Pyana User Manual

Introduction

This page collects information about Python-based analysis framework for LCLS called *pyana*. This framework design borrows heavily from the various sources such as Online Analysis (a.k.a. *myana*), BaBar framework, etc. It's main principles are summarized here:

- oriented on XTC processing, but could be extended to work with HDF5 data
- should be easy to use and extend for end users
- support re-use of the existing analysis code
- allow parallel processing on multi-core systems
- common simple configuration of user analysis code

This manual is accompanied by Reference Manual which describes interface of all analysis objects accessible to the user analysis job.

Framework Composition

The centerpiece of the framework is a regular Python application (*pyana*) which can load one or more user analysis modules which are also written in Python. The core application is responsible for the following tasks:

- loading and initializing all user modules
- reading XTC data from a list of input files
- calling appropriate user methods based on the data being processed
- providing data access to user modules
- providing other services such as histogramming to user modules

The core application have a number of configuration options which can be set or changed from configuration file or from a command line. If the same option appears both in configuration file and command line the command line value overrides value in the configuration file.

User Modules

User analysis module is a regular Python module (a Python file) which satisfies additional requirements:

- it contains a class with the same name as the module name
- the class defines constructor method with optional arguments and three regular methods: `beginjob()`, `event()`, and `endjob()`; and four optional methods: `beginrun()`, `endrun()`, `begincalibcycle()`, and `endcalibcycle()`.

The application loads one or more user modules, the names of the modules to load are specified either in the job configuration file or on the command line. After loading the modules the application creates one or more instance objects of the class defined in the module. More than one instance may be useful if one wants to run the same analysis with different set of parameters in the same job. Number of instances and their parameters are determined by the job configuration file (see below).

Initialization

User analysis class can define zero or more parameters in its constructor (`__init__()` method). Parameters are initialized from the values defined in the job configuration file (see below). All parameters are passed to the Python code as strings, if the code expects a number or some other type then it's the code responsibility to convert the strings to appropriate type. If there are no default values defined for some parameters in constructor declaration then those parameters must be present in the configuration file.

For quick example suppose that we have this class defined in user module:

mypackage/src/myana.py

```
# user analysis class
class myana(object):
    def __init__(self, name, lower, upper, bins=100)
        self.name = name
        self.lower = float(lower)
        self.upper = float(upper)
        self.bins = int(bins)
    ...
```

and this job configuration file:

pyana.cfg

```
[pyana]
modules = mypackage.myana mypackage.myana:wide

[mypackage.myana]
lower = 0
upper = 100
name = default

[mypackage.myana:wide]
lower = 0
upper = 1000
bins = 1000
name = wide
```

With this the analysis job will instantiate two analysis objects with different parameters, equivalent to this pseudo-code:

```
# import class myana
from mypackage.myana import myana

# create instances
instances = [ myana(lower = "0", upper = "100", name = "default"),
    myana(lower = "0", upper = "1000", bins = "1000", name = "wide") ]
```

(the order of parameters in constructor and configuration file does not matter as all parameters are passed as keyword parameters.)

Data access

There are two types of data that framework passes to the user analysis modules – event data and environment data. Event data contains the data corresponding to current event that triggered the call to the user methods. In case of XTC input the event data contains complete datagram as read from DAQ. Event data in user module is represented with a special object of type `pyana.event.Event` which has an extended interface for extracting individual object from datagram. This interface is described in the reference guide.

Environment data include all kinds of data which are not part of the event data. Usually environment data either stay the same for the whole job or change at a slower rate than event data. Example of the environment data could be configuration data read from XTC at the beginning of the job, EPICS data which is not updated on every event, and few other things. Environment data is represented for user code through the object of type `pyana.event.Env`. Its interface is described in the reference guide.

