

Development of a Software Environment and User Interface for the Analysis of Accelerometer Data

Nguyen Hoang Anh

hoanganh.theodore@icloud.com

Physics Dept., John Abbott College, Sainte-Anne-de-Bellevue, QC., Canada.

Instructor: Chris I. Larnder, John Abbott College, QC.

Research Supervisor: Roberta Silerova, John Abbott College, QC.

May 29, 2020

Abstract

The paper details how a software development environment needs to be set up through the utilization of readily available accelerometers in modern personal mobile devices (i.e. iPhones, Android phones, etc.) as affordable data collection instruments in an education experimentation setting, in which the precision is laissez-faire and importance is placed on understanding and familiarizing with the concepts of classical mechanic physics. The research into the applications of such accelerometers yielded positive results. The project employed the Python programming language and its various open-source libraries, i.e. keras, matplotlib, numpy, tensorflow, to build programs that can automatically process the accelerometric data gathered from the mobile devices and present them meaningfully for the end-users. To date, the project has been able to generate 16 Python modules handling low-level I/O tasks to high-level data augmentation and 28 Python scripts for demonstrating operations on data, e.g. using stochastic gradient descent (sgd) and adaptive movement estimation (adam) optimizers to predict movement of an object in 3D space. Our prediction model was able to retrieve the radius (r) for stochastically generated and real-time data then output the results in 2D and 3D graphs. Although the research mainly aims to utilize consumer electronics, it also wants to contribute to the computer science and physics communities with plans to make the tools created within the project open-source and available for everyone. In accordance, the original project was structurally overhauled to meet industry standards in packaging, as well as added documentations based on the Agile development paradigm to promote scalability and facilitate community-driven contributions. The project plans to release weekly cumulative updates.

Keywords— project management, version control, development environment, programming in physics, learning programming, pedagogy, source code maintenance, best practices, Python, machine learning, troubleshooting, project structuring, accelerometers, physics, computer science, software engineering

Forewords

This paper is heavy in computer science and software engineering content. Thus, it does not strictly follow the conventional science style for reports and research papers. This paper focuses on the process of creating a healthy software development environment for the project that can be grown more easily into a crowd-sourced programming project as well as achieving a high level of user- and developer-friendliness. The paper tries to explain in detail the concepts in programming for an audience who has yet to acquire the taste for programming in general.

Introduction

“Computer science, software development, software engineering” - these are the contemporary buzzwords in the job market in recent years. The field grows at an exponential rate with traditional law firms, accounting firms, stock markets, etc. being flooded with the use of internally-developed software for analysis, data management, and monitoring. Such expansion allows the trickle down effect to take place and ripple across the fabric of society. The availability and accessibility of the knowledge and tools previously known and used by industry experts and professionals are imparted to a wider range of users. This major propagation has opened doors to students of all ages to participate in this next industrial revolution. Now, as early as middle schools, student-run programming projects are rapidly emerging. This research project on the extensive applications of accelerometers in consumer mobile devices using Python data processing coupled with classical mechanics physics was no exception to this mass effect. This project benefited from the open-source tools the same way they are used by leading professionals. However, the project, being contributed by a group of students, was prone to disorganization which might lead to slow development progress and hard to maintain software. This paper’s goal is to detail the steps taken to identify and mitigate such problems, then hopefully would serve as a model to be referenced by others to reach an end goal of perpetually building better software.

Development

1. Infrastructures

The infrastructures of a software can determine its flourish and success. These infrastructures involve basic building blocks of a software: what language it is written in, where its source code is stored, how the functionalities are documented, what interactive framework is used for communication between the user and the program, and how thoroughly and efficiently contributions to the code are reviewed [Mar09].

The research project took a frame of reference when it came to building a solid infrastructure. Software architectural principles from Steve McConnell’s Code Complete [McC04] book on established software construction practices were relied on as base knowledge and then adapted into the needs of the project to avoid the practices becoming too much of a burden on a team of students. It was crucial to determine the ‘requirements’ of the project. Requirements are the intended functionalities and features expected by users of the software, which were collecting, formatting, analyzing, and processing of accelerometers recorded from different devices, as the recording method and the formatting of data differ for each device. Requirements were the precursors of specifications. These ‘specifications’, or ‘specs’ for short, were necessary for planning the pathway that the project would follow in the future. From the beginning, the project was planned to become open-source which meant that everyone could retrieve (download), use, and systematically modify the source codes to improve the programs. So, the programming language must be widespread, the source code needs to be hosted on a free and populous platform, documentations must be detailed, illustrated and accessible, and finally contributions to the project must be reviewed before being implemented.

