**poliastro** is an open source collection of Python subroutines useful in Astrodynamics and Orbital Mechanics. It tries to provide a simple and intuitive API and handles physical quantities with units. Some of its awesome features are:

- Orbit propagation (Keplerian orbits)
- Conversion between position and velocity vectors and classical orbital elements
- Hohmann and bielliptic maneuvers computation
- Trajectory plotting
- Initial orbit determination (Lambert problem)
- Planetary ephemerides (SPICE kernels)

And more to come!

The [source code, issue tracker and wiki](https://github.com/poliastro/poliastro) are hosted on GitHub, and all contributions and feedback are more than welcome.

poliastro works on both Python 2 and 3 and is released under the MIT license, hence allowing commercial use of the library.
from poliastro.examples import molniya
from poliastro.plotting import plot
plot(molniya)

Note: Older versions of poliastro relied on some Fortran subroutines written by David A. Vallado for his book “Fundamentals of Astrodynamics and Applications” and available on the Internet as the companion software of the book. The author explicitly gave permission to redistribute these subroutines in this project under a permissive license.
1.1 About poliastro

1.1.1 Overview

poliastro is an open source collection of Python subroutines for solving problems in Astrodynamics and Orbital Mechanics.

poliastro combines cutting edge technologies like Python JIT compiling (using numba) with young, well developed astronomy packages (like astropy and jplephem) to provide a user friendly API for solving Astrodynamics problems. It is therefore an experiment to mix the best Python open source practices with my love for Orbital Mechanics.

Since I have only solved easy academic problems I cannot assess the suitability of the library for professional environments, though I am aware that at least a company that uses it.

1.1.2 History

I started poliastro as a wrapper of some MATLAB and Fortran algorithms that I needed for a University project: having good performance was a must, so pure Python was not an option. As a three language project, it was only known to work in my computer, and I had to fight against oct2py and f2py for long hours.

Later on, I enhanced poliastro plotting capabilities to serve me in further University tasks. I removed the MATLAB (Octave) code and kept only the Fortran algorithms. Finally, when numba was mature enough, I implemented everything in pure Python and poliastro 0.3 was born.

1.1.3 Future ideas

These are some things that I would love to implement in poliastro to expand its capabilities:

- 3D plotting of orbits
- Propagation of near-Earth satellites
- TLE reading
- Groundtrack plotting
- Continuous thrust maneuvers
1.1.4 Related software

These are some projects which share similarities with poliastro or which served as inspiration:

- **astropy**: According to its website, “The Astropy Project is a community effort to develop a single core package for Astronomy in Python and foster interoperability between Python astronomy packages”. Not only it provides important core features for poliastro like time and physical units handling, but also sets a high bar for code quality and documentation standards. A truly inspiring project.

- **Skyfield**: Another Astronomy Python package focused on computing observations of planetary bodies and Earth satellites written by Brandon Rhodes. It is the successor of pyephem, also written by him, but skyfield is a pure Python package and provides a much cleaner API.

- **Plyades**: A pioneering astrodynamics library written in Python by Helgee Eichhorn. Its clean and user friendly API inspired me to completely refactor poliastro 0.2 so it could be much easier to use. It has been stalled for a while, but at the moment of writing these lines its author is pushing new commits.

- **orbital**: Yet another orbital mechanics Python library written by Frazer McLean. It is very similar to poliastro (orbital plotting module was inspired in mine) but its internal structure is way smarter. It is more focused in plotting and it even provides 3D plots and animations.

- **orekit-python-wrapper**: According to its website, “The Orekit python wrapper enables to use Orekit within a normal python environment”, using JCC. Orekit is a well-stablished, mature open source library for Astrodynamics written in Java strongly supported by several space agencies. The Python wrapper is developed by the Swedish Space Corporation.

1.1.5 About the author

I am Juan Luis Cano Rodríguez (two names and two surnames, it’s the Spanish way!), an Aerospace Engineering student with a passion for Astrodynamics and the Open Source world. I started poliastro when I was an Erasmus student at Politecnico di Milano, an important technical university in Italy which deeply influenced my life and ambitions and gave name to the library itself. It is and always will be my tiny tribute to a country that will always be in my heart and to people that never ceased to inspire me. Grazie mille!

1.2 Getting started

1.2.1 Requirements

poliastro requires the following Python packages:

- NumPy, for basic numerical routines
- Astropy, for physical units and time handling
- numba (optional), for accelerating the code
- jplephem, for the planetary ephemerides using SPICE kernels
- matplotlib, for orbit plotting
- scipy, for root finding and numerical propagation

poliastro is usually tested on Linux, Windows and OS X on Python 2.7, 3.3, 3.4 and 3.5, using NumPy 1.9 and 1.10 (single codebase).
1.2.2 Installation

The easiest and fastest way to get the package up and running is to install poliastro using conda:

$ conda install poliastro --channel poliastro

You can also install poliastro from PyPI using pip, given that you already have all the requirements:

$ pip install poliastro

You can also download poliastro source from GitHub and type:

$ python setup.py install

Development installations are also supported thanks to setuptools:

$ python setup.py develop

Warning: It is recommended that you never ever use sudo with distutils, pip, setuptools and friends in Linux because you might seriously break your system [1][2][3][4]. Options are per user directories, virtualenv or local installations.

1.2.3 Testing

If installed correctly, the tests can be run using py.test:

$ python -c "import poliastro; poliastro.test()"
Running unit tests for poliastro
[...] OK
$

If for some reason any test fails, please report it in the issue tracker.

