OpenSatCom

Development of Open Source tools for SatCom constellation analysis

Second deliverable

Author: Juan Luis Cano Rodríguez `<hello@juanlu.space>`
Publication date: 2020-12-20

Introduction

Two-line Element Sets (TLEs), arguably the most popular exchange format for orbital information nowadays, are meant to be propagated with SGP4, one of the Simplified General Perturbations models published by the United States Department of Defense. This coupling between the two makes SGP4 arguably the most widely used analytical propagation algorithm. This method has several advantages: its source code has been publicly available for a very long time, and it offers a good tradeoff between running time and accuracy.

As simulating and operating large satellite fleets have become more common, the need to make satellite propagation as fast as possible while retaining the accuracy of SGP4 has become more pressing. Simpler models, like ones considering only the oblateness of the Earth spheroid, are enough to retain some perturbation effects, but leaves out the orbit decay caused by the atmospheric drag as well as a big part of higher order effects.

Orbit propagation is a sequential problem (equivalent to the resolution of an ordinary differential equation) and therefore it is not amenable to massive parallelization strategies at the algorithm level. However, propagating several satellites for several epochs is in itself a so-called “embarrasingly parallel” problem.

Overview

As part of the Development of Open Source tools for SatCom constellation analysis, one of the sub-activities of the OpenSatCom initiative implemented by Libre Space Foundation and Inno3 for the European Space Agency, the second proposed step was the “evaluation of available Python libraries for multi-satellite propagation”, which is described in this document.

With this purpose, we selected several free/open source implementations of SGP4, mostly focused on the Python programming language, and benchmarked them in terms of running time.

Selected implementations

Name	**Source code**	**License**	**Last commit**[1]
python-sgp4 (pure Python & C++ wrapper of Vallado’s code)	https://github.com/brandon-rhodes/python-sgp4	MIT	2020-xx-xx
python-sgp4 (NumPy vectorization)	https://github.com/enritoomey/python-sgp4/tree/master[2]	MIT	2019-12-16
python-sgp4 (Numba JIT)	https://github.com/astrojuanlu/python-sgp4/tree/the-return-of-numba	MIT	2020-12-07
cysgp4 (Cython wrapper of sgp4lib)	https://github.com/bwinkel/cysgp4	GPLv3	2020-11-03

Because of time and skill limitations we were not able to include Orekit as part of the benchmarks.

Selected use cases

- One satellite, one date
- One satellite, many dates
- Many satellites, many dates

[1] In `master` branch or equivalent (applies to whole project)
[2] Forked to https://github.com/astrojuanlu/python-sgp4/tree/numpy-vectorization to change the name of the package

For each use case, the preferred option of each project has been used instead of trying them all. For example, python-sgp4 provides `Satrec.sgp4_array()` method for arrays of dates and `SatrecArray.sgp4()` for arrays of satellites and dates.

Benchmarks

The source code of the benchmarks is available online:

https://github.com/astrojuanlu/sgp4-benchmarks/tree/v2020.12.1

And they have been reviewed by Brandon Rhodes, author of python-sgp4, and Benjamin Winkel, author of cysgp4. We appreciate their prompt response and their thoughtful suggestions.

The benchmarks were run in two different systems:

- A personal laptop

```
CPU: Dual Core Intel Core i5-7200U (-MT MCP-)
speed/min/max: 500/400/3100 MHz
Kernel: 5.4.0-58-generic x86_64
Up: 8h 49m
Mem: 7668.9/15931.1 MiB (48.1%)
Storage: 1.39 TiB (44.1% used)
Procs: 291
Shell: bash 5.0.17
inxi: 3.0.38
```

- A dedicated cloud server

```
CPU: 2x 12-Core High Performance Datacenter vCPU (-MCP SMP-)
speed: 2495 MHz
Kernel: 5.4.0-37-generic x86_64
Up: 5h 27m
Mem: 532.4/96577.4 MiB (0.6%)
Storage: 30.00 GiB (29.9% used)
Procs: 293
Shell: bash 5.0.16
inxi: 3.0.38
```

