POLIASTRO: AN ASTRODYNAMICS LIBRARY WRITTEN IN PYTHON WITH FORTRAN PERFORMANCE

Juan Luis Cano Rodríguez
Universidad Politécnica de Madrid
Madrid, Spain

Helge Eichhorn
Technische Universität Darmstadt
Darmstadt, Germany

Frazer McLean
CS GmbH
Darmstadt, Germany

ABSTRACT

Python is a fast-growing language both for astronomic applications and for educational purposes, but it is often criticized for its suboptimal performance and lack of type enforcement. In this paper we present **poliastro**, a pure Python library for Astrodynamics that overcomes these obstacles and serves as a proof of concept of Python strengths and its suitability to model complex systems and implement fast algorithms.

poliastro features core Astrodynamics algorithms (such as resolution of the Kepler and Lambert problems) written in pure Python and compiled using numba, a modern just-in-time Python-to-LLVM compiler. As a result, preliminary benchmarks suggest a performance increase close to the reference Fortran implementation, with negligible impact in the legibility of the Python source. We analyze the effects of these tools, along with the introduction of new ahead-of-time compilers for numerical Python and optional type declarations, in the interpreted and dynamic nature of the language.

poliastro relies on well-tested, community-backed libraries for low level astronomical tasks, such as astropy and jplephem. We comment the positive outcomes of the new open development strategies and the permissive, commercial-friendly licenses omnipresent in the scientific Python ecosystem.

While recent approaches involve writing Python programs which are translated on the fly to lower level code, traditional Python libraries for scientific computing have succeeded because they leverage computing power to compiled languages. We briefly present tools to build wrappers to Fortran, C/C++, MATLAB and Java, which can be also useful for validation and verification, reusability of legacy code and other purposes.

1. INTRODUCTION

The Python programming language has seen broad recognition in the last decade among scientists and academics, being one of the most popular languages in astronomy[1] and for educational purposes[2]. It is highly trusted in corporative environments as well as a tool for scripting, automating tasks and creating high level APIs and wrappers.

However, the current situation of scientific codes is still delicate: most scientists and engineers that write numerical software (which often features a strong algorithmic component and has tight performance requirements) usually do not have any formal training on computer programming, let alone software engineering best practices[3]. In fact, in the Aerospace industry there is a sad track record of software failures[4, 5] that could have been avoided by following better software and systems engineering practices.

When selecting a certain programming language for a specific problem, we as engineers have the obligation to consider as much information as possible and make an informed decision based on technical grounds. For example, if defect density were to be selected as the single figure to rank the contenders, well-established languages for space applications such as FORTRAN or C would perform worse than functional languages such as Haskell or Erlang[6]. Another common misconception is to assume that each language features certain properties, while languages are abstract specifications and language *implementations* are the concrete systems we can measure. Other metrics that could be taken into account are readability and programmer productivity, specially considering that "programmers write roughly the same number of lines of code per unit time regardless of the language they use"[3].

In this paper we claim that the Python programming language, with the aid of both young projects and solid, well tested libraries, can be an optimal solution for the prototyping stage of the development and a fair complement to traditional alternatives in the production stage, in terms of performance, availability, maturity and maintainability. As a demonstrator we selected basic problems in Astrodynamics and compared the performance of existing FORTRAN or C++ implementations with our new Python implementations, comparing them in terms of code complexity and performance in section 2. Our Python code is available as part of the `poliastro` package, an open source Python library for Astrodynamics and Orbital Mechanics focused on interplanetary applications and released under the MIT license[7]. As a complement, in section 3 we present an overview of the techniques that can be used to use code written in Fortran, C/C++,

```
while count < numiter:
    y = norm_r0 + norm_r + A * (psi * c3(psi) - 1)
         / c2(psi)**.5
    # ...
    xi = np.sqrt(y / c2(psi))
    tof_new = (xi**3 * c3(psi) + A * np.sqrt(y)) /
         np.sqrt(k)

    if np.abs((tof_new - tof) / tof) < rtol:
        # Convergence check
        break
    else:
        count += 1
        if tof_new <= tof:  # Bisection check
            psi_low = psi
        else:
            psi_up = psi
        psi = (psi_up + psi_low) / 2
```

