Psana User Manual - Old

⚠️ This page includes material from PSDM space, due to that navigation on page does not work properly. To see original page go to psana - User Manual

Unknown macro: 'html'

- Introduction
- Framework Architecture
- Analysis Job Life Cycle
- User Modules
- Controlling Framework from User Module
- Job and Module Configuration
  - Configuration File Format
  - Parameter Types
  - Psana Parameters
  - User Modules Parameters
  - Accessing Configuration Parameters
- Messaging Service
- Writing User Modules
- Running Psana
  - Specifying input data
- Psana Module Examples

Unknown macro: 'html'

Introduction

This document describes C++ analysis framework for LCLS and how users can make use of its features. Psana design borrows ideas from multitude of other framworks such as pyana, myana, BaBar framework, etc. It's main principles are summarized here:

- support processing of both XTC and HDF5 data format
- user code should be independent of specific data format
- should be easy to use and extend for end users
- support re-use of the existing analysis code
- common simple configuration of user analysis code

This manual is accompanied by the psana - Reference Manual which describes interfaces of the classes available in Psana.

Framework Architecture

The central part of the framework is a regular pre-built application (psana) which can dynamically load one or more user analysis modules which are written in C++ or Python. The core application is responsible for the following tasks:

- loading and initializing all user modules
- loading one of the input modules to read data from XTC or HDF5
- calling appropriate methods of user modules based on the data being processed
- providing access to data as set of C++ classes and a set of Python classes
- providing other services such as histogramming to user modules

Other important components of the Psana architecture:

- *user module* – instance of the C++ or Python class which inherits pre-defined Module class and defines special methods which are called by the framework
- *event* – special object which transparently stores all event data
- *environment* – special object which stores non-event data such as configuration objects or EPICS data

Analysis Job Life Cycle

Psana analysis job goes through cycles of state changes such as initialization, configuration, event processing, etc. calling methods of the user modules at every such change. This model follows closely the production activities in LCLS on-line system. DAQ system defines many types of transitions in its data-taking activity, most interesting are here:

- `Configure` - provides configuration data for complete setup
- `BeginRun` - start of data taking for one run
- `BeginCalibCycle` - start of the new scan, some configuration data may change at his point
- `L1Accept` - this is regular event containing event data from all detectors
- `EndCalibCycle` - end of single scan
- `EndRun` - end of data taking for one run

- `Unconfigure` - stop of all activity

Typically there will be more than one run taken with the same configuration, so there may be more than one `BeginRun/EndRun` transition for one `Configure/Unconfigure`, but a data file from single run should contain only one `BeginRun/EndRun`. Depending on a setup there could be one or more `BeginCalibCycle/EndCalibCycle` transitions in single run.

For each of the above transitions psana will call corresponding method in user modules notifying them of the possible change in the configuration or just providing event data. Following method names are defined in the user modules:

- **beginJob()** – this method is called **once** per analysis job when first `Configure` transition happens. If there is more than one `Configure` in single job (when processing multiple runs) this method is not called, use beginRun() to observe configuration changes in this case. This method can access all configuration data through environment object.
- **beginRun()** – this method is called for every new `BeginRun`, so it will be called multiple times when processing multiple runs in the same job. This method can access all configuration data through environment object.
- **beginCalibCycle()** – this method is called for every new `BeginCalibCycle`, so it will be called multiple times when processing multiple runs in the same job or when single run contains multiple scans. This method can access all configuration data through environment object.
- **event()** – this method is called for every new `L1Accept`, it has access to event data through event object as well as configuration data through environment object.
- **endCalibCycle()** – this method is called for every new `EndCalibCycle`, it has access to configuration data through environment object.
- **endRun()** – this method is called for every new `EndRun`, it has access to configuration data through environment object.
- **endJob()** – this method is called **once** at the end of analysis job, it has access to configuration data through environment object.

Typically psana will iterate through all transitions/events from the input files. User modules have a limited control over this event loop, module can request to skip particular event, stop iteration early or abort job using one of the methods described below.

