blimpy Test Coverage

Lukas Finkbeiner

May 1, 2020

1 Introduction and Background

Test coverage is good. We want more. Also documentation. Keep issues alive and open. Also, test centralization.

Some code is unfinished. We do not want this to bring our numbers down.

At various points the tone of this report will shift toward advice, in case some future intern wishes to contribute.

2 Methods

I isolate the script that I want to work on.

The blimpy repository includes two scripts necessary for running the complete suite of tests: download_data.sh and run_tests.sh. In this repository I have included two pared-down bash scripts, fast.sh, fail.sh. fail.sh is the fastest, but can only be used to check whether new test cases pass. fast.sh is the next fastest, and will check the coverage of all scripts in the blimpy directory. Simpler bash scripts have the advantage of lower overhead, which allows the programmer to easily switch between writing tests and examining their impact. The principle downside of this pared-down approach is that fast.sh ignores repository warnings regarding which scripts are to be tested. Consequently, the programmer will receive coverage results about scripts in the directories 'calib_utils' and 'deprecated' (observe the absence of such scripts from the tables below).

I also isolate individual methods, at least for guppi

3 Coverage Results

The following table records the initial (as of February 2020) state of test coverage. It represents a view similar to what one will receive from running run_tests.sh in the shell. I have shortened the script names unless the directory is necessary for disambiguation.

Script	Statements	Misses	Coverage (%)
__init__	25	8	68
bl_scrunch	45	25	44
dice	103	48	53
ephemeris/__init__	3	0	100
compute_lsrk	28	0	100
compute_lst	15	4	73
ephemeris/config	9	2	78
observatory	41	12	71
fil2h5	42	21	50
guppi	271	137	49
h52fil	42	21	50
io/__init__	2	0	100
fil_writer	41	8	80
file_wrapper	397	101	75
hdf_writer	87	20	77
sigproc	157	28	82
match_fils	74	61	18
plotting/__init__	7	0	100
plotting/config	11	2	82
plot_all	69	5	93
plot_kurtosis	16	2	88
plot_spectrum	39	6	85
plot_spectrum_min_max	44	5	89
plot_time_series	26	3	88
plot_utils	28	4	86
plot_waterfall	28	4	86
utils	65	0	100
waterfall	228	57	75
TOTAL	1925	581	70

In this next table, I report the details of scripts for whom testing has increased. The precise numbers are accurate as of the writing of this report.

Script	Misses	Decrease	Latest Coverage (%)
bl_scrunch	25	12	72
dice	38	10	63
fil2h5	8	13	82
guppi	77	60	72
h52fil	8	13	82
file_wrapper	66*	17	83*
match_fils	0*	10	100*
waterfall	51	6	78
TOTAL†	1931	403	79

The asterisks indicate fields for which I am factoring-in the deprecations and exclusions that I mention in the 'Concluding Recommendations' section. The dagger indicates that the total includes numbers from the tests which were not shown (because their coverage had not increased).

4 Current Efforts and Obstacles

The guppi file-handling script guppi.py represented a thorn in the side of this project. Interacting with this script was especially slow. On many occasions, even running tests on this script in isolation, the code-coverage routine would experience a time-out, killing the test session. In visually analyzing the script, I have not come across major areas which could be sped up to resolve this issue. However, virtually all of the user-facing methods (e.g. histogram) deplete the file. As a consequence, each test case must create a new guppi-handler object from scratch. This places a major strain on the hard disk, the slowest piece of a computer.

One ongoing project is the accessing of command-line tools for testing purposes. Several scripts such as bl_scrunch and h52fil.py come packaged with methods which allow the user to analyze and plot data by passing in arguments directly from the shell. All such functions include a line which relies on an external dependency to parse console inputs. guppi.py has such a tool, but it also features a parameter in the method should the programmer wish to access the tool from a script. I have been modifying the blimpy scripts so that more of the command line methods have this feature; it allows the test scripts to easily pass in different inputs, verifying the rigor of the code.

Some issues are broader in scope and arguably are not well-suited for a nuts-and-bolts project such as this, or at least for an intern with less experience with the aims and priorities of the repository as a whole. For example, virtually all of the test cases rely on a set of Voyager data in the .h5 and .fil formats. Because we are essentially using a single source for all tests, there are many if statements which are not executed because they bifurcate around properties specific the file's data type (small integers, large integers, or floats), number and sizes of channels, etc.

This gets into an issue of diminishing returns. Introducing entire additional data sets will undoubtedly slow down the testing process, even using the isolation approach that I

described in the methods section.

5 Concluding Recommendations

In conclusion, we need some deprecation notices (match_fils.py). We also need to exclude some methods from our tests, because they raise NotImplementedError (file_wrapper.py).

Guppi sucks!! I HATE it! Especially because we keep getting killed. File depletion is a major nuisance. And, hard-coding represents a total off-shoot of this project.

We need more documentation all-around. Some methods are fairly self-explanatory, they just run one function and make sure that there are no runtime errors. Other methods, especially those involving command line options, are a bit more involved.

Do we want more rigorous checks for accuracy? The development process will continue to slow down. Error checking? Not likely, but it is something to think about...