied to tilt them towards or away from the array centre. The model can cater for planar and hemispherical arrays. The detector array as a whole is positioned at a distance *d* from the beam-spot and at angles ϑ about the vertical axis and φ about the beam axis. Normally, the array is assumed to point directly at the beam-spot. Optionally, global tilts angles δ and η about the *v* and *u* axes, respectively, can be used if this is not the case. The yield calculation uses these parameters in order to locate each detector in space and to determine the take-off angle at the sample surface, the distance *R* to each detector and an effective tilt angle τ, the angle between each detector normal and a ray from the beam-spot. The distance and effective tilt influence the effective solid-angle (small area approximation) of each detector. The tilt influences the path length through the detector material, and hence its intrinsic efficiency ε.

$$\Omega_n = area_n . \cos(\tau_n) / R_n^2 \qquad (3)$$

$$\varepsilon_n = 1 - e^{-\mu T / \cos(\tau_n)} \qquad (4)$$

where μ is the mass attenuation coefficient and *T* the thickness (expressed as areal density) for the planar detector material. The detector model also includes corrections for finite source size and escape losses [Ryan 1990].

In the case of Maia, $area_n = width_n * height_n$ in terms of the effective $width_n$ and $height_n$ of each detector, which are set by the rectangular apertures cut in the 3 layer Mo mask, each 100 µm thick (Fig. M4). The position and size of the apertures in each layer are arranged so that they align with rays from the beam spot positioned at a distance of 10 mm from the detector wafer (Fig. M1). The mask is treated as opaque. The slight transparency of the edges of the mask at energies in the range 15-20 keV has not been treated at this time, which may introduce errors of ~3% for extreme off-axis detector positions for element lines in this range.

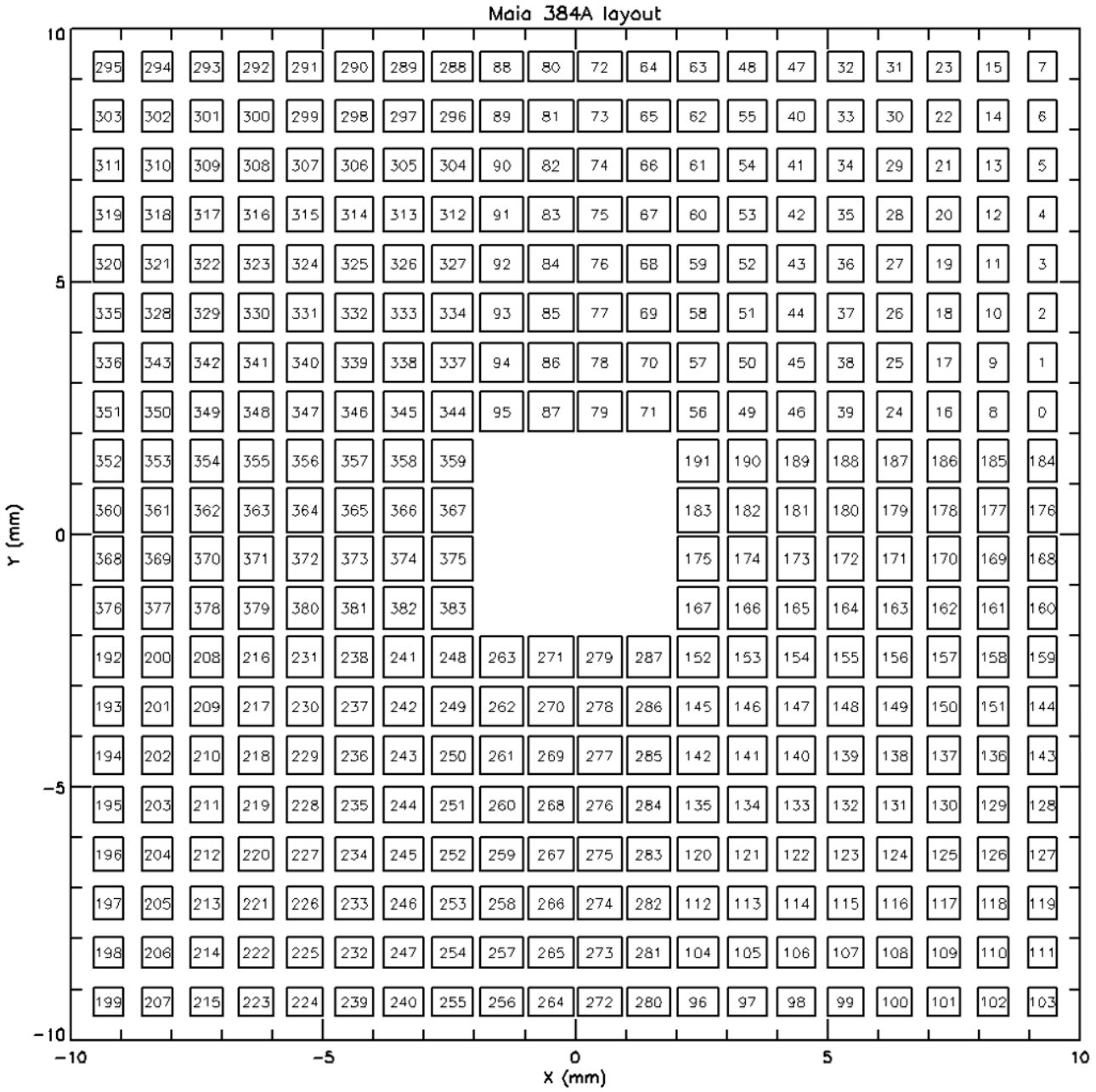


Fig. M4 Layout of the Maia 384A detector array showing detector index numbers. The effective size of each detector is shown, which reflects the collimation by the apertures in the 3 layer mask.