User Modules

A user module provides an instance of a class that inherits from the Psana `Module` class. Below we discuss this for C++. The Psana `Module` class is defined in the file `psana/Module.h` and implements several methods. These methods are already mentioned above, here is more formal description of each method:

- `void` **`beginJob`** `(Event& evt, Env& env)`
  Method called once at the beginning of the job. Environment object contains configuration data from the first `Configure` transition. Default implementation of this method does not do anything.
- `void` **`beginRun`** `(Event& evt, Env& env)`
  Method called at the beginning of every new run. Default implementation of this method does not do anything.
- `void` **`beginCalibCycle`** `(Event& evt, Env& env)`
  Method called at the beginning of every new scan. Default implementation of this method does not do anything.
- `void` **`event`** `(Event& evt, Env& env)`
  Method called for every regular event. Even data is accessible through =evt= argument. There is no default implementation for this method and user module must provide at least this method.
- `void` **`endCalibCycle`** `(Event& evt, Env& env)`
  Method called at the end of every new scan, can be used to process scan-level statistics collected in `event()`. Default implementation of this method does not do anything.
- `void` **`endRun`** `(Event& evt, Env& env)`
  Method called at the end of every run, can be used to process run-level statistics collected in `event()`. Default implementation of this method does not do anything.
- `void` **`endJob`** `(Event& evt, Env& env)`
  Method called once at the end of analysis job, can be used to process job-level statistics collected in `event()`. Default implementation of this method does not do anything.

In addition to `event()` method every module class must provide a constructor which takes a string argument giving the name of the module. Additionally it has to provide a special factory function used to instantiate the modules from the shared libraries, there is special macro defined for definition of this factory function.

Here is the minimal example of the module class declaration with only the `event()` method implemented and many non-essential details are skipped:

Package/ExampleModule.h

```
#include "psana/Module.h"

namespace Package {
class ExampleModule: public Module {
public:

  // Constructor takes module name as a parameter
  ExampleModule(const std::string& name);

  // Implementation of event() from base class
  virtual void event(Event& evt, Env& env);

};
} // namespace Package
```

Definition of the factory function and methods:

Package/ExampleModule.cpp

```
#include "Package/ExampleModule.h"
#include "MsgLogger/MsgLogger.h"
#include "PSEvt/EventId.h"

// define factory function
using namespace Package;
PSANA_MODULE_FACTORY(ExampleModule)

// Constructor
ExampleModule::ExampleModule(const std::string& name)
  : Module(name)
{
}

void
ExampleModule::event(Event& evt, Env& env)
{
  // get event ID
  shared_ptr<EventId> eventId = evt.get();
  if (not eventId.get()) {
    MsgLog(name(), info, "event ID not found");
  } else {
    MsgLog(name(), info, "event ID: " << *eventId);
  }
}
```

This simple example already does something useful, it retrieves and prints event ID (copied from standard PrintEventId module). Actual modules will do more complex things but this is a simple example of obtaining something from event data.

The easiest way to write new user modules is to use `codegen` script to generate class from predefined template. This command will create new module `ExampleModule` in package `TestPackage` and will copy generated files to the directories in TestPackage:

```
codegen -l psana-module TestPackage ExampleModule
```

Controlling Framework from User Module

Code in user modules can control framework event loop by calling one of the three methods:

- `void skip()`
  Signal framework to skip current event and do not call other downstream modules. Note that this method does not skip code in the current module, control is returned back to the module. If you want to stop processing after this call then add a return statement.

- `void stop()`
  Signal framework to stop event loop and finish job gracefully (with calling endRun/endJob/etc.). Note that this method does not terminate processing in the current module. If you want to stop processing after this call then add a return statement.
- `void terminate()`
  Signal framework to terminate immediately. Note that this method does not terminate processing in the current module. If you want to stop processing after this call then add a return statement.

Here is an example of the code using above functions:

```
void ExampleModule::event(Event& evt, Env& env) {

  ...

  if (pixelsAboveThreshold < 1000) {
    // This event is not worth looking at, skip it
    skip();
    // I do not want to continue with this algorithm either
    return;
  }

  if (nGoodEvents > 1000) {
    // we collected enough data, can stop now and go to endJob()
    stop();
    // I do not want to continue with this algorithm either
    return;
  }

  if (temperatureKelvin < 0) {
    // data is junk, stop right here and don't call endJob()
    terminate();
    // I do not want to continue with this algorithm either
    return;
  }

}
```

Skipped events can be used in further analysis or saved in the "filtered" Xtc file, as explained in Package PSXtcOutput.

Job and Module Configuration

Psana framework has multiple configuration parameters that can be changed via command line or special configuration file. Configuration file can also specify parameters for user modules so that modules' behavior can be changed at run time without the need to recompile the code.