With respect to the specs, a platform for development was decided:

- Programming language: Python (v3.7.x). It is an open-source, free and extremely powerful yet light-weight and modular language. It is currently the most popular programming language (as of this writing) with 31.17% on Google Search Index <http://pypl.github.io/PYPL.html>, so interested parties, be beginners or professionals, could start using and contributing to the source code right away without much of a learning curve. Python also optionally comes with *numpy* for scientific computing, *keras* and *tensorflow* for machine learning and deep learning, *matplotlib* for plotting graphs, all of which were used in this project. Another pros of Python is troubleshooting because Python’s popularity makes it effortless to find a solution to any error one might encounter.
- Hosting platform: GitHub is a freemium source code hosting platform that has tools specifically for collaboratively sharing, modifying, and contributing to a programming project. Though

freemium, users with a college-affiliated e-mail can register for the "Pro" variant which provides more diagnostic tools, vulnerability checks, and private repositories. Not only that, but it can also act as a small-scale data hosting platform for experiments. GitHub supports git which is a source code management tool every developer uses to collaborate.

- Documentations are written in markdown (.md) format. This format is automatically rendered by GitHub so it is displayed as a completely-formatted document whenever a user opens the file on the code repository. The syntax of markdown is intentionally made for technical writing: it supports text formatting, dynamic embedding for illustrative images, tabulation, as well as writing code blocks for demonstrations.