Spectrum analysis

Spectrum analysis uses a non-linear least-squares fitting method [Bevington 1969, Ryan 1990] based on model functions for each element built on Gaussian line shapes with exponential tails that use line relative intensities appropriate for the sub-set of detector elements that contribute to a particular spectrum. These are based on the 'central' detector relative intensity values corrected for the average model ratios of line relative intensities to the 'central' detector, weighted by the 'relative sensitivity factors', for all contributing detector elements. The model function includes up to 16 lines per element (K,L,M lines treated separately) plus escape peaks for each based on a two-sided planar escape model [Ryan 1990]. Non-linear parameters control energy calibration, peak width variation and peak tailing. Continuum background is approximated using the SNIP iterative peak clipping algorithm [Ryan 1988], optionally split in two and combined with the Bauer model [Bauer 1995], and the Compton scattering peak distribution uses the peak-shape parameterisations of Van Gysel *et al*. [Gysel 2003].

A pileup "sum" element is used to fit pileup contributions in the spectrum, which assumes that some form of pileup rejection is used and only the pileup peaks representing close to coincident pileup remain. The "sum" line relative intensities are calculated from the double and triple products of other spectral lines present in the spectrum to provide a pileup model to third order. Close lines (within 60 eV) are combined and up to 50 lines are used to describe pileup contributions with one free parameter representing effective pulse-pair timing resolution [Ryan 1990]. In the case of spectra extracted from an imaging data-set, which may represent a mixture of areas with very different major element content and pileup line intensities, this simple model may fail. In these cases, a refined pileup model is available in GeoPIXE that makes use of the product of images to better track such spatial variation in pileup content [Ryan 2005a].

Total beam fluence associated with the spectrum is expressed as 'charge' in μC units (6.242 10^{12} photons) and it is assumed to be live charge, corrected for dead-time and identified non-coincident pileup losses. The coincident pileup loss fraction, which represents intensity lost from peaks in the spectrum and deposited in the "sum" peaks, is inferred from the fit to the "sum" element, and is given by the ratio of the intensity of the largest pileup peak to its parent, the most intense peak in the spectrum. Triple pileup is lower in intensity relative to double pileup by the same factor [Ryan 1990].

Dynamic Analysis imaging

The *Dynamic Analysis* (DA) method is based on the expression of a linear least-squares fit to an X-ray fluorescence spectrum as the solution of a set of simultaneous equations, written in matrix form, and re-arranged into the form of a matrix transformation of the spectrum to elemental concentration vector [Ryan 1993, 2000]. This transformation can be applied to each event in turn, which lends itself to event-by-event processing and real-time implementation [Ryan 1995].

The DA transform

The non-linear parameters are determined in a full non-linear least squares fit to a 'master spectrum', a representative spectrum acquired under the same experimental conditions as the image data to be analyzed; often this is the total sum spectrum for all detectors from the image run or a precursor. This fixes the energy calibration and the form of the line-shape function. The remaining linear parameters a_k, representing major-line peak-areas and one representing background intensity, can be found using a linear least squares iteration that can be cast as a matrix transformation, which transforms directly from spectrum vector $\boldsymbol{S}$ to concentration vector $\boldsymbol{C}$ in terms of the matrix Γ. For the 'central' (n=0) detector, this takes the form (for details see [Ryan 2000, 2010a]):

$$\boldsymbol{C} = Q^{-1}\,\Gamma\boldsymbol{S} \qquad (5)$$

$$\Gamma_{ki} = \left(\Omega_0 \varepsilon_{k0} T_{k0} Y_{k0}\right)^{-1} . \sum_j \left[\alpha^{-1}\right]_{kj} \beta_{ji} \qquad (6)$$

in terms of the integrated beam fluence Q, detector solid-angle Ω and efficiency ε_k, X-ray absorber attenuation T_k, the α matrix and β vector, which are built from line shape functions [Ryan 2010a] and X-ray yields Y_k; initially the Y_k are assumed to be constant for each element k across the entire image area. For an array of detectors, the Γ_{ki} must be scaled by factors $(f_k)^{-1}$, where f_k is the sum of the 'relative sensitivity factors' ψ_{kn} for all detectors n used to build the images [Ryan 2010a]. This represents a generalization of a multiplicity correction. In practice these factors are applied to the images after processing. Generation of the DA matrix in GeoPIXE also outputs the unit 'pure element spectra' for each component and values used for minimum detection limit estimation for quantities extracted from images.

Image construction

In regard to Maia real-time imaging, each detected event in spectrum bin e selects a column Γ_{ke} from the right side of equation 5. This suggests a convenient strategy to accumulate elemental images M_k in concentration-fluence units by simply incrementing each image k for each event (at x,y) by Γ_{ke} and image

variance V_k by Γ_{ke}^2. This is the lookup table operation referred to above in the Maia processing pipeline. These images can be directly interrogated for average concentrations $\langle C_k \rangle$ in a region (and their uncertainties $\langle \delta C_k \rangle$), both on-line during data collection or off-line, by reference to the fluence distribution $Q(x,y)$ [Ryan 2001]:

$$\langle C_k \rangle = \frac{\left(\sum_{region} M_k(x,y)\right)}{\left(\sum_{region} Q(x,y)\right)} \qquad (7)$$

$$\langle \delta C_k \rangle = \frac{\left(\sum_{region} V_k(x,y)\right)^{1/2}}{\left(\sum_{region} Q(x,y)\right)} \qquad (8)$$

Hence, within the approximation of constant Y_k across the image area, these images are quantitative and can be directly interrogated for elemental concentrations even while data are collecting.

Implementation of DA in Maia

Real-time implementation of DA in Maia was described earlier and uses a lookup table in the FPGA to hold the DA matrix $\boldsymbol{\Gamma}$ determined for the 'central' detector and accumulates contributions of each event to each element. These accumulators are read-out as a DA record to the *blogd* server for each pixel, which is used by the *Maia Control* GUI process to display the DA pixel data as real-time element images.