Fig. 1. Fragment of BMW-Vallado algorithm for Lambert's problem in Python. Notice its resemblance to pseudocode.

```
In [1]: import numpy as np

In [2]: list = list(range(0,100000))

In [3]: %%timeit
   ...: sum(list)
   ...:
1000 loops, best of 3: 1.32 ms per loop

In [4]: array = np.arange(0, 100000)

In [5]: %%timeit
   ...: np.sum(array)
   ...:
The slowest run took 780.86 times longer than the
    fastest. This could mean that
an intermediate result is being cached
10000 loops, best of 3: 38.9 μs per loop
```

Fig. 2. Microbenchmarks of pure Python versus NumPy

Java and MATLAB from Python, and their advantages and tradeoffs.

To conclude, we remark that `poliastro` and many other software packages would not be possible without the vast number of open source projects that lay the foundations for present and future work. Other authors have highlighted the potential of open source in the aerospace industry in terms of software reusability and collaboration between academia and private companies[8]. In section 4 we comment the practical outcomes of this philosophy, the challenges that need to be solved and its potential ramifications for the Aerospace industry.

2. PYTHON AS A CORE COMPUTATIONAL LANGUAGE

The Python programming language was started by Guido van Rossum in 1989 as a successor to the ABC language, and v1.0 was released in 1994[1]. It is therefore not new, and in fact it predates the Java language, first released in 1996. On the other hand, Python first uses for scientific purposes appeared as early as 1995, with the creation of a special interest group on numerical arrays[9]. However, in recent times the ecosystem has greatly improved, with the application of Open Development principles (see 4 for further discussion), the increasing interest and involvement of private companies and the generous funds given to projects like IPython[10] and Jupyter. Nowadays, it is one of the most used languages in fields like Astronomy[11] and small-to-medium Data Science, and heavily trusted for teaching undergraduate Computer Science in top universities[2]. In figure 1 we can see a fragment of one of the algorithms to solve Lambert's problem implemented in poliastro.

One of the most important differences between Python and compiled languages like Fortran or C is its dynamic typing nature. The variety of type systems across has traditionally been a major source of debate among programmers, and in fact some studies suggest that there is "a small but significant relationship between language class and defects"[6]. Languages featuring dynamic typing, as it is the case with Python, are often easier to write and read but more difficult to debug, as there are no guarantees about the types of the arguments, and have worse performance. While there is an increasing interest in developing type inference systems (see for instance the Julia and Scala languages), these are extremely difficult to set up for languages like Python [12].

Arguably the most important library in the scientific Python stack is NumPy, which implements n-dimensional numerical arrays and its related methods in C and wraps them using the CPython API[13]. In figure 2 we can see some microbenchmarks displaying the performance differences between two equivalent ways of adding up all the elements of a sequence. The first one is implemented using a Python dynamic list, whereas the second uses a NumPy array. As a result, the Python version is two orders of magnitude slower.

NumPy is an fundamental piece of software that powers most numerical codes written in Python nowadays. However, it is not always obvious how to vectorize the operations to make it suitable for using NumPy operations, and in some cases excessive vectorization can hurt readability. In the following sections we analyze some alternatives that have been put in place to overcome these limitations without greatly affecting the philosophy of the language.