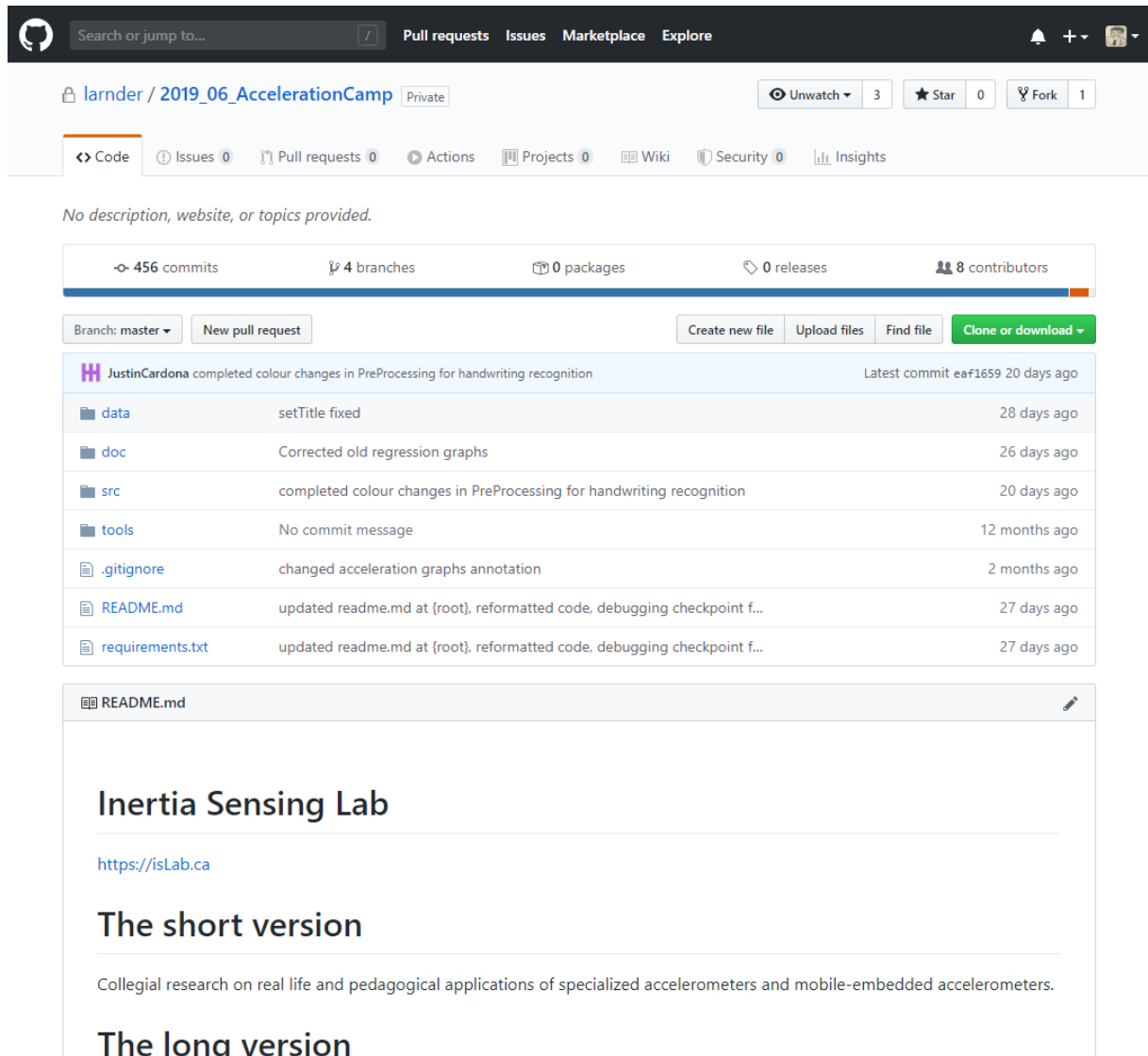
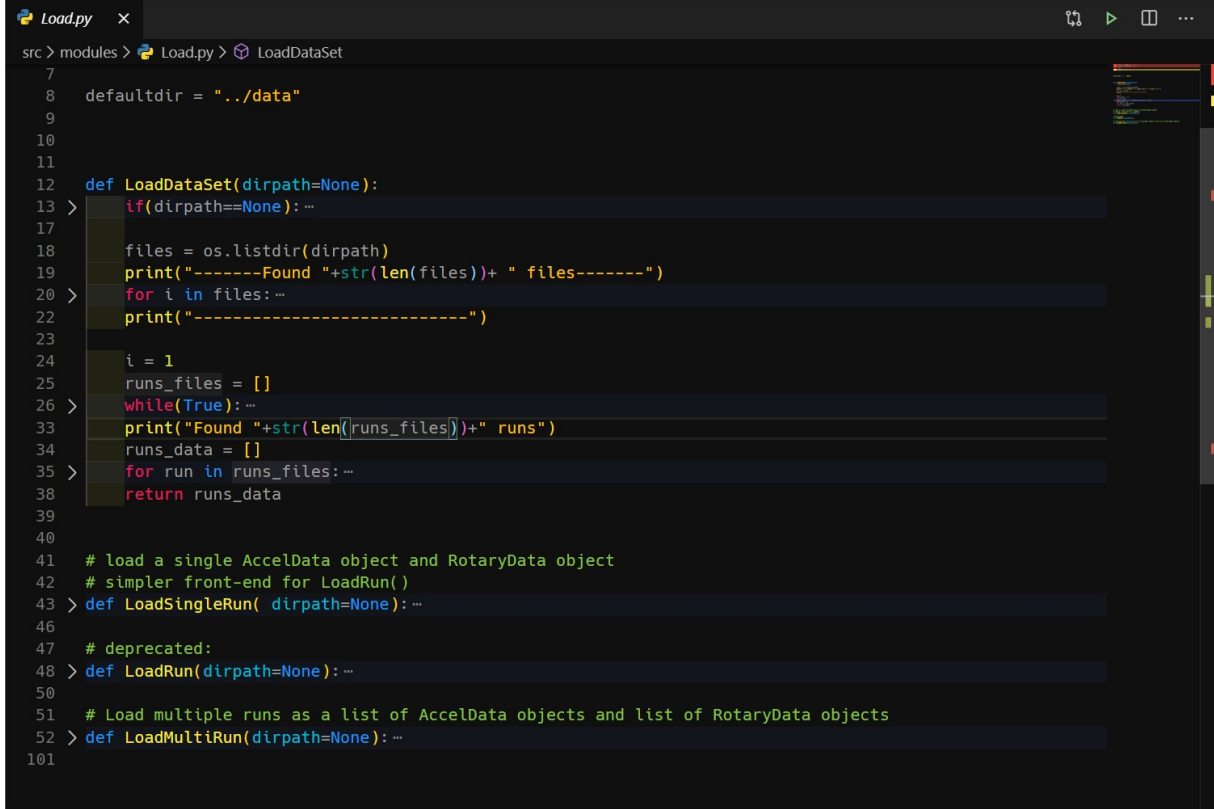