Off-line processing of Maia event records in GeoPIXE operates in a similar fashion, using energy calibration mapping and a DA matrix lookup table. Event contributions are incremented directly into concentration and variance image arrays. Maps also accumulate pileup losses, dead-time, dwell values and contributions to fluence per pixel. Following event processing, the element images are scaled by the $(f_k)^{-1}$ multiplicity correction (and variance by $(f_k)^{-2}$). The fluence map is corrected for pileup and dead-time losses to produce a "live" fluence map and normalized (or "flattened") to a uniform value. The same "flattening" is applied to the element images and counter maps.

Image corrections

Correction of images for the variation of yields due to sample composition spatial variation can be performed as a post image construction correction. It is based on a linear decomposition and matrix transformation into end-member proportion images. These end-member proportions enable the folding of yields derived for the end-members to estimate the yield for the mixture in each pixel; the ratios of the original yields to these mixture yields provides the pixel-by-pixel image corrections. It is an iterative procedure that converges quickly [Ryan 2000]. Edge effects arising from differential absorption in complex images can often be corrected using a similar approach [Ryan 2002]. The GeoPIXE method also has tools for correction of image shifts due to backlash between raster scan lines and between full image frames in a XANES image stack.

Verification of images

The procedure used to build the DA matrix from the 'master' spectrum may miss a weak element, perhaps concentrated into just a few pixels in the image area. These elements need to be discovered and added to the DA matrix. Their absence may distort the images of overlapping elements. Alternatively, with minimal overlaps, these elements will contribute to the background intensity map. Hence, a routine aspect of the method is to probe the spectra of any conspicuous features in the background and element images and fit these spectra to search for missing elements.

GeoPIXE provides interactive tools to select 'regions' of image pixels. Regions are defined in terms of bounding shapes or via element-element associations or principal component analysis (PCA). Shapes include

simple shapes and closed spline curves with varying numbers of control points. The element-element associations approach uses a closed spline curve field defined on a scatter plot of element 1 versus element 2 for all pixel data to select all pixels with concentration values within the spline field. A similar selection process can be used on principal component axes. For each region, the integral from each image, divided by total fluence for the region, provides an estimate of the average concentration of each element.

Spectra are extracted from a set of regions by re-processing the *blogd* event stream. Normally, all detector elements are combined, re-mapped onto a common energy calibration into a single spectrum per region. The display of these extracted spectra are overlaid in GeoPIXE by the 'DA fit', which is a spectrum composed of all 'pure element spectra' weighted by the product of 'charge', multiplicity factor f_k and region average concentrations for each element. This 'fit' is identical to a linear least-squares fit to the enclosed pixel spectra. Often missing elements are immediately apparent as deficiencies in the 'DA fit'. Further investigation and fitting may be needed to flush out other missing terms. These missing elements are added to the fit to the 'master' spectrum to generate a new DA matrix in order to regenerate the images (see detailed example in [Ryan 2010b]).

Image exploration

The DA images are constructed in ppm.µC units and are displayed in concentration units. Image 'regions' provide a way to explore average concentration in that selection of pixels. Similarly, the associated variance image region provides statistical uncertainty estimates. In addition to the simple shapes and spline curves mentioned above, linear and curvilinear traverse profiles can be extracted from the image data (together with uncertainty error bars) directly from the image data (and associated variance images). Each pixel within the width of a traverse region is mapped back onto the central axis of the traverse to define its distance along the traverse. Each point along the traverse is normalized to account for the number of pixels that map onto it. This is reflected in the detection limit estimates for each step along the traverse, which are also displayed.

Element-element associations are also used to explore element correlations in the data. These are accumulated from a scatter plot of two selected elements, with either linear or log concentration axes, and displayed as a colour 2D histogram. This approach does not provide a complete cluster analysis of all element data. Nevertheless, these plots provide a useful tool to delineate phases and processes in the samples, or as a shortcut to selecting pixels from certain structures with common compositional signatures. It can be used to threshold an image. Pre-filtering of each element image can be selected prior to accumulating the 2D histogram to reducing noise effects in weak trace element data, and the colour Z axis can be displayed in linear or square-root mapping to compress dynamic range.

RGB composite images, generated in 24bit colour using the concentrations of three selected elements to set the Red, Green and Blue colour weights, provide a simple way to look at the co-localization and contrasts in position of the selected elements. GeoPIXE allows these colour scales to map onto the element concentration with a linear relationship, as well as square-root and logarithmic mappings to provide dynamic range compression to allow both high and trace concentration ranges to coexist within the limited colour axis intensity range. Controls on the minimum and maximum concentrations within the available range of each axis are used interactively to bring out the features of interest.

Set-up and performance

Most aspects of the real-time processing pipeline relate to event correction and the real-time display of E and T spectra. Set-up of these are outlined below. Set-up steps required to prepare for real-time imaging using the DA method and also to filter events using the Throttle mechanism is user sample dependent and is done when required by users during an experiment. Calibration refers to the determination of key parameters of the beamline-detector system model. The following summarises the procedures used for initial set-up and calibration needed for detector operation and real-time imaging.

Real-time pipeline set-up

Linearization and gain trimming

Non-linearity of the detector response with energy is probed and characterized using the Maia built-in test pulser and the "step-cal" procedure, which produces a series of pulser peaks in selected detector channels at equally spaces intervals across the full energy range. The linearization analysis and correction is aimed at refining a correction function that removes shifts in centroid position so that these peaks have the same separation across the spectrum. Two correction functions are fitted to the centroid shifts in turn. The first seeks to correct any low energy non-linearity and takes the form: $Y = X + A_0.A_1.exp(-A_1X)$, where X is the correct centroid value, Y is the observed centroid and the A_i are the fitting parameters. The second, fitted to any residual from the first, takes the form of a 5th order polynomial: $Y = X + A_0 + A_1(X-c) + A_2(x-c)^2 + \ldots + A_5(x-c)^5$ in terms of the fitting parameters A_i and the centre of the spectrum c. For Maia 384 revision A, the dominant term is a cubic component of ~0.5%. The final linearization table is uploaded into the 'linearization' table in the Maia FPGA including 6 fractional bits.