2.1. Just-in-time compilation using numba

As discussed earlier, while it is possible to use NumPy to vectorize certain kinds of numerical operations, there might be

[1] http://python-history.blogspot.com.es/2009/01/brief-timeline-of-python.html

```
# --- LINE 29 ---
#   $103.3 = unary(fn=-, value=psi)  :: float64
#   $103.4 = global(gamma: <built-in function
    gamma>)  :: [...]
#   $const103.5 = const(int, 5)  :: int64
#   $103.6 = call $103.4($const103.5)  :: (int64,)
     -> float64
#   $103.7 = $103.3 / $103.6  :: float64
#   delta = $103.7  :: float64

delta = (-psi) / gamma(2 + 2 + 1)
```

Fig. 3. Example of numba annotation along corresponding line of Python code.

other cases where this may not be feasible and where the dynamic nature of Python leads to a performance penalty, specially when the algorithm involves several levels of nested looping. To overcome these limitations we used `numba`, an open source library which can infer types for array-oriented and math-heavy Python code and generate optimized machine instructions using the LLVM compiler infrastructure[14].

numba works by inferring the types of the variables of a Python function and refining them in several stages until it generates assembly code for the desired platform. In figure 3 we see a small fragment of the code that implements the Stumpff functions in poliastro along with its so-called Numba Intermediate Representation, which is the first stage of the optimization process.

To test the suitability of Python and `numba` for writing expensive mathematical algorithms in terms of performance and legibility we compare two algorithms for solving Lambert's problem: a simple bisection iteration over the universal variable as presented in [15] and [16] (from now on, referred to as BMW-Vallado algorithm) and the more recent algorithm by [17], based on a Householder iteration scheme over a Lambert-invariant variable (see [18] for the definition). These two algorithms are very different in nature: the former favors a simple approach that is robust and simple to implement, while the latter employs clever analytical transformations and a higher order root finding method to converge in very few iterations, hence using fewer function evaluations. Also, the former works only for single revolution solutions, whereas the latter can also find solutions corresponding to multiple revolutions.

Both algorithms implemented in Python and accelerated using `numba` are present in `poliastro`, a MIT-licensed open source library dedicated to problems focused on interplanetary Astrodynamics problems, such as orbit propagation, solution of Lambert's problem, conversion between position and velocity vectors and classical orbital elements and orbit plotting. `poliastro` documentation and source code are available online[2][3].

[2] https://github.com/poliastro/poliastro

[3] http://poliastro.readthedocs.org/en/v0.5.0/

Kernel version	Linux 2.6.32-504.3.3.el6.x86_64
Distribution	CentOS 6.6.
Processors	4x Intel(R) Core(TM)
CPU family	i7-2670QM CPU @ 2.20GHz
Memory	2974696 kB (Total)

Table 1. System information

We first performed some preliminary benchmarks to assess the performance increase due to JIT-compiling the functions. In figure 4 we displayed the distribution of running times of the two algorithms, adding a multiple revolution solution for the Izzo algorithm. The data was obtained using `pytest-benchmark` and plotted using `matplotlib` and `seaborn`[19][20]. We note several things:

- There is a significant improvement in running time for both algorithms.
- This performance improvement is more evident for the BMW-Vallado algorithm than for the Izzo algorithm. In the first case it the difference is roughly two orders of magnitude, while in the second case the running times decreased in half.
- For the single revolution case, the Vallado algorithm was significantly slower without JIT compiling, whereas it was the fastest when `numba` was enabled. On the other hand, only the Izzo algorithm is capable of computing multiple revolution solutions.

The difference in effectiveness is probably due to the fact that the Vallado algorithm performs more iterations while the Izzo algorithm is optimized for converging faster, at the expense of evaluating derivatives up to third order[17]. These results suggest that `numba` provides more performance boost when applying to algorithms with heavy looping or several levels of nesting.

2.2. Benchmarks against Fortran

Several implementations of the BMW-Vallado algorithm are freely available on the Internet as the companion software of Vallado's book[4], and the Fortran version was incorporated in `poliastro` v0.2 and distributed under the same terms with explicit permission of the author (see 3 for discussion on calling Fortran functions from Python). In `poliastro` v0.3 the algorithm was translated to Python and accelerated with `numba`.