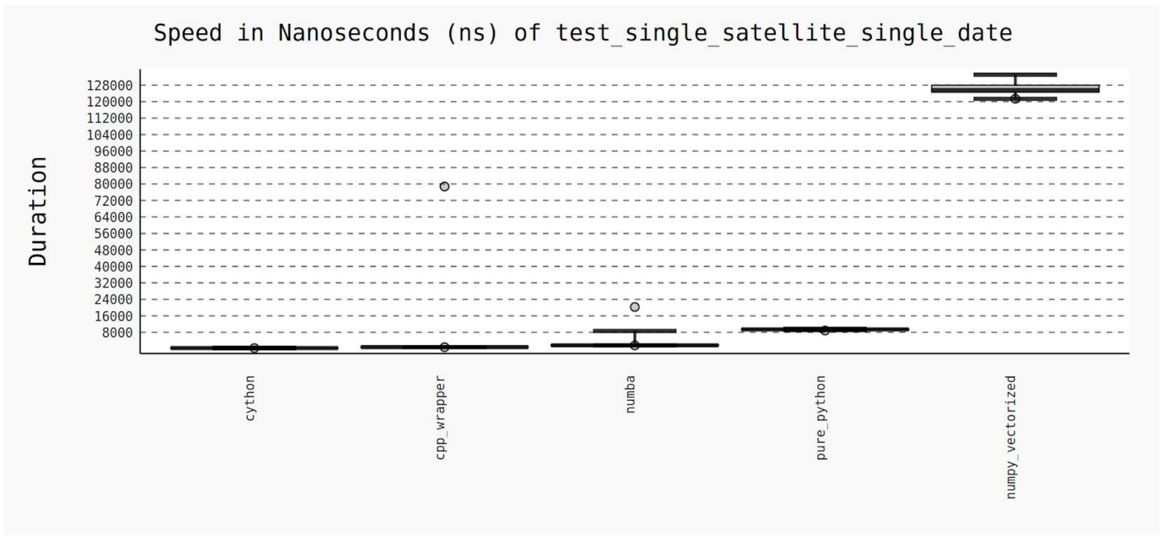
And the results were plotted using the pytest-benchmark histogram plotting functionality.

Results

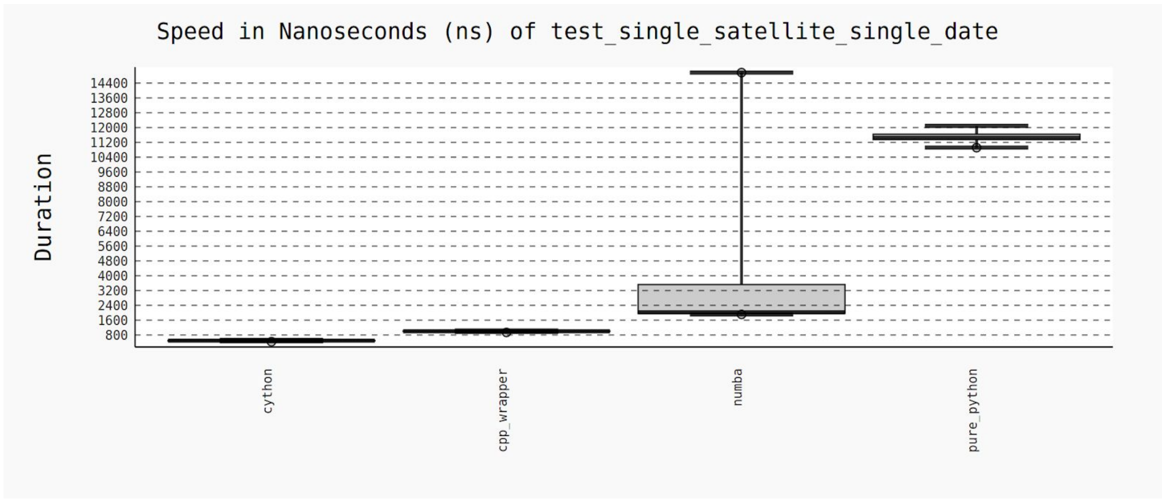
Single satellite, single date

cysgp4 is the fastest in this case (1x), while the NumPy vectorized implementation happened to perform very poorly (+300x), even slower than the pure Python case (24x). The numba implementation showed the largest variance, and was slower (4x) than both cysgp4 and the Vallado C++ wrapper (2x).

Results were similar in the dedicated server and are not included here.