Differences in gain and offset across detector channels are characterized using the built-in test pulser and the "gain-trim" procedure, which produces a pair of pulser peaks in both E and T in all detector channels in turn. The method uses a pair of peaks, at both ends of the E and T range, and determines a linear mapping for E and T for each detector channel to match the centroids of these peaks to the average centroids of E and T for all channels. The complete table of linear coefficients for gain and offset for E and T for all channels is uploaded into the 'gain-trim' table in the Maia FPGA.

Pileup rejection

Pileup rejection in Maia uses a table of low and high T limits for each E to define accepted events (Fig. M3). These are defined in *GeoPIXE* or *Maia Control* using spline curves placed on a T versus E plot of data from Maia once linearization and gain-trimming have been set-up and enabled. These are uploaded into the Maia FPGA as the 'pileup-limit' table.

'Throttle' - data sub-sampling

Maia Control and *GeoPIXE* have tools for crafting a 'throttle' modulo vector based on sample spectral data. This procedure is sample spectrum dependent and typically targets sub-sampling of intense major element lines or scatter peaks. A new 'throttle' modulo vector can be uploaded by a user into the Maia FPGA 'throttle' table to selectively reduce data-rates for these common events. The presence of active 'throttle' is logged into the blogd data-stream.

DA imaging

Set-up for quantitative element imaging in Maia involves these steps: (i) construction of the DA matrix and its upload into the Maia FPGA as a lookup table, (ii) upload of individual detector channel energy calibrations, (iii) loading the 'relative sensitivity factors' into the *Maia Control* GUI for normalization of extracted concentration quantities linked to the enabled detector channels, (iv) loading calibration of the time-over-threshold scale for dead-time measurement, and (v) loading ion chamber preamplifier sensitivity details relating to the upstream ion-chamber used for flux monitoring. A scan across a selection of Micromatter foils [Micromatter] (e.g. Ti, Ni, Pt, YF_3, acquired at 18.5 keV), is used to produce spectra for each detector channel with lines that span ~4 to ~15 keV. These are fitted in *GeoPIXE* to determine linear energy calibration parameters per channel. These are uploaded into the Maia FPGA along with a DA matrix, to map event 'E' onto matrix column for real-time imaging. These steps are handled by the *Maia Control* GUI.

System calibration

In order to perform quantitative analysis, certain aspects of the detector and absorber model need to be constrained, such as (i) calibration of the dead-time scale, (ii) refinement of the thickness of absorbers within the detector model, (iii) refinement of external 'filter' absorber thickness, and (iv) determination of a

calibration factor relating ion chamber accumulated counts to beam fluence. Calibration of the dead-time scale, relating time-over-threshold (T) values to real-time for each event, uses a series of images run at a wide range of detector count-rates on a uniform thin metal foil to induce measurable dead-time losses. This is illustrated in the performance tests section below. Dispersion of this calibration across SCEPTER channels of the array has not been detected within the accuracy of this approach.

Detector efficiency and filters

The thickness of various absorbing layers in the detector model and external absorbers or 'filters' are constrained empirically to yield consistent analyses of reference foils containing elements that span the energy range of the detector. All layers may not be determined independently. 'Filters' that can be added and removed can be determined independently by fitting spectra containing both high and low energy lines, acquired with and without the filter of interest, using the method of Ryan *et al*. [Ryan 1990]. The high energy line suffers little absorption, while the low energy lines effectively sample filter thickness. A double ratio of low to high energy line counts, with and without the filter in place, provides a direct measure of effective filter thickness independent of other calibration issues (e.g. sample composition, detector efficiency and flux measurement).

Beam fluence

Once the consistency of quantitative analysis over the detector energy range has been established, routine application still requires the calibration of the accumulated beam fluence in the focussed beam spot in terms of ion chamber counts. Calibration is accomplished using one or more reference foils. The constraint to reproduce the concentration of the main element in the foil is used to provide the ratio of fluence to ion chamber count, which is referred to as the 'conversion factor', which is used in subsequent analyses using the same beam energy and under the same conditions. If conditions change (e.g. KB refocussing, slit or energy changes), then a new foil calibration is performed and the 'conversion factor' revised.

Performance tests

Once the system is set-up and calibrated, we need some confidence that it will still produce quantitative results as the total count-rate changes and if various sub-sets of detector elements are selected for a particular application. For example, contrasting images processed separately using inner and outer detector elements, which correspond to contrasting take-off angle at the sample surface and register varying degrees of self-absorption on outgoing fluorescence X-rays, provide a direct depth contrast. Hence, we need to test consistency in quantitative results as a function of detector count-rate, individual detector selection and as a function of energy.

Dead-time and pileup correction with count-rate

Scans over a copper Micromatter foil were collected at a range of count-rates using a 10 keV X-ray beam at XFM to probe dead-time and pileup behaviour and correction. Flux on target was varied to produce count-rates per detector ranging from ~300 to ~30,000, which corresponds to total detector array rates of 0.1 to 11 M/s. To avoid issues relating to systematic errors between scales, the same ion chamber pre-amplifier gain scale was used for all count-rate runs. This made the measurement sensitive to ion chamber dark current, which needed to be accounted for; normally for applications, the appropriate gain scale for a particular flux would be selected, which obviates the need for a dark current correction. Dark current was read from the EPICS data, logged regularly in the blog list-mode files, at the start of each run prior to opening the shutter, using *blog_browse* and assumed to remain a constant offset through each run. The pileup loss fraction was monitored directly by counting the fraction of events rejected by the pileup field (Fig. M5). A rather tight pileup field was used, which tended to reject ~10% of events even at low rates.