For this paper we compared our own Python implementation of the Izzo algorithm with the one written in Fortran 2003/2008 available in `Fortran-Astrodynamics-Toolkit` (`https://github.com/jacobwilliams/Fortran%2DAstrodynamics%2DToolkit`), which is itself based

[4] http://celestrak.com/software/vallado-sw.asp

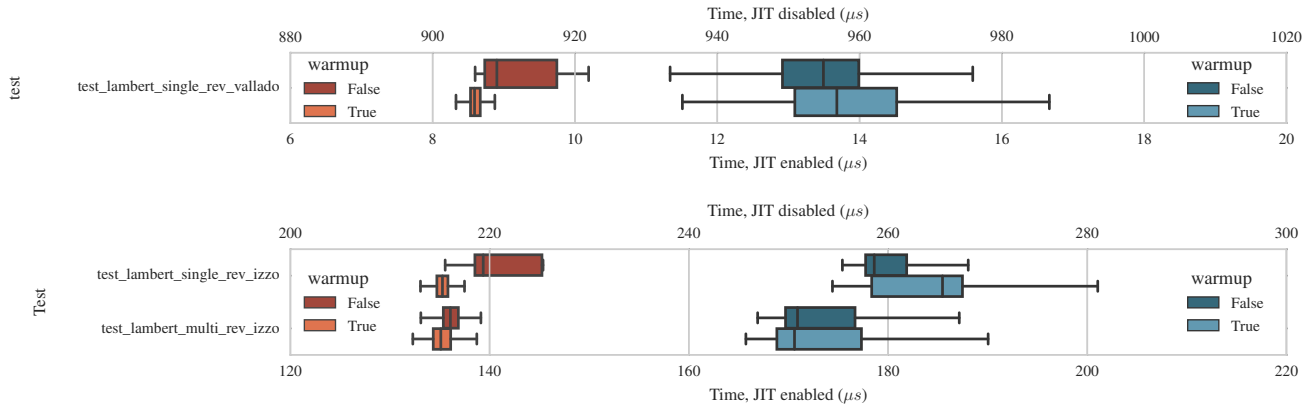

Fig. 4. Comparison of running times of the BMW-Vallado and Izzo algorithms, with and without JIT compiling. Notice the difference in time scales. Darker boxes were measured without warming up.

on the original code written by Izzo in C++[5]. This algorithm has the advantage that it can compute solutions corresponding to several full revolutions, allowing for less expensive trajectories at the cost of transfer time.

The tests were performed in a virtualized CentOS machine to avoid interferences from the outside world and provide homogeneous results. In table 1 we can see a brief summary of the system information.

The tests consisted in measuring the number of solutions per unit time that the programs could compute, to take advantage to the already written benchmarks for `Fortran-Astrodynamics-Toolkit`. We compiled the Fortran code with GNU gfortran and Intel ifort, while the Python code was benchmarked with and without JIT compilation for reference purposes. The results are summarized in table 2.

The fastest was the Fortran version compiled with Intel ifort, followed by the GNU gfortran compiler, whereas the Python versions are the slowest: this agrees with our expectations. However, **we note that, although there are evident performance differences between all the versions, the numba version manages to run at 44.8 % the speed of the gfortran version, and at 32.7 % of the ifort version**. This is a clear difference from the behaviour of the pure, non-accelerated Python version, which lags two orders of magnitude slower than the Fortran programs.

With these modest performance results in mind, we point out that the Python version has fewer lines of code, is arguably more readable and works on all major operative systems without changes or intermediate compilation steps. We argue that these assets are also of high value and that should be taken into account when selecting a programming language.

[5]https://github.com/esa/pykep/

```
def greeting(name: str) -> str:
    return 'Hello␣' + name
```

Fig. 5. Code annotated with type hints

2.3. Gradual typing

We have already discussed the advantages of using `numba` to accelerate numerical Python code. We have also seen that whether we manually specify the expected types of our functions or let the computer automatically infer them, in both cases there is some sort of type enforcement, which on the other hand is not part of the language itself and does not support its complete set of features. Python 3.5 introduced a new provisional module adding *type hints* (also known as *gradual typing*), this time focusing on providing indirect help to be used by Integrated Development Environments (IDEs) and other tools to supply more useful information to the developer[6]. In figure 5 we can see how this syntax is implemented.

