blimpy Test Coverage

Lukas Finkbeiner

May 1, 2020

1 Introduction and Background

The current test coverage of the blimpy repository, as reported by the dedicated shell script in the tests directory, is approximately 70%. This represents the coverage specifically of a subset of scripts, marked as active. The most obvious example of this is the deprecated folder: scripts which are being phased out are no longer tested for compatibility.

Our current figures for code coverage represent an underestimate because there are promising avenues for further exclusion of code. The two primary examples of this are match_fils.py and file_wrapper.py. The code of match_fils.py is largely dominated by a command-line utility which cannot be used outside of Green Bank machines. However, it is still tested by the testing script, so we generate artificial statement misses. Danny Price has recommended that this code be marked for deprecation. In the FilReader class in file_wrapper.py, there are three unimplemented methods: read_all, read_row, and read_rows. However, it is more accurate to call these unfinished since there is code in the body which is inaccessible because a NotImplementedError has been raised at the beginning of each.

Code exclusion is important, but the main purpose of this report is to document constructive (rather than destructive) improvements in test coverage. The current (and perhaps indefinite; see conclusion) emphasis of the tests appears to be the catching of run-time errors (i.e. script crashes) rather than logical errors (i.e. incorrect results). As a result, code coverage may sometimes be increased in such quick manners as simply calling a plotting routine. In other cases, it can require creative combinations of arguments to ensure that a routine raises the appropriate error.

Finally, there is another primary objective of this project which cannot be concisely documented here, but which is still open and significant to the maintenance of the repository. While surveying the existing tests, I discovered a general limitation in the quantity of specific documentation. I have been documenting my new test cases as I implement them. Documentation in the test cases benefits all future blimpy programmers. If a particular adjustment to the code causes the tests to fail, we want the programmer to immediately know which action failed. The shell will automatically report the name of the test method that failed, but without documentation in the method, one would have to study the code to learn or even relearn its conceptual purpose.

2 Methods

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).

To further decrease the wait time (between writing and testing a test) I practice isolation on multiple layers. First, I move to a separate directory all scripts which I am not currently editing, so that the shell script will only run one test script at a time. For some of the slower scripts such as guppi.py (see 'Current Efforts and Obstacles') I also isolate all test-functions except for the one I am currently developing. While the coverage numbers will decline in isolated testing environments, the programmer need only seek that the coverage increases *compared* to point at which the isolation began.

On this topic, I have two recommendations for future test development. One is centralization, of which file_wrapper.py is the worst offender. There is a dedicated test script, but if one runs this script by itself, the coverage for file_wrapper.py decreases dramatically (by contrast, most other scripts will maintain roughly the same level of coverage for their tested script if you exclude the remaining tests). Test centralization is important because it facilitates the isolation approach. When isolating the file_wrapper test script, the file_wrapper coverage decreases so dramatically that one would lack a good sense of which tests were redundant; for example, one could add a function and notice a coverage increase, but that increase could evaporate when all tests are run together (some unnoticed lack of synergy between test cases). I have expanded test_file_wrapper.py to an extent which is not reflected in the coverage increases because I worked on enhancing the centralization of that script.

The second recommendation is compartmentalization. Within a testing script, there are several instances of single methods testing multiple features (for example, h5 and guppi file handling). If we focus on greater quantities of smaller methods which test discrete features, test failures will be easier to debug.

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. We are essentially using a single source for all tests. Consequently, there are many if statements which are not executed because they bifurcate around properties specific to 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...