If no options are specified on the command line then psana tries to read configuration file named `psana.cfg` from the current directory if that file exists. The location of the configuration file can be changed with the `-c <path>` option which should provide path of the configuration file.

Configuration File Format

Configuration file has a simple format which is similar to well-known `INI file format`. The file consists of the sections, each section begins with the section header in the form:

```
[<section-name>]
```

Section names can be arbitrary strings, but in psana case section names are the names of the modules which cannot be arbitrary and should not contain spaces.

Following the section header there may be zero or more parameter lines in the form

```
<param-name> = <param-value>
```

Parameter name is anything between beginning of line and '=' character with leading and trailing spaces and tabs stripped. Parameter value is anything after '=' character with leading and trailing spaces and tabs stripped, parameter value can be empty. Long parameter value can be split over multiple lines if the line ends with the backslash character, e.g.:

```
```

```
files = /reg/d/psdm/AMO/amo00000/xtc/e00-r0000-s00-c00.xtc \
        /reg/d/psdm/AMO/amo00000/xtc/e00-r0000-s01-c00.xtc \
        /reg/d/psdm/AMO/amo00000/xtc/e00-r0000-s02-c00.xtc
```

Lines starting with '#' character are considered comments and ignored.

Parameter Types

Configuration file does not specify parameter types, all values in the file are strings. Psana framework provides conversion of these strings to several basic C++ types or sequences. Following types and conversion rules are supported by framework:

- `bool`
  value strings "yes", "true", "on" become `true`, "no", "false", "off" become `false`. Strings which represent non-zero numbers become `true`, string "0" becomes `false`.
- `char`
  value string must be single-character string and it will be assigned to a result.
- C++ numeric types
  option value must represent valid number.
- `std::string`
  option value will be assigned to result string without change.
- C++ sequence types (e.g. `std::list<T>`)
  option value will be split into single words at space/tab characters, individual words will be converted to resulting type `T`.

When the conversion fails because of the incorrectly formatted input framework will throw an exception with the type `ExceptionCvtFail`.

Psana Parameters

The parameters that are needed for the framework are defined in `psana modules` section. Here is the list of parameters which can appear in that section:

- `modules`
  list of module names to include in the analysis job. Each module name is built of a package name and class name separated by dot (e.g. `TestPackage.ExampleModule`) optionally followed by colon and modifier. Modifier is not needed if there is only one instance of the module in the job. If there is more than on instance then modules need to include unique modifier to distinguish instances. If the module comes from psana package then package name can be omitted. Module names can also be specified on the command line with `-m` option, for multiple modules use multiple `-m` options or comma-separated names in single -m option.
- `input` or `files`
  specifies input data, list of datasets or file names to process. Input data can also be specified on the command line which will override anything specified in configuration file. See section Specifying input data for more details on dataset syntax.
- `events`
  maximum number of events to process in a job, can also be given on the command line with `-n` or `--num-events` option.
- `skip-events`
  number of events to skip before starting even processing, can also be given on the commnad line with `-s` or `--skip-events` option.
- `instrument`
  Instrument name.
- `experiment`
  Experiment name. Instrument and expriment names can be specified on the commnad line with `-e` or `--experiment` option, option value has format `XPP:xpp12311` or `xpp12311`. By default instrument and experiment names are determined from input file names, you can use these options to override defaults (or when your file has non-standard naming).
- `calib-dir`
  Path to the calibration directory, can also be given on the commnad line with `-b` or `--calib-dir` option. Path can include {instr} and {exp} strings which will be replaced with instrument and experiment names respectively. Default value for path is `/reg/d/psdm/{instr}/{exp}/calib`.

Here is an example of the framework configuration section:

```
[psana]
# list of file names
files = /reg/d/psdm/AMO/amo00000/xtc/e00-r0000-s00-c00.xtc \
        /reg/d/psdm/AMO/amo00000/xtc/e00-r0000-s01-c00.xtc \
        /reg/d/psdm/AMO/amo00000/xtc/e00-r0000-s02-c00.xtc
# list of modules, PrintSeparator and PrintEventId are from psana package
# and do not need package name
modules = PrintSeparator PrintEventId psana_examples.DumpAcqiris
```

User Modules Parameters

Parameters for user modules appear in the separate sections named after the modules. For example the module with name "TestPackage.ExampleModule" will read its parameters from the section `[TestPackage.ExampleModule]`.