These types can also be written to separate files ("stub files"), in a similar way than it is already done for C and C++ header files, which can, in addition, be distributed inside the project. This has the advantage that the original sources are not altered and that it keeps backwards compatibility with older versions of Python.

3. INTERFACE WITH COMPILED LANGUAGES

While we promote the advantages of Python as a numerical computing language, we also recognize the tremendous value and expertise already present in mature, battle-tested programs and libraries. In fact, the scientific Python community is the best example of how to combine both compiled and

[6]https://www.python.org/dev/peps/pep-0484/

Version	Minimum	Maximum	Median	Relative	IQR
Intel ifort, `-O2`	594620.8	654121.4	623536.2	**1.0**	25861.2
GNU gfortran, `-O2`	358478.2	505127.0	454613.6	**0.729**	68265.5
poliastro, numba	197610.9	206153.2	203615.8	**0.327**	3296.5
poliastro, pure Python	3502.7	3703.0	3639.6	**0.006**	65.6

Table 2. Benchmarking results

interpreted languages: since the very beginning, many Python libraries were written as wrappers to old FORTRAN or C++ code, using tools like f2py and SWIG (discussed below)[9]. It is therefore our intention to build on Python for new code and at the same time take advantage of available libraries written in different languages to avoid duplicating efforts.

3.1. C and C++: ctypes, Cython, SWIG, CFFI

As the reference Python implementation (CPython) is written in C, interfacing with C libraries is relatively simple. CFFI is an improvement over Python's `ctypes` module in the standard library, both of which use `libffi`. These modules allow C libraries to be exposed to Python. Additionally, CFFI automatically parses type declarations and function signatures, avoiding the need to re-write them in Python before they can be used.

Cython and SWIG are both hybrid approaches for interacting with C. SWIG supports multiple languages, so it is not limited to Python. While SWIG generates bindings for parts written in C, Cython takes a different approach which allows starting with pure Python code that gets compiled into C for an immediate improvement in performance. Type declarations can be added to certain variables and functions to allow more code to run natively. Cython has the advantage that it can talk to Python and C in the same source file due to its own Python-like syntax. This hybrid approach allows gradual improvement as pain points are found. Cython also integrates well with NumPy's type system. For the best performance improvements, it is possible to write in pure Cython, and have the compiler create two separate versions of a function: one exposed to Python, and a faster one used within Cython.

3.2. Fortran: f2py

Although we could use an intermediate C wrapper to easily call Fortran code using the ISO C binding capabilities introduced in version 2003, there is no reliable way of doing this with Fortran 95 and earlier due to the lack of standardization. To solve this problem, specially for FORTRAN 77 code, the `f2py` project was created in the early days of the scientific Python community, which wraps FORTRAN 77 and a subset of Fortran 95 directly in Python[21].

3.3. Others: Java, MATLAB

As we already said for the Fortran case, once we have solved the problem of the interoperability with C we could easily write intermediate wrappers to many other languages, as C is *lingua franca* in the computing world. For the case of Java and MATLAB there are some tools that can help the developers with some parts of the process.

- To call Java libraries from Python programs one of the most visible options is `JCC` (`http://lucene.apache.org/pylucene/jcc/`). According to its website, `JCC` is "a C++ code generator that produces a C++ object interface wrapping a Java library via Java's Native Interface (JNI)". Besides, "`JCC` also generates C++ wrappers that conform to Python's C type system making the instances of Java classes directly available to a Python interpreter". It is successfully used by the Orekit Python wrapper, which allows using the Orekit Java library from a Python program.

- Regarding the MATLAB environment, the most modern alternative is `pymatbridge` (`https://arokem.github.io/python-matlab-bridge/`), a communication layer between MATLAB and Python based on the `ZeroMQ` socket library[22]. An equivalent tool for the GNU Octave project[23], which allows running MATLAB-like programs using only free software, is `oct2py` (`https://github.com/blink1073/oct2py`). The latter had been successfully tested in `poliastro` v0.1 before the Octave code was replaced by Fortran subroutines.