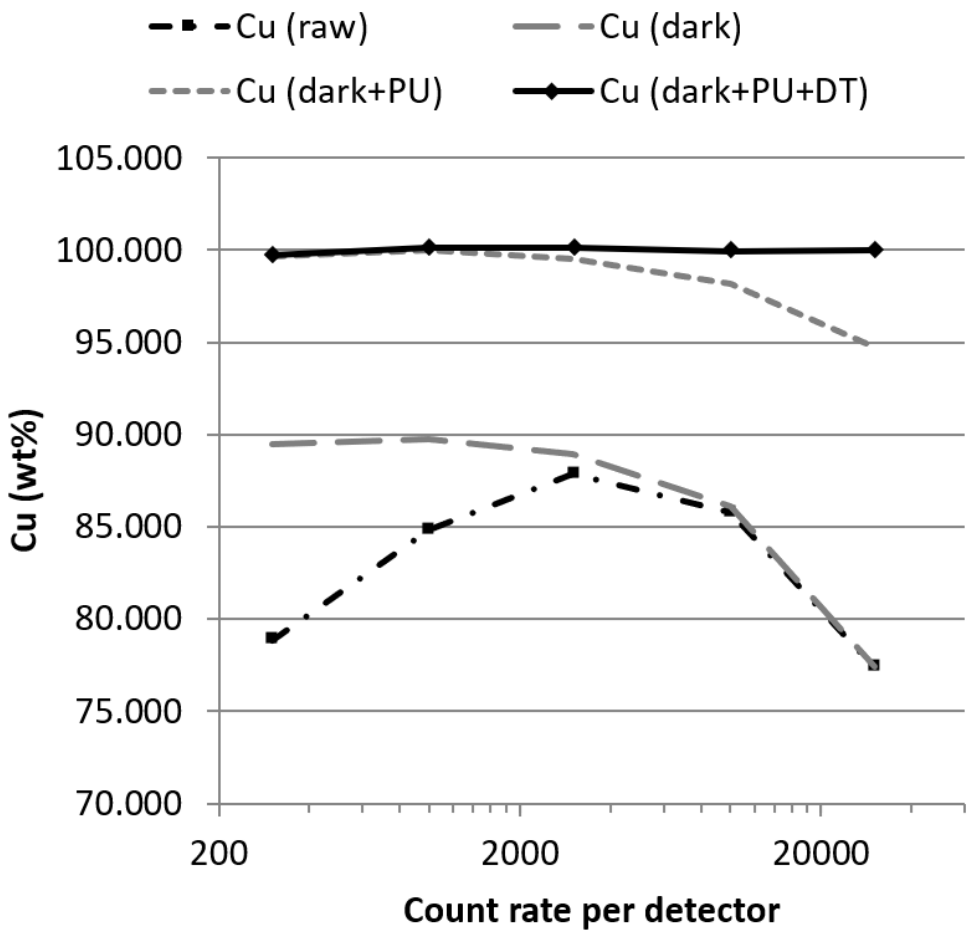


Fig. M5 Cu concentration inferred from images collected on a Micromatter Cu foil (49.2 µg/cm^2) for a range of count rates showing the sequential corrections applied for dark current (dark), pileup rejection (PU), and dead-time correction (DT). The final Cu values show a standard deviation of 0.17%.

The remaining term is dead-time losses, which requires a calibration of the time-scale of event T values. This was done empirically to constrain deduced copper concentrations in the foil to constant values towards high count-rates (Fig. M5). The result was found to be 16.9 ns per T ADC step. The derived Cu concentrations in the foil have a standard deviation of just 0.17% for count rates from 300 to 30,000 per detector, or up to 11 M/s for the array. Repetition of this series of images over a 2 day period showed reproducibility standard deviation of 0.7%.

Relative sensitivity factors

Model versus measured 'relative sensitivity' factors ψ_{kn} were compared for a member of the Cu foil count-rate series. The model values were based on the Maia detector model, with individual solid-angle terms constrained by the Mo mask apertures, and yields calculated for a 49.2 µg/cm^2 Cu layer excited with a 10 keV X-rays. The measured ψ_{kn} were determined from the ratio of the Cu counts recorded by each detector relative to the average of the central 8 detector elements. The model values were similarly re-normalized to the same 8 detectors. The model and measured values agree well, with a standard deviation on the measured to model ratio of 3.1% (see Fig. M6). Some systematic trends in these ratios can be compensated by applying a global tilt to the array of δ=1.3° about v and η=-0.4° about u, probably arising from Maia gantry positioning errors and the deflection of the beam from the KB focussing mirrors. This reduces the standard deviation to 2.8%.