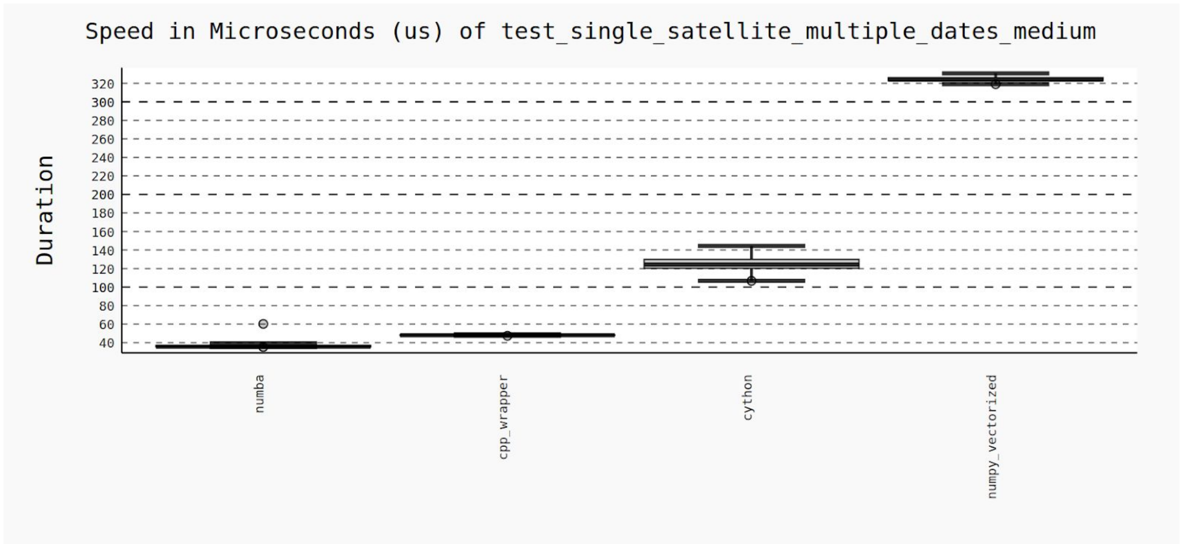
Personal laptop, all implementations



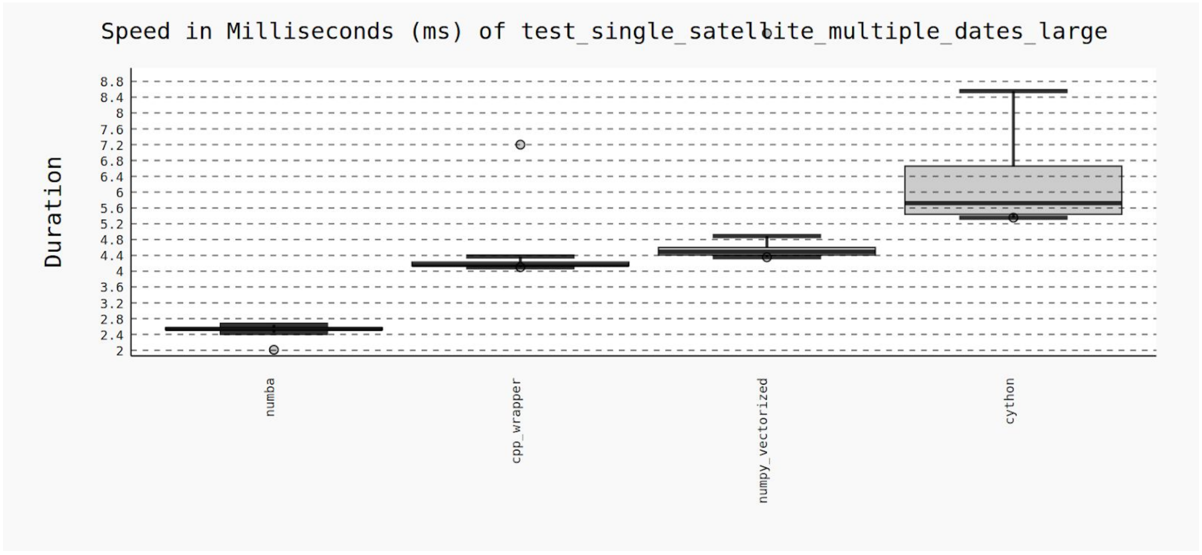
Personal laptop, NumPy vectorized excluded

Single satellite, multiple dates

We considered both a **medium** case with 101 dates and a **large** case with 10 100 dates. On the personal laptop the numba implementation was the fastest (1x), followed by the Vallado C++ wrapper (1.3x, 1.4x respectively), even though numba exhibited slightly larger standard deviation. The NumPy vectorized version performed better in the large case (9x, 1.8x) while cysgp4 was more consistent (3.5x, 2.5x). The pure Python version was 100x and 148x slower and is not displayed in the plots anymore.

