Documentation of the HSRL python data processing code

Brad Schoenrock

February 3, 2017

Abstract

Documentation - what a neat concept.

The purpose of this document is to aid in my (and others) understanding of the inner workings of the HSRL data processing taking place in the practical operation of the HSRL by NCAR. The goal of this work is to develop maintainable software in C++ to replace the current Python framework which is poorly understood and maintained.

Contents

0.1 Introduction

The High Spectral Resolution Lidar (HSRL) is a system that can be deployed on the National Science Foundation/NCAR HIAPER Gulfstream V (GV) aircraft or in a customized shipping container to study atmospheric phenomena around the world. The Gulfstream V High Spectral Resolution Lidar (G5-HSRL) is an eye-safe calibrated lidar system that can measure back scatter cross section, extinction and depolarization properties of atmospheric aerosols and clouds [**?**].

The HSRL was originally designed and built by the University of Wisconsin Lidar Group. This group also provided some data processing code in MATLAB, and later switched to python to suit its needs. One function of the UW processing is that it provides real time and archived data which can be viewed at the GVHSRL in Madison WI website. Scientists have expressed interest in keeping this functionality and on improving the error estimation on the extinction as a measure of data quality.

Chapter 1

Python

To begin there is a wiki describing how to set up the python environment nessicary for the execution of the python processing code [?].

The HSRL_Python package can be checked out from the NCAR GitHub here:

https://github.com/NCAR/hsrl_python

Also of interest are the following repositories:

https://github.com/NCAR/hsrl_dpl_tools
https://github.com/NCAR/hsrl_configuration
https://github.com/NCAR/hsrl_instrument

I have a personal copy with some edits here aimed at probing for understanding:

https://github.com/BradSchoenrock/HSRL_python_Brad

1.1 the python data processing

The data processing begins with some control scripts hsrl_dq and cset-cfradial.py which can be used to generate plots and create some Pseudo CFRadial files respectivly. These call functions in the python codebase which start the processing from Raw NetCDF files which are located on /scr/eldora1/HSRL_data. The first functions called are in maestro/rti_maestro.py.

1.1.1 write_netcdf

This function sanatizes some inputs, and acts as a switch for different lidar setups. Ultimitly it calls another function called makeNewNCOutputFor and pasess it the format of files and directory locations.

1.1.2 makeNewNCOutputFor

This function takes in a framestream (which is what?) as well as format and file location for output.

There are some try except statements in here which are not clear why they are needed.

This function has some strange lines of python which confuse me.

Line 1735: if True:#not cfradial: then it does stuff. Why the if true statement?

Line 1753: loop which does nothing in it (just a pass), but seems to be doing all the writing. The resulting CFRad file has no actual data in it if this is commented out. In the for loop it is itterating over an instance of the class dpl_netcdf_artist which is found in lg_dpl_toolbox/dpl/dpl_artists.py.

This is the line for reference.

for f in artists.dpl_netcdf_artist(framestream,template,outputfilename=filename,output=output,fo

pass

1.1.3 dpl_netcdf_artist

This is a class being called by makeNewNCOutputFor and has several functions. __init__ for initialization, acceptableMetaframe, __opentemplate, render, and __del__. All of these functions seem nessicary for getting full and properly formatted output, some of which are being called multiple times for unknown reasons.

1.2 HSRL2Radx

1.3 Hawkeye

Chapter 2

Commands

Some usefull commands saved here for recolection
————————-

Raw NetCDF files to Pseudo CFRadial files
Python code

```
./scripts/csetcfradial.py
```

Pseudo CFRadial files to CFradial
Hsrl2Radx

```
/h/eol/brads/git/hsrl_configuration/projDir/ingest/params
Hsrl2Radx -params Hsrl2Radx.test -debug -f /tmp/gvhsrl_20150729T1930_20150729T2000.nc
```

CFRadial to HawkEye display

```
/h/eol/brads/test/display/params
HawkEye -params HawkEye.hsrl-brad
```

See the contents of a NETCDF file

```
ncdump gvhsrl_20150729T1930_20150729T2000.nc — less
/opt/local/bin/ncdump gvhsrl_20150729T1930_20150729T2000.nc
```

Chapter 3

Misc emails

Hi Ed, Is there an option to write out the raw photon counts from each channel when writing a netcdf of processed data? I see that it writes out the profile counts (time integrated as I understand), but what about the time resolved counts?

Can you point me to where in the code write_netcdf() resides?