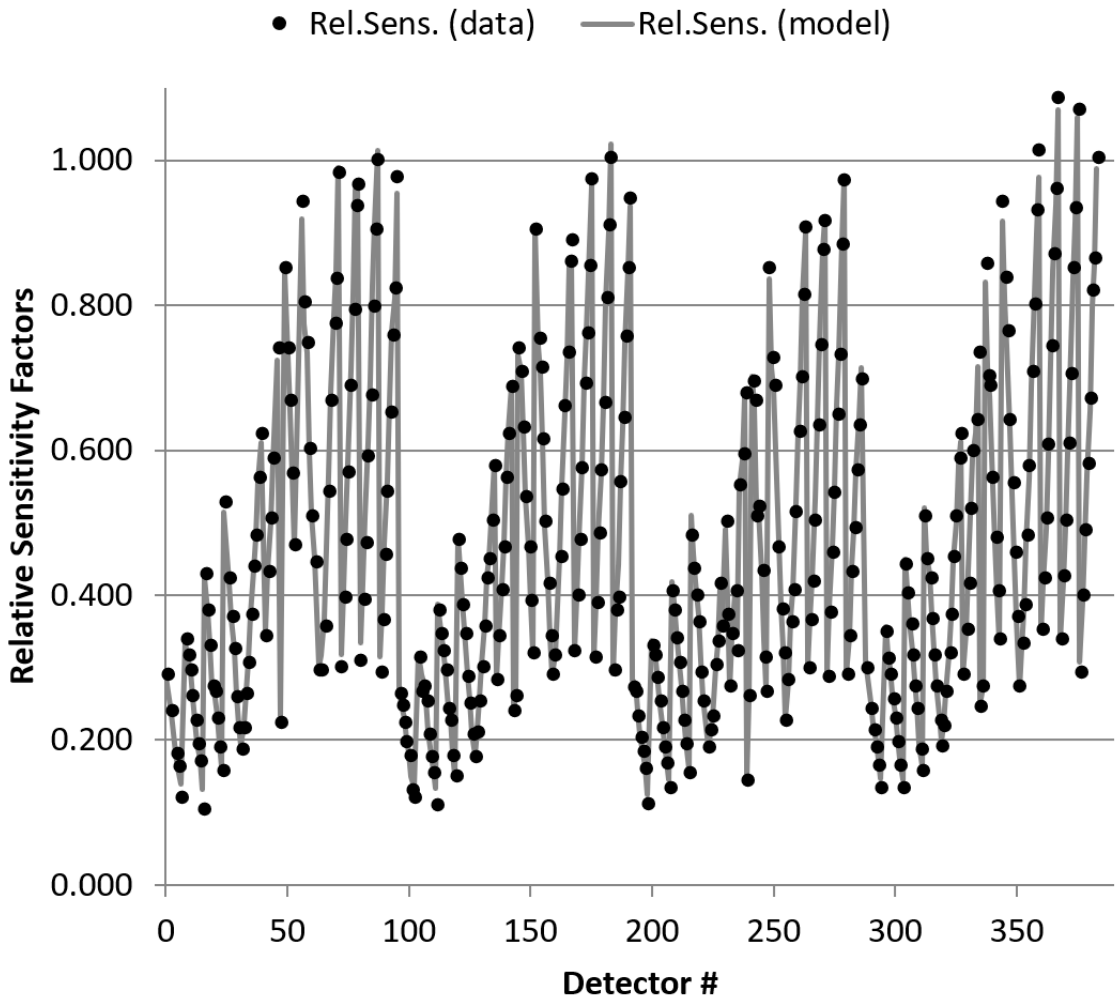


Fig. M6 Relative sensitivity factors per detector channel calculated from the model ("model", grey lines), with a global tilt of δ=1.3° about v and η=-0.4° about u applied, versus those inferred from the Cu foil data ("data", solid dots). Std. dev. of data to model ratio is 2.8%.

Detector efficiency

The product of intrinsic detector efficiency and the absorption contributions from external filters was measured using a set of Micromatter XRF standard foils excited at a range of beam energies (10, 18.5, 23.5 keV; Fig. M7). The external filter in this case is just the air path between the sample and the 100 µm Be entrance window to Maia at a distance of 5 mm normal to the sample surface. The thickness of the window was assumed in the detector model. Any errors in the window thickness will contribute to the effective air path thickness. The results, shown in Fig. M7 plotted against the energy of the alpha line for each element, are consistent with a 5.0 mm air path and 300 µm detector thickness.

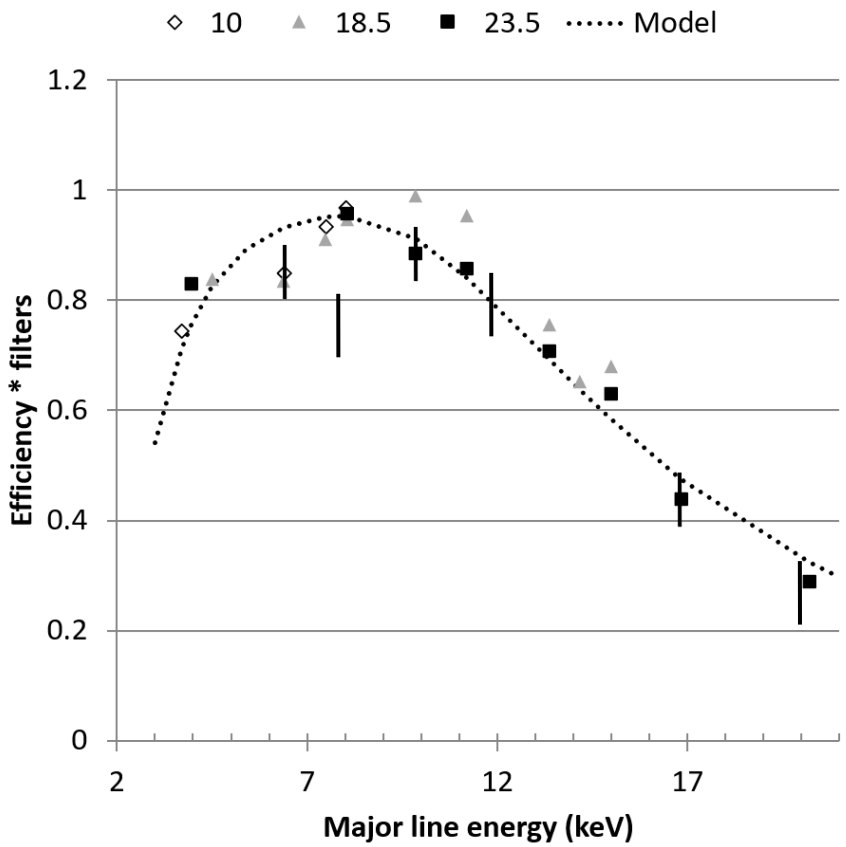


*Fig. M7 Detector efficiency * filter absorption inferred from using data from Micromatter XRF standard foils (5% uncertainty shown as representative error bars) using the beam energies: 10, 18.5 and 23.5 keV, shown against the model using 5.0 mm air path and 300 µm detector thickness.*

References

H.E. Bauer (1995), "A fast and simple method for background removal in Auger electron spectroscopy", ***Fresenius J Anal Chem*** 353, 450-455.