Data Source Address

For some pieces of data one needs to specify data "address" which identifies (maybe partially) particular DAQ device which produced the data. This is needed because the instrument setup may include multiple devices producing the same data type. The DAQ defines a type which serves as a most specific device identification, the type is `xtc.DetInfo` in package `pypdsdata`. One can pass this `DetInfo` instance to a method which accepts device address to select that specific device. `DetInfo` object contains four essential pieces of information:

- `detector` – one of the `DetInfo.Detector.*` values
- `detId` – ID number selecting one of multiple detectors
- `device` – one of the `DetInfo.Device.*` values
- `devId` – ID number selecting one of multiple devices

One can build `DetInfo` out of these four values, but `DetInfo` constructor (which mimics C++ behavior) needs also a nuisance `processId` parameter. In most cases using the address string described below is preferred to manually building `DetInfo` objects.

In many cases the `DetInfo` object is not known to the user or the code uses only partial information to identify one or many data sources. In such cases it is easier to use address string as an argument to methods accepting `address` parameter. The most generic format of address string is:

```
[detector][-[detId]]["|"[device][-[devId]]]
```

In words address string is an optional detector name followed by optional dash and detector ID separated by vertical bar from optional device name followed by optional dash and device ID. Detector name and device name are the names defined in DetInfo.Detector and DetInfo.Device enums respectively. Examples of detector names are AmoGasdet, AmoETof, Camp, etc. Examples of device names are Acqiris, pnCCD, Evr, etc. For complete list check the Reference Manual. If any piece in the address string is missing or is replaced with a special '*' character then it means "match any value".

Here are few example address strings:

- `"AmoETof-0|Acqiris-0"` – selects data produced by detector AmoETof, detId 0, device Acqiris, devId 0
- `"AmoETof|Acqiris"` – selects data produced by detector AmoETof, any detId, device Acqiris, any devId
- `"AmoETof-`**`|Acqiris-`**`"` – same as above
- `"AmoETof-0"` – selects data produced by detector AmoETof, detId 0, any device, any devId
- `"|Acqiris-0"` – selects data produced by any detector, any detId, device Acqiris, devId 0
- `"-|Acqiris-0"` – same as above

Methods

As mentioned above the class in the user module defines number of methods. These methods are called by the Pyana framework at the appropriate moments during data analysis. Here is the explanation when these methods are called and what arguments they accept.

- `beginjob(evt, env)` – this method is called for at a Configure transition. Typically this is the place to initialize various things that may depend on the data being processed. Configuration objects which are part of the Configure transition are accessed through the env object. `evt` object provides interface to the datagram data and can be used to extract all contained data too, but preferred way to access configuration data objects is through the environment object. This method is usually called once per job, but in case when pyana is instructed to process multiple runs it can be called several times if there is more than one Configure transition happened during those runs.

- `endjob(env)` – this method is called at Unconfigure transition. Typically used to process collected statistics, close output files, etc. Like `beginjob()` it can be called multiple times if there is more that one Configure transition happes during the run range being processed.

- `beginrun(evt, env)` – this method is called for at a BeginRun transition. There is usually no data associated with this transition so `evt` object would be empty, but `env` object contains all configuration objects. This method is called once for every run and is a good place to prepare for the processing of the next run.

- `endrun(env)` – this method is called for at a EndRun transition. Typically used to process statistics collected during the run.

- `begincalibcycle(evt, env)` – this method is called for at a BeginCalibCycle transition. This method is called once for every calibration cycle.

- `endcalibcycle(env)` – this method is called for at a EndCalibCycle transition. Typically used to process statistics collected during the calibration cycle.

Methods `beginrun()`, `endrun()`, `begincalibcycle()`, and `endcalibcycle()` are optional, analysis module does not have to define them and they are called only if defined.

Two methods `evt.put` and `evt.get` allow to transfer data between different modules.

- Save new object in event:
  `evt.put( object, object_name )` – this method is called when any newly evaluated `object` needs to be saved in the `evt` store. To access this object from other module it needs to be associated with unique `object_name` – string parameter.

- Retrieve object from event:
  `object = evt.get( oobject_name )` – this method is called when `object` needs to be retrieved form the `evt` store.

Event Loop Control

Code in user modules can control framework event loop by returning a value from `event()` method which is different from `None` (if there is no return statement in the method it is equivalent to returning `None`). Following values are recognized by framework:

- `pyana.Skip`
  This will skip `event()` all downstream modules
- `pyana.Stop`
  This will stop event loop, all `end*()` methods are called as usual
- `pyana.Terminate`
  This will cause immediate job termination, `end*()` methods are not called

Values `pyana.Stop` and `pyana.Terminate` only work in single-process mode, in multi-process they are ignored with warning message issued if user module tries to use them.