To help manage configuration options, Psana provides a way select between several sets of parameters in a config file, as well as to override a default set with a few specific values. When specifying a module to load, it can be tagged as follows:

```
modules = TestPackage.Analysis:mode1
```

The modifier after the colon tells Psana to first look for configuration parameters in the section `[TestPackage.Analysis:model]` and then in the section `[TestPackage.ExampleModule]`. It is also possible to load the same module several times, specifying different configuration options for each instance. Psana will construct each instance with a different name - based on the tag provided.

Here is an example of configuration for some fictional analysis job:

```
[psana]
modules = TestPackage.Analysis:mode1 TestPackage.Analysis:mode2

[TestPackage.Analysis]
# these are common parameters for all TestPackage.Analysis modules,
# but instances can override then in their own sections
calib-mode = fancy
subpixel = off
threshold = 0.001

[TestPackage.Analysis:mode1]
# parameters specific to :mode1 module
range-min = 0
range-max = 1000000

[TestPackage.Analysis:mode2]
# parameters specific to :mode2 module
range-min = 1000
range-min = 10000
subpixel = on
```

Accessing Configuration Parameters

User module base class defines few convenience methods which simplify access to configuration parameters. Here is the list of the methods:

- `std::string` **`configStr`** `(const std::string& param)`
  this method takes the name of the parameter and returns full parameter value as a string. If parameter cannot be found the exception will be thrown.
- `T` **`config`** `(const std::string& param)`
  this method takes the name of the parameter and returns parameter value converted to type `T`. If parameter cannot be found the exception will be thrown.
- `std::string` **`configStr`** `(const std::string& param, const std::string& def)`
  this method takes the name of the parameter and returns full parameter value as a string. If parameter cannot be found then the value of second argument will be returned.
- `T` **`config`** `(const std::string& param, T def)`
  this method takes the name of the parameter and returns parameter value converted to type `T`. If parameter cannot be found then the value of second argument will be returned.
- `Seq configList(const std::string& param)`
  this method takes the name of the parameter and returns parameter value converted to sequence. Sequence can be any of standard container types such as `std::list<std::string>` or `std::vector<double>`. If parameter cannot be found the exception will be thrown.
- `std::list<T> configList(const std::string& param, const std::list<T>& def)`
  this method takes the name of the parameter and returns parameter value converted to `std::list<T>`. If parameter cannot be found then the value of second argument will be returned.

Here is an example of the code in user module which uses these methods:

```
Source src = configStr("source", "DetInfo(:Evr)");
int repeat = config("repeat");
std::list<std::string> options = configList("options");
```

Messaging Service

In many cases the user modules want to produce/print messages such as errors, warnings, or debugging information. In most cases C++ code uses standard C++ facilities such as `std::cout`, `std::cerr`, or even `printf` to format/print something to the terminal or log file. Psana framework provides different approach for messaging which provides better control for the output level (e.g. turning on/off debugging) and better flexibility.

Each message produced by messaging service carries corresponding level. There are several levels of messages defined by the service:

- `debug` – lowest message level reserved for debugging messages, normally turned off during normal running
- `trace` – one level higher than `debug`, normally turned off during normal running
- `info` – level for regular informational messages, normally printed but can be turned off
- `warning` – level for warnings which are not errors
- `error` – level for error messages
- `fatal` – level for fatal errors, after the message is published the program will terminate
  The levels are ordered, enabling messages of one level also enables messages of all higher levels.

Each logging message is associated with one *logger*. Loggers have names which form hierarchical structure such as "GrandParent.Parent.Child". Top-level logger has no name and is called *root logger*. Loggers were introduced for flexibility, it is possible to configure individual loggers, for example to enable debug logging from one particular logger. Good practice is to use logger name which is the same as user module name for identification purposes.

To use messaging service one has to include header file `"MsgLogger/MsgLogger.h"` which defines a set of macros for message logging and all related classes. User code interacts with the messaging service through this set of macros:

- `MsgLog(logger, level, message)` // send a message to specific logger, takes logger name, logging level, and message. Message is a construct which can appear after stream insertion operator (e.g. `cout << message`).
- `MsgLogRoot(level, message)` // same as above but message is sent to root logger.