P.R. Bevington, (1969), "Data reduction and error analysis for the physical sciences", McGraw-Hill, New York.

J.L. Campbell, (2003), ***Atomic Data and Nuclear Data Tables*** 85, 291–315.

A. Dragone, G. De Geronimo, J. Fried, A. Kandasamy, P. O'Connor and E. Vernon, (2005), ***IEEE Nuclear Science Symposium Conference Record*** 2, 914.

H. Ebel, R. Svagera, M.F. Ebel, A. Shaltout and J.H. Hubbell, (2003), ***X-Ray Spectrometry*** 32, 442–451.

W.T. Elam, B.D. Ravel, J.R. Sieber, (2002), ***Radiation Physics and Chemistry*** 63, 121–128.

G. De Geronimo, P. O'Connor, R.H. Beuttenmuller, Z. Li, A. J. Kuczewski, and D. P. Siddons, (2003), ***IEEE Trans. Nucl. Sci.*** 50, 885-891.

M. van Gysel, P. Lemberge, P. van Espen, (2003), ***X-ray Spectrometry*** 32, 139.

R. Kirkham, P.A. Dunn, A. Kucziewski, D.P. Siddons, R. Dodanwela, G. Moorhead, C.G. Ryan, G. De Geronimo, R. Beuttenmuller, D. Pinelli, M. Pfeffer, P. Davey, M. Jensen, D. Paterson, M.D. de Jonge, M. Kusel and J. McKinlay, (2010), "The Maia Spectroscopy Detector System: Engineering For Integrated Pulse

Capture, Low-Latency Scanning And Real-Time Processing", ***AIP Conference series***, 1234, 240-243.

"Micromatter XRF calibration standards", MICROMATTER 4004 Wesbrook Mall, Vancouver, British Columbia, V6T 2A3, Canada (***www.micromatter.com***)

W. Reuter, A. Luria, F. Cardone, and J.F. Ziegler, (1975), "Quantitative analysis of complex targets by proton-induced X-rays" ***J. App. Phys***. 46, 3194-3202.

C.G. Ryan, E. Clayton, W.L. Griffin, S.H. Sie and D.R. Cousens, (1988), ***Nucl. Instr. Meth***. B34, 396-402.

C.G. Ryan, D.R. Cousens, S.H. Sie, W.L. Griffin, G.F. Suter and E. Clayton, (1990a), ***Nucl. Instr. Meth***. B47, 55-71.

C.G. Ryan and D.N. Jamieson, (1993), ***Nucl. Instr. Meth***. B77, 203-214.

C.G. Ryan, D.N. Jamieson, C.L. Churms and J.V. Pilcher, (1995), ***Nucl. Instr. Meth***. B104, 157-165.

C.G. Ryan, (2000), "Quantitative Trace Element Imaging using PIXE and the Nuclear Microprobe", ***International Journal of Imaging Systems and Technology***, Special issue on "Advances in Quantitative Image Analysis", 11, 219-230.

C.G. Ryan, (2001), "Developments in Dynamic Analysis for quantitative PIXE true elemental imaging", ***Nucl. Instr. Meth***. B 181, 170-179.

C.G. Ryan, E. van Achterbergh, C.J. Yeats, S.L. Drieberg, G. Mark, B.M. McInnes, T.T. Win, G. Cripps, G.F. Suter, (2002), "Quantitative, high sensitivity, high resolution, nuclear microprobe imaging of fluids, melts and minerals", ***Nucl. Instr. Meth***. B 188, 18-27.

C.G. Ryan, E. van Achterbergh and D.N. Jamieson, (2005a), ***Nucl. Instr. and Meth***. B 231, 162-169.

C.G. Ryan, B.E. Etschmann, S. Vogt, J. Maser, C.L. Harland, E. van Achterbergh, D. Legnini, (2005b), "Nuclear microprobe – synchrotron synergy: Towards integrated quantitative real-time elemental imaging using PIXE and SXRF", ***Nucl. Instr. Meth.*** B 231, 183-188.

C.G. Ryan, R. Kirkham, R.M. Hough, G. Moorhead, D.P. Siddons, M.D. de Jonge, D.J. Paterson, G. De Geronimo, D.L. Howard and J.S. Cleverley, (2010a), "Elemental X-ray imaging using the Maia detector array: The benefits and challenges of large solid-angle", ***Nucl. Instr. Meth*** A 619, 37-43.

C.G. Ryan, D.P. Siddons, R. Kirkham, P.A. Dunn, A. Kuczewski, G. Moorhead, G. De Geronimo, D.J. Paterson, M.D. de Jonge, R.M. Hough, M.J. Lintern, D.L. Howard, P. Kappen and J. Cleverley, (2010b), "The New Maia Detector System: Methods For High Definition Trace Element Imaging Of Natural Material", X-ray Optics and Microanalysis, ***AIP Conference Series*** 1221, 9-17.

C.G. Ryan, R. Kirkham, G.F. Moorhead, D. Parry, M. Jensen, A. Faulks, S. Hogan, P.A. Dunn, R. Dodanwela, L.A. Fisher, M. Pearce, D.P. Siddons, A. Kuczewski, U. Lundström, A. Trolliet and N. Gao, (2018), "Maia Mapper: high definition XRF imaging in the lab", ***J. Instrumentation*** 13, C03020.