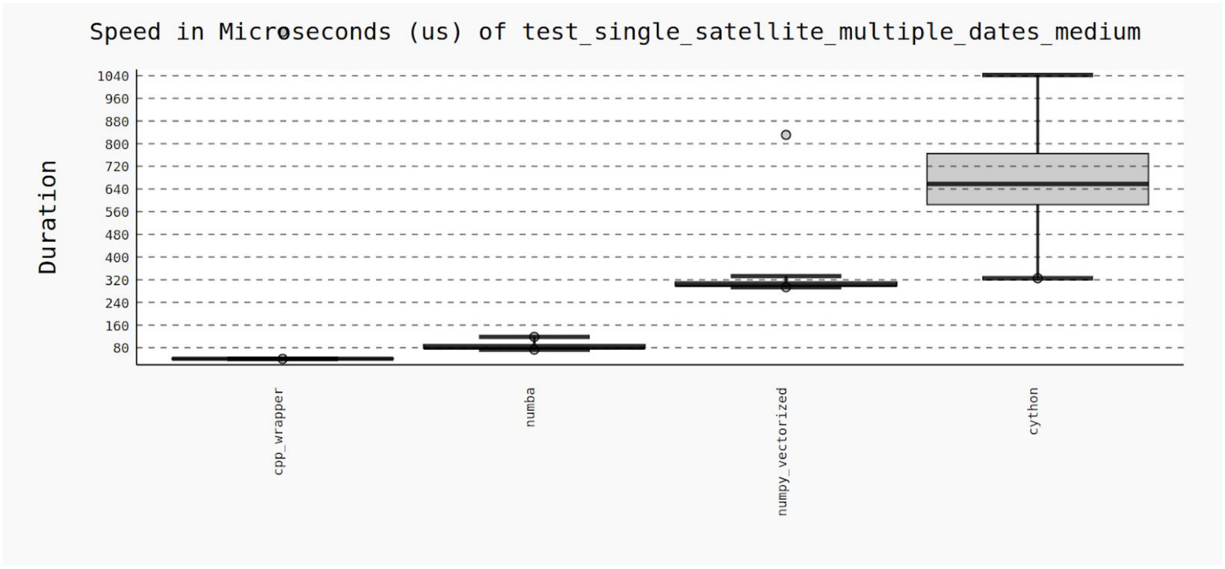


Personal laptop, pure Python excluded

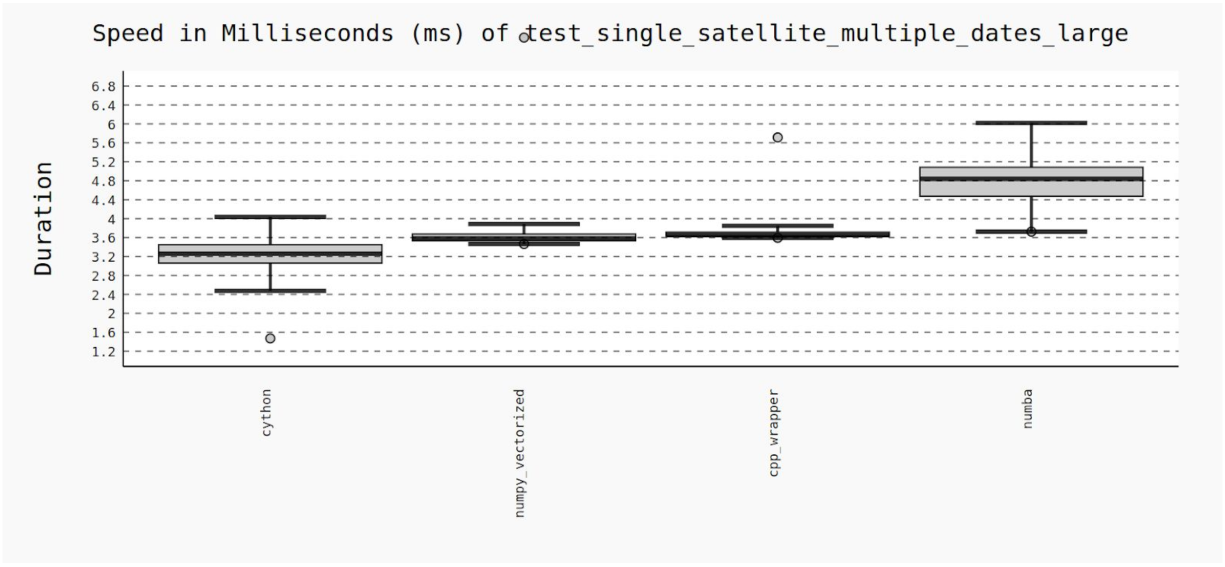


Personal laptop, pure Python excluded

On the dedicated server the results were quite different: the C++ wrapper and cysgp4 were the fastest for the medium and large cases respectively (1x and 1x) while numba lost the advantage in both (2x, 1.5x). numba happened to be the slowest in the large case (excluding the pure Python version) although the running times were all very similar, while cysgp4 performed rather slowly in the medium case (16x) and exhibited a large standard deviation.