Fig. 1. Screenshot of project's Github repository showing part of the introductory *README.md* file automatically rendered by the GitHub repository hosting service as a legible document, as well as the root folder structure of the project with its own commit message and latest commit date.

2. Maintenance

Initially, there already existed a dozen modules in the project. These modules and tools had been developed around 6 months before the start of this research project; therefore, those modules needed to be tested for version deprecation with the planned infrastructures before being used and/or modified by team members. Testing was a 3-step process involving (1) naively executing (a.k.a. running) the modules and tools, (2) noting down breakpoints, warning, and errors, (3) resolving those raised issues and detailed documenting in a daily development note.

Once, initial testing phase had been concluded, the project moved onto the reorganization (i.e. refactoring in software term) phase. This was done intermittently with the development workflow. By refactoring that way, a holistic view of how the tools depended on and interacted with each other could be mapped, avoiding over-adjusting the project structure resulting in an incongruous development environment for the team. This phase involved not only the physical organization of files and folders, but also the organization of written codes. All existing lines of codes and newly-added ones were documented on what they do, what input parameters the codes took in, and what output value & data types the codes returned. See *Fig. 2* as an example.



```

src > modules > Load.py > LoadDataSet
7
8  defaultdir = "../data"
9
10
11
12 def LoadDataSet(dirpath=None):
13 > if(dirpath==None):...
14
15
16
17
18     files = os.listdir(dirpath)
19     print("-----Found "+str(len(files))+ " files-----")
20 > for i in files:...
21
22     print("-----")
23
24
25     i = 1
26     runs_files = []
27 > while(True):...
28
29
30     print("Found "+str(len(runs_files))+ " runs")
31
32     runs_data = []
33 > for run in runs_files:...
34
35     return runs_data
36
37
38
39
40
41 # load a single AccelData object and RotaryData object
42 # simpler front-end for LoadRun()
43 > def LoadSingleRun( dirpath=None):...
44
45
46
47 # deprecated:
48 > def LoadRun(dirpath=None):...
49
50
51 # Load multiple runs as a list of AccelData objects and list of RotaryData objects
52 > def LoadMultiRun(dirpath=None):...
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101

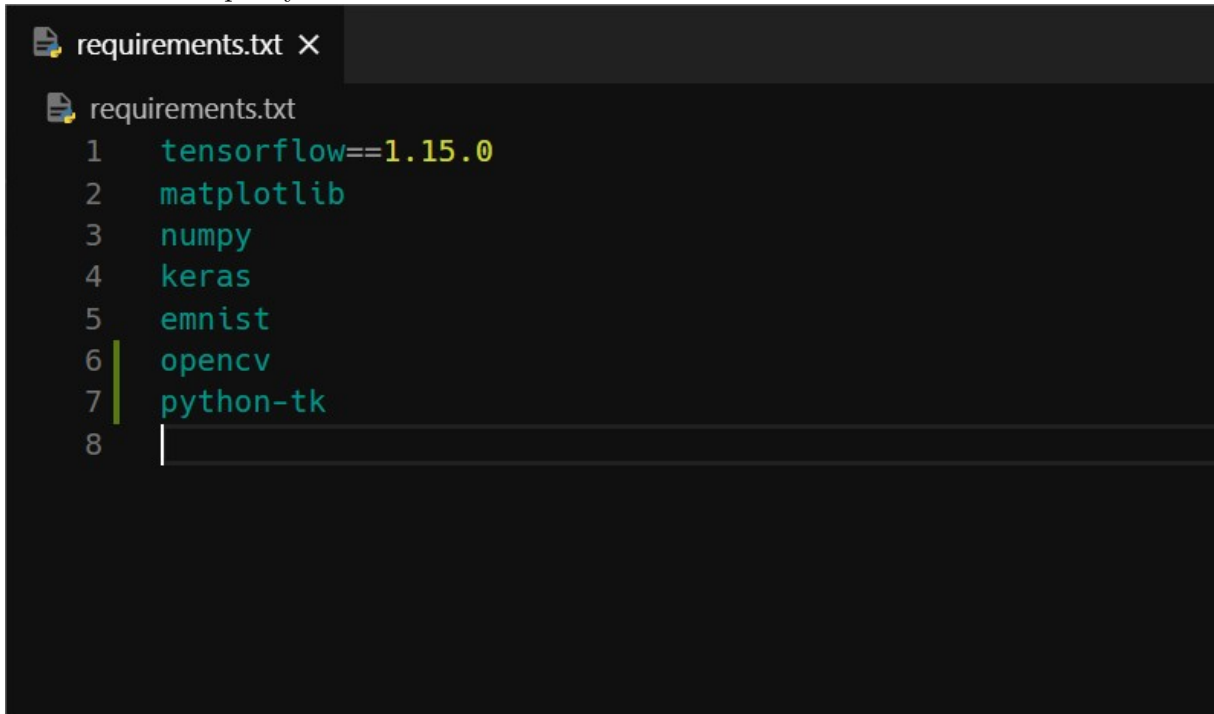
```