4. DEVELOPMENT APPROACH

`poliastro` relies on well-tested, community-backed libraries for low level astronomical tasks, such as astropy[1] and jplephem. The library Orekit is another successful example of a software project developed in the open in the Astrodynamics community[24]. In this section we comment the positive outcomes of the new open development strategies and the permissive, commercial-friendly licenses omnipresent in the scientific Python ecosystem.

4.1. Free/Open Source software

Software licensing is usually an underestimated topic that should not be taken lightly for the reasons explained below. In particular, some surveys suggest that a high percentage of the software available on the Internet has no license whatsoever[7], which, under modern copyright law, might mean the opposite of what original authors intended[8]. This problem is more pervasive than anticipated: for example, the IERS software did not have a proper software license put in place until 2009[9], despite being extremely important in Flight Dynamics to compute changes of coordinate reference frames. Releasing works into the public domain is not a sensible choice, since copyright law is different from country to country and the implications remain unclear[25].

Open source software has a long history, with its roots dating back to the creation of the Free Sofware movement in the mid eighties[26]. A distinction is often made between "open source" and "free", in that the latter requires derivative works and linking programs to be released under the same license (which is often referred to as the "viral" nature of such licenses, see[27]). For obvious reasons, this distinction has profound implications on the commercial availability of the software.

Fortunately, the open source culture dominates in the scientific Python community, and is therefore safe to assume that most numerical Python libraries can be reused in commercial, closed source products[10]. This is an important advantage, since it means that companies usually do not have to worry about licensing or linking issues regarding their products, as long as they ship the proper citations and disclaimers. It also poses a great challenge for open source maintainers, since commercial users do not always contribute code back and non-commercial users often feel entitled to ask for features to be developed for free. The question on how to make open source software sustainable is still open and research is ongoing, and requires a deep implication of the stakeholders[28].

4.2. Open Development

For some scientific software packages it is common practice to upload new releases to a website or FTP directory to make them available to users, while there is no public list of open issues and they have to be privately reported to a private email address. That is the case with the Standards of Fundamental Astronomy (SOFA) Software Collection.

The value of open source software cannot be overstated: some studies suggest that, under certain circumstances, open source software will be of higher quality than the equivalent closed source counterpart[29]. However, as some have pointed out, for smaller projects it might not be viable to just publish releases when they are ready[30]. Many software projects follow a more open approach, with various degrees of adoption and success, which we will refer to as "Open Development"[11]. Among the characteristics of this approach we can name:

- Carrying development discussions on public mailing lists,
- Displaying a public list of open issues and known defects (as well as fixed ones),
- Publishing the complete history of the project using source control management tools,
- Performing public code reviews based on the previous two,
- Using Continuous Integration environments and striving for a high rate of statement or branch coverage,
- Embracing democratic and transparent decision making processes, with a focus on diversity and safety

The benefits of these development practices to commercial or hybrid projects have been studied (see [31]), and some companies have started public debates to explore their viability[12]. We therefore encourage astrodynamicists and private companies to engage in the discussion and explore novel ways of developing software for higher quality and better reuse.

5. CONCLUSIONS

Developing numerical and scientific software in a sustainable, consistent way is key for the success of many engineering projects. We claim that, for Astrodynamics projects, we can consider options different than the ones that have been traditionally used, specially since they already have strong roots in scientific and academic fields. We have shown that we can use `numba`, a young yet powerful tool, to increase the performance of our numerical Python code, with dramatic speed increments in some cases. We have performed some software benchmarks of the Izzo algorithm implemented both in Fortran 2003 and Python, obtaining that the best performance is achieved with Fortran compilers. However, the JIT compiled Python version stayed within the same order of magnitude of the Fortran versions and two orders of magnitude above the non accelerated version, and should therefore be considered as a valid alternative, at least in the initial stages of the project. To combine Python with other languages pervasive in Astrodynamics we have listed some tools that we can use to call C/C++, Fortran, Java and MATLAB/Octave code from Python. To conclude, we have described the advantages of open source and open development methods and highlighted