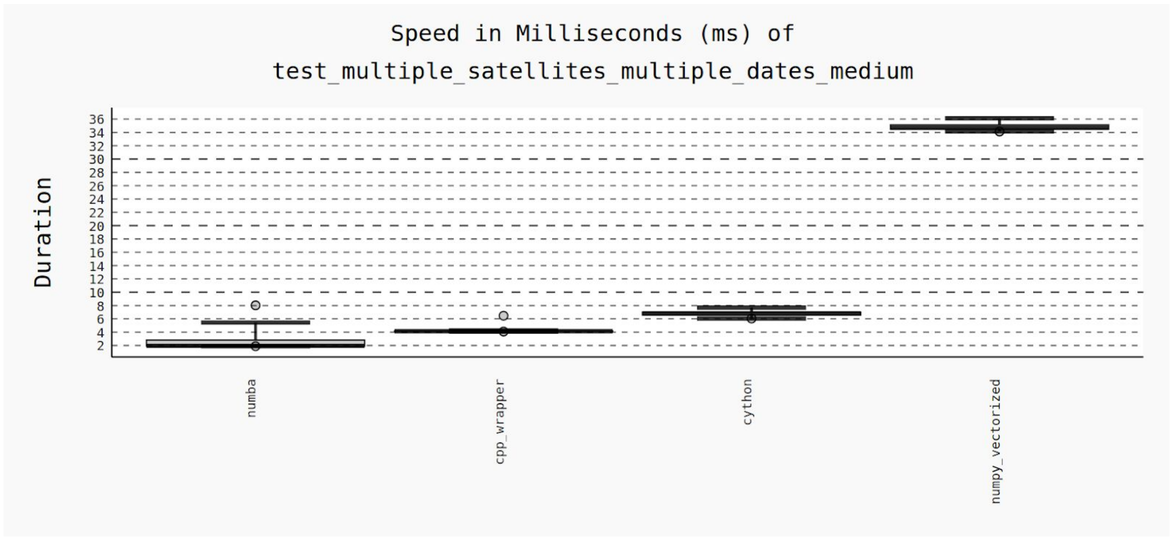
Dedicated server, pure Python excluded



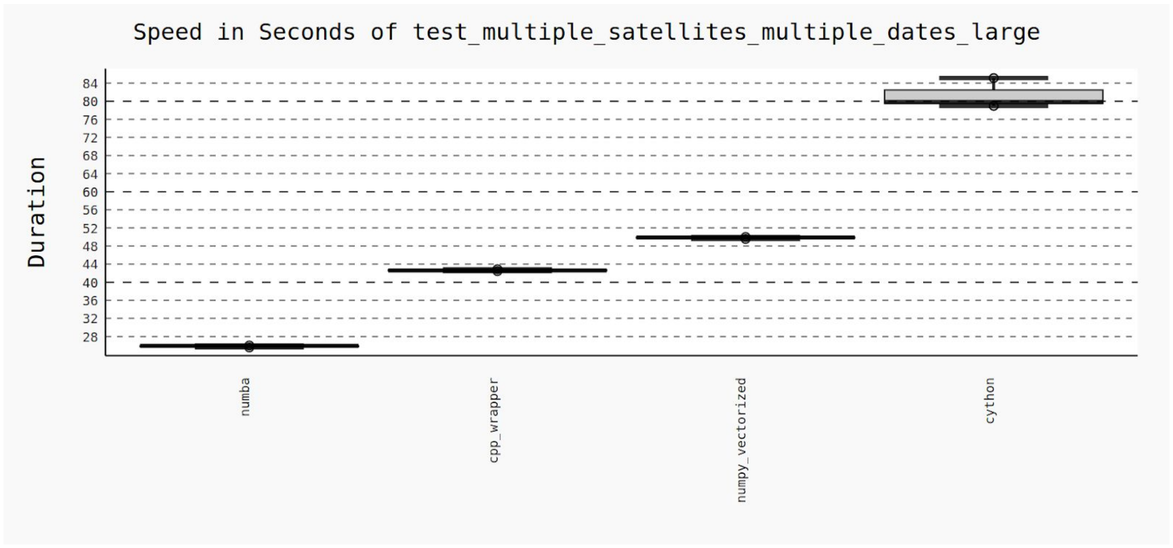
Dedicated server, pure Python excluded

Multiple satellites, multiple dates

Similarly to the previous case, we considered both a **medium** case with 100 satellites and 101 dates and a **large** case with 10 000 satellites and 10 100 dates. In all four combinations (medium/large, personal laptop/dedicated server) the fastest was the numba implementation (1x), with the pure Python version too slow to even be considered. On the personal laptop, the C++ wrapper was second (2.1x, 1.6x) and again the NumPy vectorized version performed better in the large case than in the medium case (18x, 3x) while cysgp4 was more consistent (3.5x, 3x).