Fig. 2. Screenshot of *Load.py* module with collapsed code block. This is a normal layout of a Python module or script, depending on the context of the program. Functions are initiated by *def* followed by a meaningful name and the parameters that it takes as input in round brackets (line 12). Comments are initiated by a hash (number sign) and formatted in green color. Variables are implicitly defined by a name followed by the data that they store (line 8).

These maintenance actions were based on Agile software development principles [Mar09]. These principles included (1) naming scheme for, in increasing level of abstraction, variables, classes, functions, modules, and packages, among these were also the naming of data folders and documentation. These names had to be explicit, accurate, and descriptive in terms of what information it stored or what function it performed, eliminating the need for further comments. For example, the function to load a dataset is called **LoadDataSet** (See *Fig. 2*) and the local variable storing the loaded dataset is called **runs_data**. Then, where more complicated codes dealing with multiple functions at a time were written, comments were added to explain what those lines of code did. Functions and higher abstractions in the software were intentionally coded short and fragmented so that when revisiting them in the future for either modifications or deletions, the job would be less of a hassle. (2) The beauty of writing code lies in the notion of being able to identify edge cases that would break the software early on in the development process. Dealing with accelerometer data involved working with numbers, integrations, multiple degrees of derivations, and basic arithmetics. For example, Division by Zero was often the main cause of the program breaking or yielding cryptic numeric results when performing an operation. To eliminate such edge cases in the program operations, error handling and error catching were implemented in various stages of the software to either exit the process early and output logical results or trace back the original user input and display a message to troubleshoot. (3)

All future functions needed to be compared with existing functions to ensure that there would be no duplications with respect in what task those functions trying to achieve. It was highly encouraged to use already available abstractions to perform a task instead of creating a new one every time. (4) This finally led to the notion of garbage collection. The term garbage collection refers to the task of finding duplications, deprecations (no longer in use), and superfluous variables and eliminating them. Garbage collection helps shrink the source code size which allows for faster uploading/downloading of the source code and faster program executing time which could propagate exponentially when running a multi-functional program. (5) Although the team only met once per week, codes and data were added in smaller commits throughout the week. So, at the end of each work week, all codes and data were checked and merged onto the master branch one more time for quality assurance, as well as marking a checkpoint in the development process.

A neat built-in feature of the Python programming language is its package manager *pip* which comes with Python upon installation in the development environment. With this package manager, the developer can create a file called '*requirements.txt*' dictating all the Python modules used in the project, then have *pip* read the file and install all the modules. This was extremely useful because of the ability to select the version of the modules and have every other team member install the correct versions, all of which eliminated the need to manually install the modules and the risk of encountering deprecation or discrepancy errors across different versions of the modules.



```
requirements.txt X
requirements.txt
1  tensorflow==1.15.0
2  matplotlib
3  numpy
4  keras
5  emnist
6  opencv
7  python-tk
8
```

Fig. 3. The inside of the project's *requirements.txt* file. The module names are listed separately on each line with the option to specify which version needed via double equal sign followed by the version number.