[7] http://www.theregister.co.uk/2013/04/18/github_licensing_study/
[8] https://opensource.com/law/13/8/github-poss-licensing
[9] https://igscb.jpl.nasa.gov/pipermail/igsmail/2009/006005.html
[10] http://nipy.sourceforge.net/software/license/johns_bsd_pitch.html
[11] https://opendevelopmentmethod.org/
[12] http://paypal.github.io/InnerSourceCommons/

their importance for future developments both for commercial and non commercial software.

Acknowledgements. I am grateful to the Free and Open Source Community in general, and specially the developers that work on Python, NumPy, SciPy, matplotlib, Jupyter and astropy. This extends to the people that make it possible to have free (as in freedom) operative systems, compilers, text editors, IDEs and browsers. I also thank Helgee Eichhorn and Frazer McLean for their input, Alberto Lorenzo for his invaluable work on the Continuous Integration infrastructure and GMV for allowing me to prepare this article in working hours. To conclude, I thank the European Commission, the Erasmus Programme and the Technical University of Madrid for allowing me to study one year at Politecnico di Milano, which sparked my passion for Astrodynamics and enrichened my vision of the world.

6. REFERENCES

[1] Thomas P. Robitaille et al., "Astropy: A community Python package for astronomy," *Astronomy & Astrophysics*, vol. 558, pp. A33, sep 2013.

[2] Philip Guo, "Python is Now the Most Popular Introductory Teaching Language at Top U.S. Universities," `http://web.archive.org/web/20160218070508/http://cacm.acm.org/blogs/blog-cacm/176450%2Dpython%2Dis%2Dnow%2Dthe%2Dmost%2Dpopular%2Dintroductory%2Dteaching%2Dlanguage%2Dat%2Dtop%2Dus%2Duniversities/fulltext`, 2014.

[3] Greg Wilson et al., "Best Practices for Scientific Computing," *PLoS Biology*, vol. 12, no. 1, pp. e1001745, jan 2014.

[4] Arden Albee et al., "Report on the loss of the Mars Polar Lander and Deep Space 2 missions," 2000.

[5] JL Lions et al., "Report by the inquiry board on the Ariane 5 flight 501 failure," *Joint Communication ESA-CNES*, 1996.

[6] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu, "A large scale study of programming languages and code quality in github," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014*. 2014, Association for Computing Machinery (ACM).

[7] Juan Luis Cano Rodríguez et al., "poliastro 0.5.0," mar 2016.

[8] Sven Ziemer and Gernot Stenz, "The case for open source software in aeronautics," *Aircraft Eng & Aerospace Tech*, vol. 84, no. 3, pp. 133–139, may 2012.

[9] K. Jarrod Millman and Michael Aivazis, "Python for Scientists and Engineers," *Comput. Sci. Eng.*, vol. 13, no. 2, pp. 9–12, mar 2011.

[10] Fernando Perez and Brian E. Granger, "IPython: A System for Interactive Scientific Computing," *Comput. Sci. Eng.*, vol. 9, no. 3, pp. 21–29, 2007.

[11] I. Momcheva and E. Tollerud, "Software Use in Astronomy: an Informal Survey," *ArXiv e-prints*, jul 2015.

[12] Brett Cannon, *Localized type inference of atomic types in python*, Ph.D. thesis, CALIFORNIA POLYTECHNIC STATE UNIVERSITY San Luis Obispo, 2005.

[13] Stéfan van der Walt, S Chris Colbert, and Gaël Varoquaux, "The NumPy Array: A Structure for Efficient Numerical Computation," *Comput. Sci. Eng.*, vol. 13, no. 2, pp. 22–30, mar 2011.