Here is simplified example of this feature use:

```
# import is necessary to use return codes
import pyana

class ExampleModule(object):

    def event(self, evt, env):

        ...

        if pixelsAboveThreshold < 1000:
            # This event is not worth looking at, skip it
            return pyana.Skip

        if self.nGoodEvents > 1000:
           // we collected enough data, can stop now and go to endjob()
           return pyana.Stop

        if temperatureKelvin < 0:
            # data is junk, stop right here and don't call endJob()
            return pyana.Terminate
```

Exception Handling

Pyana does not do anything special to handle exceptions which happen in user modules, main reason for this is that it is not safe in general to continue after unknown exception was raised. If user code knows which exception can be raised and is prepared to handle those exception then corresponding code should be added to the user module.

If user module generates an exception and does not handle it the whole job is terminated immediately. In single-process mode the standard traceback will printed by the interpreter and you should see clearly the reason and location of the exception. In multi-process mode (see #Multi-processing) the job will still fail but failure will look more complex. The original exception will cause termination of only a single worker process, the standard traceback for that exception will be printed as usual. The termination of one worker process will cause communication failure inside the main process which will terminate immediately with error message (Broken pipe). This in turn will cause exceptional failures of other worker processes which will print their own tracebacks. So instead of one single traceback for an exception there will be more than one error message appearing in the output.

Configuration

Analysis job can read the configuration options from the command line and/or the configuration file. Command line can be used to set options only for the pyana application itself but not user analysis modules. Options for user modules can be set in configuration file only.

Configuration File

The default name for the configuration file is "pyana.cfg" in the current directory. The name can be changed with the `--config` or `-c` command line option. The format of the configuration file follows the widely accepted syntax for INI files. File consists of series of sections, beginning of each section is marked by the section name in square brackets. Each section contains arbitrary number of options in the form `option = value`. Both section header and option name must start on first column, white space at the beginning of the line means that line is a continuation of the previous option line. Empty lines and lines beginning with word # (hash sign) or `;` (semicolon) are considered comments. Semicolon anywhere on the line as considered a beginning of the comment if it follows space character. The word `REM` (case insensitive) at the beginning of the line starts a comment if it is followed by space, TAB or new-line.

Here is an example of the configuration file syntax:

```
# Comment line
; Another comment
[section1]
name = Default    ; In-line comment

# next line shows line continuation
modules = module.A module.B
          module.C

REM REM-style comment
[section2]
limit = 1000
```

Core Options

By default the core application options are read from `[pyana]` section of the configuration file. If the option `-C` *`name`* or `--config-name=`*`name`* is given on the command line then additional section `[pyana.`*`name`*`]` is read and values in that section override values from `[pyana]` section.

Here is the list of all command line and configuration file options availabale currently:

Short	Long	Config File	Option type	Default	Description
-v	--verbose	verbose	integer	0	Command line options do not need any values but can be repeated multiple times, configuration file option accepts single integer number.
-c *file*	--config=*file*		path	pyana.cfg	Name of the configuration file.
-C *name*	--config-name=*name*		string		If non-empty string is given then configuration will be read from section `[pyana.`*`name`*`]` in addition to `[pyana]`.
-l *file*	--file-list=*file*	files	path		The list of input data files will be read form a given file which must contain one file name per line.
-n *number*	--num-events=*number*	num-events	integer	0	Maximum number of events to process, this counter will include damaged events too.
-s *number*	--skip-events=*number*	skip-events	integer	0	number of events to skip
-j *name*	--job-name=*name*	job-name	string		Sets job name which is accessible to user code via environment method. Default name is based on the input file names.
-m *name*	--module=*name*	modules	string		User analysis module(s). Command line options can be repeated several times, configuration file option accepts space-separated list of names.
-p *number*	--num-cpu=*number*	num-cpu	integer	1	Number of processes to run, if greater than 1 then multi-processing mode will be used.