3. Workflow

To collaborate and work on the same source code, the project team needed to create and abide to a unified workflow so as to avoid conflicts and discrepancies when modifying various interconnected tools of the software. This workflow needed to accommodate both the software team, where it came down to the minutiae of differences in code, and the experimental team, where data were being constantly added, modified and accessed. Since the project used *git* as the source code version control, understanding and adopting the standard git workflow was undoubtedly imperative. See **Fig. 4**.

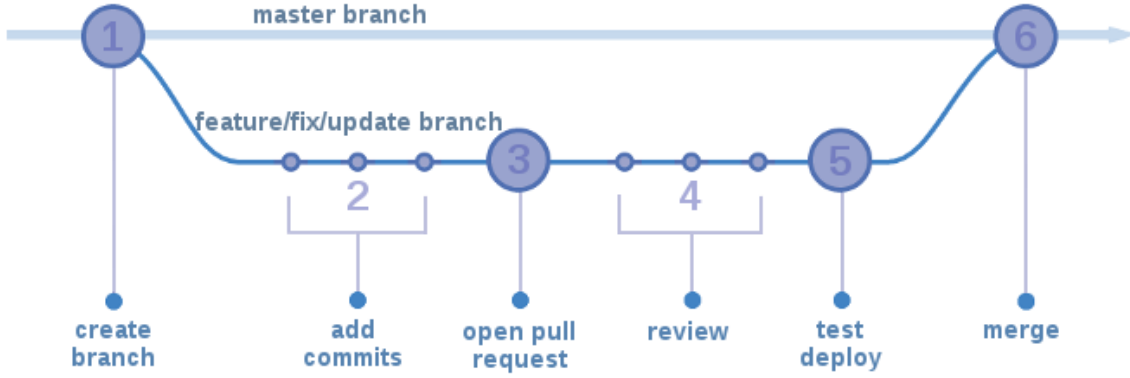


Fig. 4. Standard git workflow diagram. By ARCC @ University of Wyoming, 2018.

(The following paragraph refers to **Figure 4**)—The diagram shows the simplified overview of the git workflow. Again, *git* is a source code management tool and *GitHub* is the host of the repository on the cloud. A branch is analogous to the idea of a timeline of parallel universes where changes made are unique to that branch. Generally, there are two types of branch: master branch (on the cloud) and development branch (on the local PC). **Step 0** is implicit; this is where the team member has to “pull” incoming changes from the remote server to their local repository at the start of every development session. Then, to start adding and editing features, the team member has to create a branch (**step 1**) from the master. At this point, the development branch is the carbon-copy of the master branch. The team member would then work on this development branch (**step 2**) exclusively throughout the working session and commit changes. After completing the feature, they will open a pull request (**step 3**) which sends a message to the owner/maintainer of the project, letting the latter know that there are new changes coming in to inspect, test, and deploy (**step 4 + step 5**) the newly-added. After testing and troubleshooting errors with the features, the new code would then be merged (**step 6**) from the development branch to the master branch, thus concluding the development session.

For the experimental team, an adoption of the standard git workflow was not adequate as the team also dealt with multiple data sources and operated on large datasets. It was more of integrating the git workflow into the experimental team’s own workflow than having the team follow the git procedures. They would run experiments, collect the accelerometer data with whichever device they were using, and transfer those data off the device onto the development branch while renaming and reformatting the datasets according to rules preset by the whole project team. Finally, the experimental team would merge the data from their local branch into master branch at the end of their experimental session.

Results



Fig. 5. GitHub automatically-generated repository metrics on total changes made by all research team members to the master branch, excluding conflict merges, from June 2, 2019 to May 25, 2020 with focus on October 23, 2019 to May 25, 2020 time span.