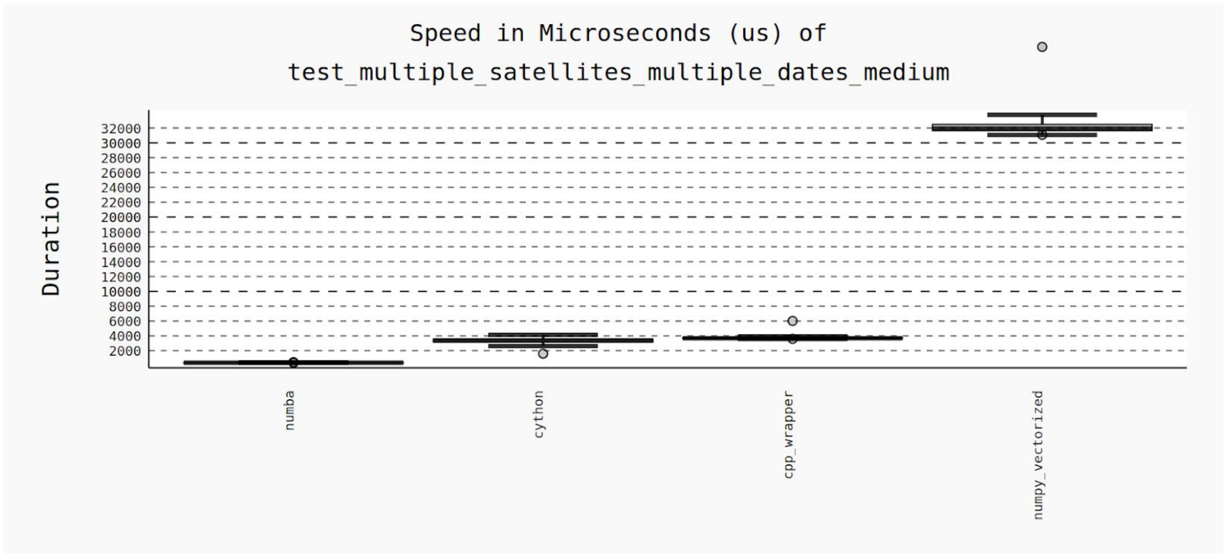


Personal laptop, pure Python excluded

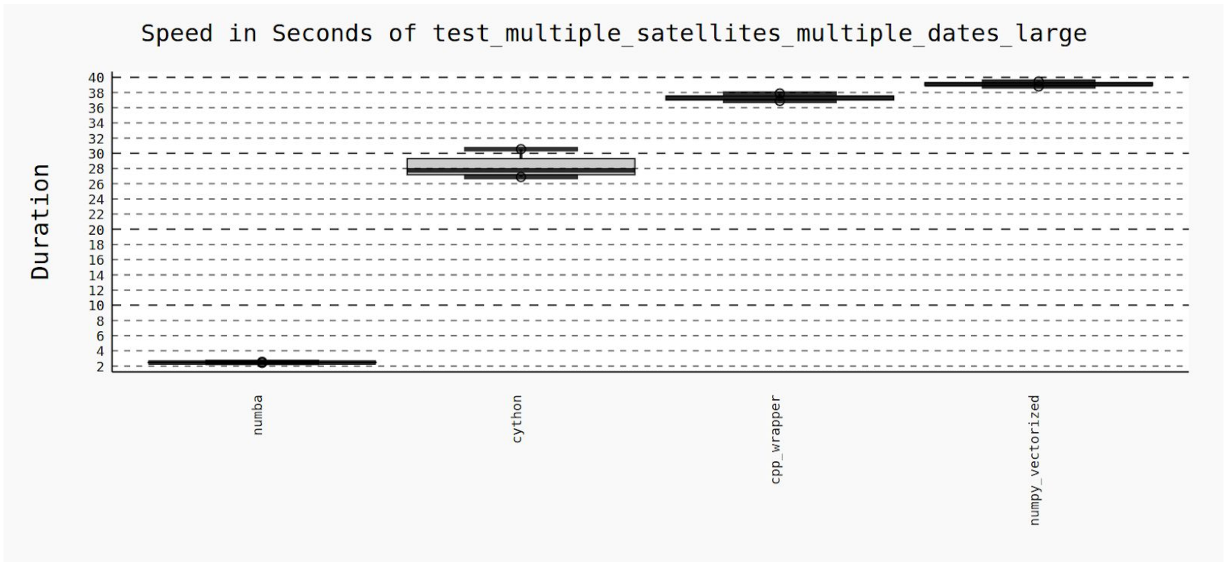


Personal laptop, pure Python excluded

On the dedicated server, cysgp4 is consistently second to numba, although noticeably slower (8.8x, 11x) while the C++ wrapper was third (9.6x, 15x) and again the NumPy vectorized version performed better in the large case than in the medium one, despite being the slowest one (84x, 15.8x).



Dedicated server, pure Python excluded



Dedicated server, pure Python excluded

Conclusions and future work

The results are very different depending on the architecture and the nature of the problem. For the simplest case of single satellite and single date, there is a very clear advantage of cysgp4 (Cython) and the Vallado C++ wrapper, with numba offering a speedup over the pure Python version at the cost of very large compilation times. On the other extreme, for the multiple satellite multiple dates case, numba is the fastest option, and more so in machines with a large number of CPUs. In the middle, the results are more heterogeneous, and in particular it can be observed that the NumPy vectorized version performs much better the larger the problem is.

Several lines of future work are suggested:

Micro optimizations to the implementations

Naturally, more work could be devoted to improving any of the existing implementations, by refactoring the code, exploring the effect of intermediate variables, inlining some operations... In particular, the numba compilation times were very large because there were over a hundred typed variables involved, most of which are intermediate results stored by the Vallado algorithm.

Comparing RAM usage