Thanks, Matt

Hi Matt,

The fields stored in the netcdf are configured by a CDL template file.

I think NCAR usually uses the template lg_dpl_toolbox/config/hsrl_cfradial.cdl . The fields in the file are bound to the runtime fields using the dpl_py_binding attribute. It does look to me like the counts are in the template (e.g. netCDF field molecular_counts' bound to python's rs_mean.molecular_counts). The parts in the rs_mean substructure match the same time axis as the inverted data, but are already corrected for dark count, afterpulse, and pileup. There is also rs_mean.raw_molecular_counts for just pileup-corrected counts. The true raw form doesnt exist on the final grid, or with a real altitude axis (different from the native range distance-from-instrument axis). The rs_mean structure contains resampled data either by average or sum (photon and energy counts are always summed, making it statistically more sound).

Does this help?

Joe and Ray: As I am trying to debug makearchiveimages.py, I realize that I simply dont have any way to proceed w/o more explanation and documentation of how the code is intended to work. Ive been examining this code for days, and stepping through it with the WingIDE debugger (which

I highly recommend) and Im still quite baffled how this code is supposed to work.

Correct me if Im wrong, but I dont think that anyone else but Ray and Joe can maintain a significant amount of this code in its present state. There needs to be much better documentation so other programmers can comprehend and maintain this code.

I strongly suggest for all the classes in Picnic and DplKit (and in the older HSRL python code)

Each class needs a description of its intended use. Each class method needs a description of its purpose and what object(s) it returns. Each parameter of each method needs to be documented. Each complex class should have examples of use along with test cases (using unittest, as was done for dplkit/simple/filter.py)

We also need a more high level descriptions of how these classes are used together.

For example, in dplkit/role/decorator.py, I have little idea what any of the classes are trying to do, or what problem they are trying to solve.

1) what is a nested frame description and why would I want/need to use it?

2) what is the scopelockraiser protecting against?

3) What does autoprovidenested accomplish?

4) What does autoprovide accomplish?

5) What do the methods has_requires” and has_provides” accomplish?

6) Why does __autoprovidesbase.__call__ contain both __fakeIter__ and __iter__ methods? What problem does this technique solve?

Elsewhere in the code:

7) In DplKit/python/dplkit/role/librarian.py, the documentation for the search() method is very helpful in explaining how librarian, zookeeper, narrator interact. 7a) It would be very helpful if this description were expanded to discuss the role of artists”

7b) Is this description consistent with how the hsrl.dpl.dpl_hsrl class is coded? dpl_hsrl is derived from aLibrarian, but then creates an instance of HSRLLibrarian, which is also derived from ALibrarian

8) Why does HSRLRawNarrator.read() yield pathnames for files that can’t possibly contain the requested interval? What’s the point of generating/yielding data that wasn’t asked for?

I am running makearchiveimage.py with the arguments ”gvhsrl 2013.01.02.13-2013.01.02.18” However, HSRLRawNarrator.read() yields: ’path’: u’/scr/eldora1/HSRL_data/20 ’width’: datetime.timedelta(0, 3960), ’start’: datetime.datetime(2013, 1, 2, 2, 0, 2), ’filename’: u’gvhsrl_20130102T020002_data_fl1_pol.nc gvhsrl_20130102T020002_data_fl1_pol.nc can not possibly contain data between 2013.01.02.13-2013.01.02.18

9) In lg_dpl_toolbox/filters/substruct.py, I have no way of knowing what any of the classes are supposed to be used for.

Joe Garcia writes:

Ive noticed that in many cases, the RTI Maestro object is whats being used to interact with the processed data. While this is the oldest and most interactive interface for processing HSRL data, it isnt intended to be the most flexible or efficient. Its mostly geared as a lab environment, which is why it does so much work at initialization to create images and hold on to intermediate data with the intent of allowing the user to decide what to do next.

Underlying all of this are objects that we call Actors, in the grand scheme of a Data Pot Luck, or DPL. For an example of what these are capable of doing, take a look at hsrl/dpl/dpl_live.py . Its a rudimentary but simple script that creates and uses actors the same way RTI does, but for a single use-case instead of one that is general or adaptive. It initializes the HSRL processing engine, and feeds that into an artist to make images, and iterates upon this continuously to create a much more efficient form of Rtis loop() command.