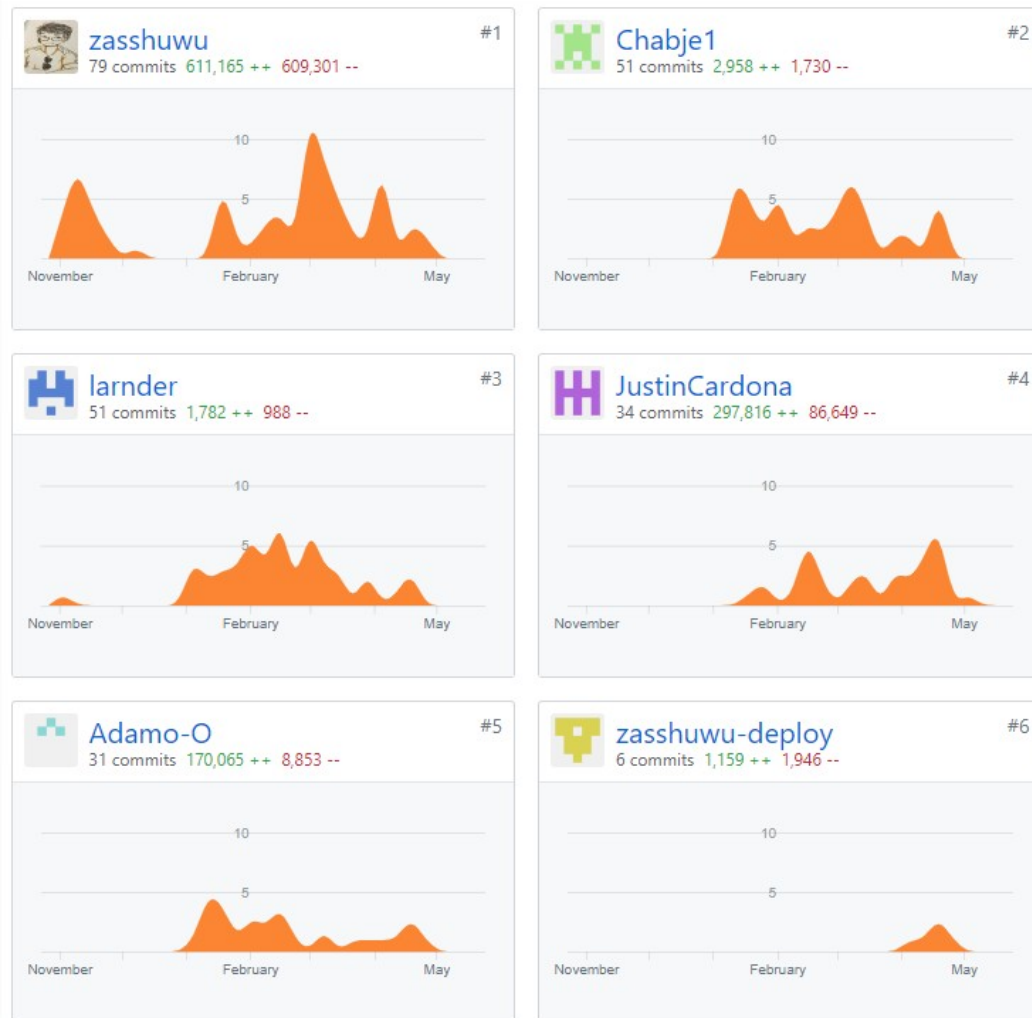


Fig. 6. GitHub automatically-generated tracking of individual commits and number of lines of code changed on the master branch, excluding conflict merges, from October 23, 2019 to May 25, 2020 - the whole research project time span.

The implementation of the ground rules, best practices and standards in the **Development** section proved to greatly benefit the team's efficiency in developing new features and gathering data for the project. As suggested by the graphs in **Figure 5 + Figure 6**, there were consistent commits to the master branch by all members of the team from the start date of the project on October 23, 2019 (initial meet-and-greet of team members and the instructor) to the start of May, 2020 - the end of the research period. This consistency infers that the team was able to perform their tasks efficiently without many conflicts. It is to be noted that there is a drop-off period on the tracking graph around the end of March due to the transition of work nature during the outbreak of coronavirus (COVID-19) pandemic throughout Canada. There was also a drop-off point around December to January, 2020 due to Christmas and New Year break.