It was observed during the benchmarks that cysgp4 exhibited a distinct pattern in use of RAM. It would be interesting to add this information to the benchmarks as well, since it affects the sizing of the computational resources.

More systematic comparison of running time vs problem size

The chosen problem sizes were totally arbitrary, and in particular it was observed that the NumPy vectorized version exhibited much better results when the problem was large, going from 4th to 3rd place in some rankings. On the other hand, although the running time should in principle be linearly proportional to the problem size and inversely proportional to the number of cores, this hypothesis was not tested at all, and surprising results could emerge as a result of overheads and other factors.

Ad-hoc algorithm for satellite constellations

Satellite constellations usually have very similar orbits, with significant differences only the right ascension of the ascending node and the argument of latitude. This fact could be exploited by caching some of the intermediate calculations of SGP4 and accelerating the propagation of multiple satellites.

Inclusion of non-Python projects

Python was chosen because of familiarity, its pervasiveness in the open source ecosystem, the availability of benchmarking tools, and the ease of installation. However, more projects could be included in the benchmarks, in particular Orekit, which was left out from this analysis.

Summary

	Personal laptop	**Dedicated server**
Single satellite, single date	cysgp4, C++ wrapper	cysgp4, C++ wrapper
Single satellite, multiple dates	numba, C++ wrapper	?
Multiple satellites, multiple dates	numba, C++ wrapper	numba, cysgp4