In this way, you can replace the image artist with a netcdf artist, provide a start and end time (so its not open-ended), and create a batch-process script without as much overhead as the Rti implementation. Specifically, a deliberate script could skip making images and in-memory data accumulation, instead opting to dump the data chunk-by-chunk into a CFRadial file. Short of a bug or missing feature, additional manipulations of the data can be done by injecting additional filter actors, in the same way similar the dpl_window_caching_filter is used in the script, which does nothing more than maintain a rolling cache of the last couple hours, storing new frames while it drops the oldest. A filter simply consists of an object that iterates on a source actor, operates upon each frame, and yields the result for downstream actors to consume.

The interfaces between actors arent very well defined or mature, but it is already useful to us.

========================== More explanation about DPL frame streams

Joe,

1) with the DPL frame streams, a 'simple' frame would be a single time record with a flat namespace. A 'compound' frame is multiple time records

in a flat namespace. on top of that, we have the idea of a 'complex' frame, where different compound parts are potentially at different time or altitude scales. These are generally frowned upon, but it is how the HSRL processing currently operates, which it does by 'nesting' structures within the frame structure. Breaking apart the monolithic black box from making these monolithic complex frames is something I've been progressively working to do, to the point of setting traps in the code to identify places where monolithic code style or frames are required, or else it would crash.

2) first, this prevents multiple threads from trying to access the autogenerated 'provides' dictionary (see points 3,4) while it is being generated. We still have a problem that sometimes coding glitches will cause this generation to short circuit silently. I still have more to understand on why this happens, and why I can't get the resulting exception to properly reveal itself

3,4) part of DPL framestreams is to expose at initialization time a dictionary describing the content of a frame. the specific protocol of this hasn't been decided yet, but at the very least, every value in the frame should have an entry in the provides dictionary. because the HSRL code dynamically changes its content depending on the deployment and instrument, knowing this structure a priori has become a moving target and near impossible. the autoprovide decorator will cause the provides dictionary to be created at the moment it is requested by running an iteration of the framestream, and describing the resulting frame into a dictionary. the autoprovide decorator allows only for a 1-level frame, with a given object type for its frame it is permitted to descend. autoprovidenested allows for multiple object types and multiple layers, with any instance of the identified object types to be descended into as a nested frame.

5) this identifies a dpl primitive abstract class to have either a provides dictionary, requires dictionary, or both. the 'requires' dictionary is a future plan for DPL objects to document, and potentially validate automatically that the pipeline has the fields it needs and will work correctly.

6) when the autoprovides starts an iterable instance for generating the provides, it retains that instance and the frame it had used to generate the description. when the host object is then iterated on for the normal processing, it checks if it has a retained iterable, and if it does, yields the already generated frame, and resumes the retained iterator. this avoids wasting time generating the first frame for a second time, which for HSRL data can be relatively rather time consuming.

7b) no, this is not consistent with how it should be. My technique of having actors return actors is something Ray has voices opposition to, but has allowed it because another method of being able to execute a 'search' to yield an object that can be fully iterated on (meaning all parameters and runtime are accessible there), while still having actor-like meta data hasn't been described yet.

8) the librarian usually errs on the side of inclusive behavior, leaving it up to the data consumer to ignore parts it won't or can't use. It is probably better that it omit records that couldn't possibly be useful, but if a data gap exists at the start of the interval, returning the prior record that is outside of the requested window may be used by an interpolator to extrapolate missing records, if such a dataflow were in place.

9) this is part of documentation i've been needing to get back to, but never have any time to focus on actually getting it done. as these classes get better implemented and documented in generally useful ways, they may get promoted to be a tool in dplkit.

Please understand that these codebases are potentially under very dynamic development from week-to-week, often opting to get code that does a specific task, or progressively facilitates getting toward a goal. Because of that, the code is always growing faster than the documentation, so deliberate documentation would often become as vestigial as some code already has. In the future, there's already been demonstrated desire to start a 'release' branch of tested code, which I need to find the time to start executing and gathering test cases for.

The end goal currently is to get the HSRL code into a state where functional parts of the science code are decoupled, with clear scope to specific tasks, and in the process get DPLkit a trial by fire with the complex nature of the HSRL processing, figure out what does and doesn't work, and simplify the parts that can be matured into describable interfaces and protocols that aren't as spaghetti as they are currently. As that gets more general, we'll be able to have a more approachable set of modules that can be assembled to a processing engine, including external modules and instruments as scientists demand.

– Joe Garcia Instrumentation Technologist UW Lidar Group University of Wisconsin - Madison

---