In total, the research team refactored, reformatted, maintained, and developed a total of 16 working modules for the collection, analysis, and processing of accelerometric data. In conjunctions to these base modules, the team wrote 28 test scripts to generate proof-of-concept outputs from running the base modules, as well as to look for and eliminate edge cases that would break certain functionalities of the program. Among the set of 28 test scripts, 9 modules utilized the TensorFlow library for machine learning and some of the base modules to predict parameters of the object's motions. As part of the general development principle, the team studiously generated documentation on how to use the software for both beginners and advanced users accompanying the modules. A document dedicated for troubleshooting the software development environment was also created to help with onboarding the project.

Conclusion

At the end of the research term, although the team did not reach the original goal of having the software become open-source and available for all interested parties to get involved via an online platform, the team did successfully reach milestones that should propel the project closer to the goal. These milestones regarding the software aspect of the project include: redesigning the project structure and the software architecture using established practices in the computer programming field, designing a user-friendly tutorial for both beginners and advanced users to utilize the software, planning an efficient workflow for coding and experimentations, and developing a holistic framework for non-professional software development in a pedagogical setting.

In the future, the project would have a standalone website containing interactive tutorials, code usage demonstrations, and related research papers on classical mechanics and accelerometers. The website would become a hub for communication and information between the prospective users and the maintainers of the project. A demo of the website is currently available as of this writing <https://islab.nogamioka.com> and will become officially available at <https://islab.ca>. A simple graphical user interface (GUI) that allows a smoother onboarding process for beginners would also be implemented in the future, instead of having to manually run each script and module to access the functionality offered by the software.

With the outlook of the project, a framework for student-run research project having high involvement with computer programming was successfully developed. To be more precise, the target audience of the framework are students outside the computer science and software field who need to work with code without prior knowledge. This framework would pave the way for other prospective student teams and aid them in achieving their goals more effortlessly when it comes to designing a software package in scientific research and pedagogical pursuits.

Acknowledgement

The author would like to show their great appreciations for the following parties:

- Research Instructor & P.I.: Chris I. Larnder
- Research Supervisor: Roberta Silerova

- Team members: Adamo (dev.), Jerome (dev.), and Justin (exp.) – for their contributions.

References

- [Mar09] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Robert C. Martin Series. Pearson Education, Inc., 2009. ISBN: 978-0-1323-5088-4.
- [McC04] Steve McConnell. *Code Complete: A Practical Handbook of Software Construction. 2nd Edition*. Microsoft Press, 2004. ISBN: 978-0-7356-1967-8.

Glossary

- abstraction** A Computer Science term and field of study. [https://en.wikipedia.org/wiki/Abstraction_\(computer_science\)](https://en.wikipedia.org/wiki/Abstraction_(computer_science)). (Loosely) Refers to the generalization depth of functionalities in a program”. 4
- architecture** is the process of converting software characteristics such as flexibility, scalability, feasibility, reusability, and security into a structured solution that meets the technical and the business expectations. 8
- branch** In the context of git, a branch is a source code environment that exists separately from other branches (environments). Changes made are specific to each branch. 5
- class** is an abstraction of object-oriented programming and a type of data structure, containing methods that could be passed in with data. A variable could be instantiated as the class, inheriting all the methods. 4
- cloud** A modern storage solution where data are stored on computers and data centers physically inaccessible to users, but are available for access online whenever needed. This is a decentralized and redundant style of storage where data loss is ideally minimal and remote collaboration relies on. 6
- commit** (N, V) Refers to the act of officially placing the changes of code in a local branch. 3
- infrastructure** In the context of this paper, this refers to the building blocks of a software. E.g. programming language, operating system, source code structure, requirements & specifications, etc. 2
- module** A way to store definitions (variables, functions, classes) when closing and reopening a Python interpreter. Allows easier maintenance and reuse in the future. A module could be imported in another module, becoming a dependency. 1
- package** is a collection of Python modules in a folder, recognized implicitly by the Python interpreter. 4
- parameter** also called a *formal argument* is a special variable that exists as a positional placeholder in a function that could be passed in as data for the function to operate upon. 4
- remote** Refers to the copy of data stored on a hosting service or platform that can be downloaded by authorized parties as opposed to local data where only the party with access to the physical storage can access. Remote data is stored on the cloud. 6
- script** A bunch of Python codes bundled in a file to be run in a top-down manner. This action is called ‘executing a script’. 1
- source code** is the umbrella term for all the physical lines of code written for a program/software. 2