[14] Numba Development Team, "Numba," `http://numba.pydata.org`, 2016, Version 0.24.

[15] Roger R Bate, Donald D Mueller, and Jerry E White, *Fundamentals of astrodynamics*, Courier Corporation, 1971.

[16] David A Vallado, *Fundamentals of astrodynamics and applications*, vol. 12, Springer Science & Business Media, 2001.

[17] Dario Izzo, "Revisiting Lambert's problem," *Celest Mech Dyn Astr*, vol. 121, no. 1, pp. 1–15, oct 2014.

[18] RH Gooding, "A procedure for the solution of Lambert's orbital boundary-value problem," *Celestial Mechanics and Dynamical Astronomy*, vol. 48, no. 2, pp. 145–165, 1990.

[19] Michael Droettboom; John Hunter; Thomas A Caswell; Eric Firing; Jens Hedegaard Nielsen; Phil Elson; Benjamin Root; Darren Dale; Jae-Joon Lee; Jouni K. Seppänen; Damon McDougall; Andrew Straw; Ryan May; Nelle Varoquaux; Tony S Yu; Eric Ma; Charlie Moad; Steven Silvester; Christoph Gohlke; Peter Würtz; Thomas Hisch; Federico Ariza; Cimarron; Ian Thomas; James Evans; Paul Ivanov; Jeff Whitaker; Paul Hobson; mdehoon; Matt Giuca;, "matplotlib: matplotlib v1.5.1," 2016.

[20] Michael Waskom; Olga Botvinnik; Paul Hobson; John B. Cole; Yaroslav Halchenko; Stephan Hoyer; Alistair Miles; Tom Augspurger; Tal Yarkoni; Tobias Megies; Luis Pedro Coelho; Daniel Wehner; cynddl; Erik Ziegler; diego0020; Yury V. Zaytsev; Travis Hoppe; Skipper Seabold; Phillip Cloud; Miikka Koskinen; Kyle Meyer; Adel Qalieh; Dan Allan;, "seaborn: v0.5.0 (November 2014)," 2014.

[21] Pearu Peterson, "F2PY: a tool for connecting Fortran and Python programs," *International Journal of Computational Science and Engineering*, vol. 4, no. 4, pp. 296–305, 2009.

[22] Pieter Hintjens, *ZeroMQ: Messaging for Many Applications*, O'Reilly Media, Inc., 2013.

[23] John Wesley Eaton, David Bateman, and Søren Hauberg, *Gnu octave*, Network thoery London, 1997.

[24] Véronique Pommier-Maurussane and Luc Maisonobe, "Orekit: an Open-source Library for Operational Flight Dynamics Applications," in *International Conference on Astrodynamic Tools and Techniques (ICATT), ESA/ESAC, Madrid, Spain*, 2010, pp. 3–6.

[25] Ronan Deazley, *Rethinking copyright: history, theory, language*, Edward Elgar Publishing, 2006.

[26] Richard Stallman, "The GNU Manifesto," *j-DDJ*, vol. 10, no. 3, pp. 30–??, mar 1985.

[27] Richard Stallman, "Viewpoint Why open source misses the point of free software," *Communications of the ACM*, vol. 52, no. 6, pp. 31–33, 2009.

[28] Sonali K. Shah, "Motivation Governance, and the Viability of Hybrid Forms in Open Source Software Development," *Management Science*, vol. 52, no. 7, pp. 1000–1014, jul 2006.

[29] Jennifer W. Kuan, "Open Source Software as Consumer Integration Into Production," *SSRN Electronic Journal*, 2001.

[30] Andreas Prlić and James B. Procter, "Ten Simple Rules for the Open Development of Scientific Software," *PLoS Comput Biol*, vol. 8, no. 12, pp. e1002802, dec 2012.

[31] Audris Mockus, Roy T Fielding, and James D Herbsleb, "Two case studies of open source software development: Apache and Mozilla," *ACM Transactions on Software Engineering and Methodology*, vol. 11, no. 3, pp. 309–346, jul 2002.