User Module Options

For every user module the configuration file may contain one or more configuration sections. The section header for the user module has format `[module]` or `[module:`*`name`*`]`. When defining the user modules either with `--module` command line option or `modules` configuration file option one can optionally qualify the module name with a colon followed by arbitrary single-word string. Without this optional qualification the framework will load the user module and will use the options from `[module]` section to initialize the instance of the analysis class (as explained in Initialization section). If, on the other hand, the qualified name is used then the framework will initialize the instance with the options combined from the sections `[module]` and `[module:`*`name`*`]` with the latter section overriding the values from the former section. One could use several qualified forms of the same module name to produce several instances of the analysis class in the same job with different options.

Here is an almost identical example from Initialization section above which illustrates the inheritance and overriding of the user options:

pyana.cfg

```
[pyana]
modules = mypackage.myana mypackage.myana:wide

[mypackage.myana]
lower = 0
upper = 100
name = default

[mypackage.myana:wide]
; 'lower' option will be reused from [mypackage.myana] section
bins = 1000   ; this overrides default module value
; two options below will override [mypackage.myana] values
upper = 1000
name = wide
```

Analysis Tools

Initially there is only a small set of the analysis tools available to the user modules, but with time it will certainly grow. this section describes all the tools that we have today.

Writing Files

one of the simplest tasks that can be done is to write a subset of data to a file and then analyze those data with some external application such as Matlab. Writing data to files in Python is sufficiently simple once you know the format of the in-memory data and output data. If output data needs to be in ASCII format one can use Python `print` statement to format the data and write them to file. For binary data things could become more involved but there are multiple modules that deal with this problem such as `struct` and `array` Python modules.

One significant complication comes from the multi-processing capabilities of Pyana. With multi-processing enabled jobs runs in many processes with each process analyzing only a subset of the data set. At the end of the job the output files from all independent processes needs to be merged into a single file. Depending on the format of the output files merging can be either very easy, or very hard, or impossible. Pyana supports one simple merging mechanism for files when the files from all processes are copied into a single output file, very much like `'cat file1 ... fileN > file'` command does. The order in which files are copied is not specified, so if the order is important some additional processing may be required. To enable Pyana merging mechanism one needs to use a special construct when opening output file from an analysis code. Instead of plain `open(...)` or `file(..)` functions one needs to use `env.mkfile(...)` method with the same arguments. In this call a temporary file will be created somewhere (most likely in `/tmp` directory) with a unique name. The function returns a regular Python file object which can be used with all standard tools. At the end of the job Pyana will collect the names of those temporary files and merge them together into one file with the same name as was given to `env.mkfile(...)` deleting all temporary files. this special method is safe to use even when running in a single-process mode in which case it is equivalent to regular `open(...)` method so there is no unnecessary copy involved.

When this simple merging procedure is not sufficient users need to do merging themselves (if at all possible). In that case the file in every process should be opened in a standard way using `open(...)` but the name of every file should be made unique. One simple way to generate a process-specific name is to use method `env.jobNameSub()` which returns a regular job name followed by dash and a process number ranging from 0 to a number of sub-processes. When this method is called in single-process job it returns regular job name. User can also construct arbitrary process-specific file name using the sub-process ID returned from `env.subprocess()` method.

Histogramming

At present we use Python interface to ROOT. The main interface for creating new histograms is a special histogram manager object which is responsible for histogram bookkeeping in Pyana jobs. This object is accessible to user code through the method `env.hmgr()`. The object has several methods for booking new histograms such as `h1d(...)`, `h2i(...)`, etc. For detailed description of the methods and calling conventions consult Reference Manual. Filling of the histograms is performed through the methods of the histograms objects, the Reference Manual has links to the relevant documentation.

The histogram manager is also responsible for merging of the histograms from all sub-processes at the end of multi-processing job. As the result of the job there should be just one file with the name "*job-name*.root" in the current directory independently of whether the job runs as single process or multiple processes.