D.P. Siddons, A. Dragone, G. De Geronimo, A. Kuczewski, J. Kuczewski, P. O'Connor, Z. Li, C.G. Ryan, G. Moorhead, R. Kirkham, P.A. Dunn, (2006), "A High-speed Detector System for X-ray Fluorescence Microprobes", proc. ***IEEE Nuclear Science Symposium***, *Medical Imaging and Room Temperature Detector Workshop*, San Diego, October 2006.

Dynamic Analysis for Real Time Imaging

This brief sketch is an adjunct to the GeoPIXE manual. Refer to the manual for a guide to spectrum fitting and building the Dynamic Analysis matrix. Use these notes to make use of a DA matrix to perform real-time processing of an event stream (or pixel spectra) to accumulate DA element images.

The process of using a Dynamic Analysis matrix for real-time image projection works something like this:

1. For each event, use the detector number to index to the correct energy calibration table for this detector.

2. Convert detector pulse-height channel number to energy (keV) using its calibration.

3. Now using the energy calibration of the DA matrix (often the same as the first detector in the array, but with a different offset), convert this energy to channel, or column in the DA matrix.

4. This column contains increments, one for each element, to make to the elemental images at the current X,Y position.

Some details modify this a little. It is nice to have the images quantitative and record ppm-flux units regardless of how many detectors are selected from a detector array. This is done by scaling the matrix column values down by the effective multiplicity, which is the sum of the “relative sensitivity factors”. Similarly, we scale the variance image increments down by this number squared. This has not been done below.

This C code fragment illustrates the process for a single detector:

```
/*     Assume we have an event (N,E,X,Y) from detector N:

       matrix is array is         float  DA[columns][rows]
       matrix cal is              float  DA_calA, DA_calB

       detector calibrations:     float  calA[], calB[]

       images are arrays:         float  image[el][y][x]
       variance arrays:           float  var[el][y][x]
       number of elements:        long   N_el
*/
  long column;
  float e, f, dinc;

  e = E * calA[N] + calB[N];                  /* event energy     */
  column = (e - DA_calB) / DA_calA;           /* DA matrix col    */

  for (i=0; i<N_el; i++) {
    dinc = DA[column][i];

    image[i][Y][X] = image[i][Y][X] + dinc;
    var[i][Y][X] = var[i][Y][X] + dinc*dinc;
  }
```

These are quantitative image in the sense that we can at any time extract estimates of average elemental concentration (e.g. within an arbitrary shaped area, or along a profile traverse) by integrating an area of the `image` array and dividing by the integrated beam flux for the same area. Similarly, one-sigma uncertainty comes from integrating the variance array `var`, taking the square-root, and dividing by flux. Hence, concentration colour-bar legends can be applied in real-time, and interactive concentrations can be extracted from images, even during data collection (for [partly] completed parts of the image area).

The ‘export_DA.pro’ procedure dumps a binary file containing the matrix and other details, such as energy calibration and the identity of the active detectors.

Note that byte-ordering has already been fixed for a Little Endian target processor, such as a PC (Linux or Windows). This option is set in the ‘Export_DA2.pro’ routine.

The file has this format:

File is dumped as follows:

```
      long   version;                          /* version number   */
      struct header       header;              /* header           */
      unsigned byte       b[N_el][ns];         /* element labels   */
      float               DA[N_el][size];      /* DA matrix        */
```

where the header struct is:

```
struct header
{
      long   N_el;        /* number of elements                        */
      long   size;        /* number of columns in DA matrix            */
      float  DA_calA;     /* energy cal gain of DA matrix              */
      float  DA_calB;     /* energy offset of DA matrix cal            */
      long   ns;          /* number of characters in element names     */
};
```

where the ‘b’ strings are null terminated and typically contain 4 chars (ns = 4, including NULL). All energy calibration data is in keV.

Note that the matrix does not usually extend all the way to zero energy, or zero original channel number. Only the fitted energy range is usually included. So the mapping using calibration of the matrix and detector elements is very important to pick up the correct column.

The first element is usually “Bac”, a truncation of “Back”, which is the background image and maps the variation of the underlying background component as fitted to the master spectrum. Other element labels are standard element mnemonics (2 chars plus NULL), perhaps with an appended ‘L’ or ‘M’ to indicate the use of L or M lines (3 chars plus NULL).

Back is a useful diagnostic, as any missing elements (i.e. elements present but omitted from the build of the DA matrix) usually contribute to the background image. Watch out for intense spots here that may indicate a concentration of an element that was initially not noticed but is actually significant in a small area of the image.

Some Refinements

Rather than calculate ‘e’ from initial channel number and then ‘column’ from ‘e’, a set of lookup tables ‘clookup’ can be built initially, one for each active detector:

```
  for (j=0; j<N_det; j++) {
    for (i=0; i<size; i++) {
       x = (float)i;
       clookup[j][i] = (long))(calA[j]*x + calB[j]-DA_calB)/DA_calA + 0.5);

       if (clookup[j][i] < 0)||( clookup[j][i] >=size) {
          clookup[j][i] = -1;
       }
```

```
    }
  }
```

where the addition of 0.5 provides some simple rounding to form '`clookup`'. And then the main data projection loop from above simplifies to:

```
  j = lookup[N];
  f = 1./(float)N_det;
  If (j >= 0) {
    column = clookup[j][E];

    if (column >= 0) {
      for (i=0; i<N_el; i++) {
        dinc = DA[column][i] * f;

        image[i][Y][X] = image[i][Y][X] + dinc;
        var[i][Y][X] = var[i][Y][X] + dinc*dinc;
      }
    }
  }
```