Here are few examples of using these macros:

```
MsgLog("MyModule", info, "reading pedestals from file " << fileName);
MsgLog("MyModule", debug, "intermediate result: count=" << count << " sum=" << sum);
MsgLogRoot(warning, "warp engine overheating");
```

Note: in user module replace "MyModule" string with the name() call which returns the name of the user module.

Above macros are simple to use in most cases as they hide all details from user. In more complex situations (printing array elements) there are two macros which provide access to underlying stream object which can be used in more interesting ways:

- Unknown macro: 'html'

  this macro declares stream object which can be used by the code in compound statement which follows the macro. The lifetime of the stream is the code block, after the code block is executed the message is published and stream disappears.
- Unknown macro: 'html'

  variation of the above macro which publishes message to root logger.

Here is an example of their use:

```
WithMsgLog("MyModule", debug, str) {
  str << "array elements:";
  for (int i = 0; i < size; ++ i) {
    str << " " << array[i];
  }
}
```

When messaging service delivers (prints) the message in addition to message itself it provides additional information about message. In psana it will print level name and logger name; for trace messages it will also print timestamp; for debug and error messages it will print timestamp and location (file name and line number) where message originated.

By default psana enables messages of the `info` level (and higher). To enable lower level messages one can provide -v option to psana: one -v will enable `trace` messages, two -v options will enable `debug` messages. To disable `info` and `warning` messages one can provide one or two -q options. Error and fatal messages cannot be disabled.

Note: when the message level is disabled the code in the corresponding macros is not executed at all. Do not put any expressions with side effects into message or code blocks, these are strictly for messaging, not part of your algorithm.

Writing User Modules

Here are few simple steps and guidelines which should help users to write their analysis modules.

- Everything is done in the context of the off-line analysis releases, your environment should be prepared and you should have test release setup based on one of the recent analysis releases. Consult Workbook which should help you going.
- You need your own package which may host several analysis modules. Package name must be unique. If the package has not be created yet run this command:

```
newpkg MyPackage
mkdir MyPackage/include MyPackage/src
```

- Generate skeleton module class from template:

```
codegen -l psana-module MyPackage MyModule
```

  this will create two files: `MyPackage/include/MyModule.h` and `MyPackage/src/MyModule.cpp`
- Edit these two files, add necessary data members and implementation of the methods.
- For examples of accessing different data types see collection of modules in `psana_examples` package. Reference for all event and configuration data types is located at https://pswww.slac.stanford.edu/swdoc/releases/ana-current/psddl_psana/
- Reference for other classes in psana framework: Psana Reference Manual
- Run `scons` to build the module library.
- Create psana config file if necessary.
- Run `psana` providing input data, configuration file, etc.
- It is also possible that somebody wrote a module which you can reuse for your analysis, check the module catalog: psana - Module Catalog

To add your own compiler or linker options to the build (such as to link to a third party library), see this section on customizing the scons build.

Running Psana

After writing and compiling the modules (or choosing standard modules) one can run psana application with these modules. Psana application is pre-built and does not need to be recompiled. To start application one needs to either provide a configuration file or corresponding command-line options. Some information (e.g. user module options) cannot be specified on the command line and always require configuration file. Here is the list of command-line options recognized by psana:

```
Usage: psana [options] [dataset ...]

  Available options:
    {-h|-?|--help    }         print help message
    {-v|--verbose    } (incr)  verbose output, multiple allowed (initial: 0)
    {-q|--quiet      } (incr)  quieter output, multiple allowed (initial: 2)
    {-b|--calib-dir  } path    calibration directory name, may include {exp} and {instr}, if left empty then do
not do calibrations (default: "")
    {-c|--config     } path    configuration file, by default use psana.cfg if it exists (default: "")
    {-e|--experiment } string  experiment name, format: XPP:xpp12311 or xpp12311, by default guess it from data
(default: "")
    {-j|--job-name   } string  job name, default is to generate from input file names (default: "")
    {-m|--module     } name    module name, more than one possible
    {-n|--num-events } number  maximum number of events to process, 0 means all (default: 0)
    {-s|--skip-events} number  number of events to skip (default: 0)
    {-o|--option     } string  configuration options, format: module.option[=value]

  Positional parameters:
    dataset - input dataset specification (list of file names or exp=cxi12345:run=123:...)
```

If both options `-c` and `-m` are missing from the command line then psana reads configuration file `psana.cfg` from current directory. Otherwise if `-c` option is provided with the file name psana reads corresponding configuration file.