Here is a brief example of booking and filling of few histograms in user module:

mypackage/src/myana.py

```
# user analysis class
class myana(object):
    def beginjob(self, event, env):
        # get histogram manager
        hmgr = env.hmgr()

        # book histograms, store references
        self.hist1 = hmgr.h1d('energy', 'Energy Distribution', 100, 0., 1.)
        self.hist2 = hmgr.h2d('energy vs something', 'Histo2 title', 100, 0., 1., 100, -20., 20.)

    def event(self, event, env):
        # fill histograms
        energy = ...
        something = ...
        self.hist1.Fill(energy)
        self.hist2.Fill(energy, something)
```

SciPy Algorithms

Few data classes such as `camera.FrameV1` and `acqiris.DataDescV1` present their data as NumPy arrays. There are several packages out there that implement efficient algorithms working with NumPy arrays. Probably one of the most widely used packages is SciPy which is a collection of various types of algorithms including optimization, integration, FFT, image processing, statistics, special functions, and few more. The rich interface and close integration with NumPy makes it a good candidate for use in user analysis modules.

SciPy is installed as a part of the standard LCLS analysis releases. SciPy interactive features and Matlab-like API make it attractive for interactive use, but it can also be efficiently used in batch analysis jobs. There is extensive on-line documentation of both NumPy and SciPy.

Here is very brief example of using SciPy for integration of experimental data:

mypackage/src/myana.py

```
from scipy import integrate

class myana(object):

    def event(self, event, env):
        # Get Acqiris waveform object
        ddesc = evt.getAcqValue( "AmoITof", channel, env )
        wf = ddesc.waveform()
        ts = ddesc.timestamps()

        # integrate it using Simpson's rule
        integral = integrate.simps (wf, ts)
```

Multi-processing

The framework can be run in single-process or multi-process mode, by default everything runs in singe-process mode. In multi-process mode analysis job spawns a number of processes all running on the same host. In that case framework is also responsible for distributing individual events to multiple sub-processes and collecting and merging the results of the processing at the end of the job.

Framework handles few data types specially. For example EPICS data which is a part of the vent data does not appear in every L1Accept transition but every sub-process needs to have an access to current value of EPICS variable. So for EPICS the framework reads EPICS data from every event and accumulates current state in a separate structure. This structure is made available to all sub-processes as a part of the environment so the sub-processes need to access EPICS data through the environment and not reply on event data.

Every sub-process receives only a subset of all L1Accept transitions. Right now there is no control of the sequence and order of the events distributed to individual sub-processes. This fact could affect the type of the analysis that could be done on the data (e.g. event-to-event correlation may be hard to implement) and also the type of the result that analysis can produce in multi-processing mode. At the end of the analysis job the results of individual sub-processes may need to be merged together to produce a single result of the whole job. Frameworks currently knows how to merge two types of data:

- data written to a file which has been open with `env.mkfile()` method, the files from sub-processes will be concatenated together in no specific order
- histograms created by the histogram manager

For other types of the data it would be a user responsibility to store it in some location and then do manual merging after the job is finished.

Writing User Modules

Preferred way to run user analysis code is to create a separate package for each user and store all user modules in `src` directory in that package. If you plan to commit you code to repository then the package name must be unique and probably include your user name (or experiment name). To create an empty package run this command (it implies that analysis environment has been set):

```
% newpkg expname_ana
% mkdir expname_ana/src
```

Replace `expname_ana` with the true name of the package. This will create a directory `expname_ana` with a special `SConscript` file and `doc/` subdirectory containing `README` and `ChangeLog` files.

There is a special template for user analysis modules and to create a module based on that template one can run the command:

```
% codegen -l pyana-module -o expname_ana/src expname_ana my_ana
```

Replace `expname_ana` with the name of the package you created above, and `my_ana` with the name of the module. The command will create a file `my_ana.py` in the directory `expname_ana/src` which will contain basic structure of the analysis module, you will only need to fill it with some code.

There are few simple examples of the user analysis modules located in the `src` directory of `pyana_examples` package. These can be used as a basis for writing your own modules. To see the examples extract `pyana_examples` package:

```
% addpkg pyana_examples
```

and browse through the `src` directory for `*.py` and `*.cfg` files.