Modules loaded by psana can be specified in configuration and on command line with `-m` option. If `-m` option is provided then its value overrides module list specified in the configuration file. One can provide comma-separated list of module names or multiple `-m` options on the command line, following command lines are all equivalent:

```
% psana -m ModuleA,ModuleB,ModuleC ...
% psana -m ModuleA -m ModuleB -m ModuleC ...
% psana -m ModuleA,ModuleB -m ModuleC ...
```

Option `-j` can change job name which defines then names of the output histogram file. By default job name is constructed from the name of the first input file.

Input data can also be specified in the configuration file or on command line, command-line arguments override configuration file values. Check section below for complete description of dataset format.

Command-line options `-v` and `-q` can increase or decrease verbosity of the output generated by messaging service. By default psana outputs messages at `info` and higher levels. With one `-v` option `trace` messages will be printed also, and with two or more `-v` options `debug` messages will be printed too. With `-q` option `info` messages will not be printed, only `warning`, `error`, and `fatal`.

Here are few examples of running psana applications:

```
% psana -m EventKeys /reg/d/psdm/...
% psana -m psana_examples.EBeamHist -j ebeam-hist-r1000 /reg/d/psdm/...
% psana -c psana_examples/data/DumpAll.cfg exp=cxi12345:run=123
% psana                  # everything will be specified in psana.cfg file
```

Specifying input data

Input data for psana are specified on command line or in configuration file using special dataset syntax. More than one dataset can be specified in arbitrary order, psana will will order datasets accordingly, so that events from all datasets are time-ordered.

In simplest case dataset is just a file name containing input data in either XTC of HDF5 format. File name should be given as a full path name, if there are more than one stream or chunk in XTC data, all of them must be specified.

More advanced and recommended way is to provide input data as a special dataset string. The dataset string encodes various parameters, some of which are needed to locate data files, while others specify optional behavior such as filtering or live data reading. The general syntax of the dataset string is a list colon-separated parameters, parameters have optional values separated from parameter name by equal sign:

```
param[=value][:param[=value][...]
```

These are some of the parameters which are supported in psana:

- experiment name (which may optionally contain the name of an instrument)

  ```
  exp=CXI/cxi12313
  exp=cxi12313
  ```

- run number specification (can be a single run, a range of runs, a series of runs, or a combination of all above)

  ```
  run=1
  run=10-20
  run=1,2,3,4
  run=1,20-20,31,41
  ```

- file type, if not specified then 'xtc' is the default

  ```
  xtc
  h5
  ```

- Location of the files, if not specified then files will be searched in a standard location (/reg/d/psdm/...). If this parameter is specified it needs to be full path name of the directory where files are located

  ```
  dir=/reg/d/ffb/cxi/cxi12345/xtc
  ```

- Input number stream number for XTC files, if value is omitted then one pseudo-random stream is selected (this is useful to balance the load on FFB storage system for example):

  ```
  one-stream=1
  one-stream
  ```

- allow reading from live XTC files while they're still being recorded (by the DAQ or by the Data Migration service). Note that this feature is only available when running **psana** at PCDS, in all other cases the option will be ignored:

  ```
  live
  ```

Few examples of dataset specification:

- To read XTC data from specific run number:

  ```
  exp=xpp12345:run=123
  ```

- To read HDF5 from several runs:

```
exp=xpp12345:run=1,5,7-10:h5
```

- To read live XTC data from a random stream from FFB directory

```
exp=xpp12345:run=1123:live:one-stream:dir=/reg/d/ffb/xpp/xpp12345/xtc
```

The complete description of the data set string syntax and allowed parameters can be found in the specification document.

Psana Module Examples

A set of psana modules is available in current release as explained in Psana Module Catalog. Part of them demonstrates how data can be accessed from user module code . Other modules can be used in data analysis or event filtering. Example of application for these modules are available in separate document:

- Psana Module Examples - for advanced modules for analysis and event filtering

We continually develop algorithms for the standard set of psana modules. If the algorithm you need is missing in our collection we would be interested in hearing about it (email pcds-help@slac.stanford.edu). We are interested in implementing algorithms that are useful to our users. Of course, following this document, you can develop a Psana modules that implements the algorithm. A resource for sharing the module is the Users' Software Repository.