1.3 User guide

1.3.1 Defining the orbit: State objects

The core of poliastro are the State objects inside the poliastro.twobody module. They store all the required information to define an orbit:

- The body acting as the central body of the orbit, for example the Earth.
- The position and velocity vectors or the orbital elements.
- The time at which the orbit is defined.

First of all, we have to import the relevant modules and classes:

```
import numpy as np
from astropy import units as u

from poliastro.bodies import Earth, Sun
from poliastro.twobody import State
```
**From position and velocity**

There are several methods available to create `State` objects. For example, if we have the position and velocity vectors we can use `from_vectors()`:

```python
# Data from Curtis, example 4.3
r = [-6045, -3490, 2500] * u.km
v = [-3.457, 6.618, 2.533] * u.km / u.s
ss = State.from_vectors(Earth, r, v)
```

And that’s it! Notice a couple of things:

- Defining vectorial physical quantities using Astropy units is terribly easy. The list is automatically converted to a `Quantity`, which is actually a subclass of NumPy arrays.
- If no time is specified, then a default value is assigned:

```python
>>> ss.epoch
<Time object: scale='utc' format='jyear_str' value=J2000.000>
>>> ss.epoch.iso
'2000-01-01 12:00:00.000'
```

If we’re working on interactive mode (for example, using the wonderful IPython notebook) we can immediately plot the current state:

```python
from poliastro.plotting import plot
plot(ss)
```

This plot is made in the so called **perifocal frame**, which means:

- we’re visualizing the plane of the orbit itself,
- the \(x\) axis points to the pericenter, and
- the \(y\) axis is turned \(90 \text{ } \circ\) in the direction of the orbit.

The dotted line represents the osculating orbit: the instantaneous Keplerian orbit at that point. This is relevant in the context of perturbations, when the object shall deviate from its Keplerian orbit.
From classical orbital elements

We can also define a \texttt{State} using a set of six parameters called orbital elements. Although there are several of these element sets, each one with its advantages and drawbacks, right now poliastro supports the \textit{classical orbital elements}:

- \textit{Semilatus rectum} $(p)$.
- Eccentricity $(e)$.
- Inclination $(i)$.
- Right ascension of the ascending node $(\Omega)$.
- Argument of pericenter $(\omega)$.
- True anomaly $(\nu)$.

\textbf{Note:} poliastro uses the \textit{semilatus rectum} instead of the semimajor axis $(a)$ to avoid singularities when working with parabolic or near-parabolic orbits, where the latter takes infinite values.

In this case, we’d use the method \texttt{from\_classical()}: 

```python
# Data for Mars at J2000 from JPL HORIZONS
a = 1.523679 * u.AU
ecc = 0.093315 * u.one
p = a * (1 - ecc**2)
inc = 1.85 * u.deg
raan = 49.562 * u.deg
argp = 286.537 * u.deg
nu = 23.33 * u.deg
ss = State.from_classical(Sun, p, ecc, inc, raan, argp, nu)
```

Notice that whether we create a \texttt{State} from \texttt{(r)} and \texttt{(v)} or from elements we can access many mathematical properties individually:

```python
>>> ss.period.to(u.day)
<Quantity 686.9713888628166 d>
>>> ss.v
<Quantity [ 1.16420211, 26.29603612, 0.52229379] km / s>
```

To see a complete list of properties, check out the \texttt{poliastro.twobody.State} class on the API reference.

1.3.2 Changing the orbit: Maneuver objects

\texttt{poliastro} helps us define several in-plane and general out-of-plane maneuvers with the \texttt{Maneuver} class inside the \texttt{poliastro.maneuver} module.

Each \texttt{Maneuver} consists on a list of impulses $(\Delta v_i)$ (changes in velocity) each one applied at a certain instant $(t_i)$. The simplest maneuver is a single change of velocity without delay: you can recreate it either using the \texttt{impulse()} method or instantiating it directly.

```python
dv = [5, 0, 0] * u.m / u.s
man = Maneuver.impulse(dv)
man = Maneuver((0 * u.s, dv))  # Equivalent
```

There are other useful methods you can use to compute common in-plane maneuvers, notably \texttt{hohmann()} and \texttt{bielliptic()} for Hohmann and bielliptic transfers respectively. Both return the corresponding \texttt{Maneuver} object,
which in turn you can use to calculate the total cost in terms of velocity change ($\sum |\Delta v_i|$) and the transfer time:

```python
>>> ss_i = State.circular(Earth, alt=700 * u.km)
>>> hoh = Maneuver.hohmann(ss_i, 36000 * u.km)
>>> hoh.get_total_cost()
<Quantity 3.6173981270031357 km / s>
>>> hoh.get_total_time()
<Quantity 15729.741535747102 s>
```

You can also retrieve the individual vectorial impulses:

```python
>>> hoh.impulses[0]
(<Quantity 0 s>, <Quantity [ 0. , 2.19739818, 0. ] km / s>)
>>> hoh[0]  # Equivalent
(<Quantity 0 s>, <Quantity [ 0. , 2.19739818, 0. ] km / s>)
>>> tuple(_.decompose([u.km, u.s]) for _ in hoh[1])
(<Quantity 15729.741535747102 s>, <Quantity [ 0. , 1.41999995, 0. ] km / s>)
```

To actually retrieve the resulting `State` after performing a maneuver, use the method `apply_maneuver()`:

```python
>>> ss_f = ss_i.apply_maneuver(hoh)
>>> ss_f.rv()
(<Quantity [-3.60000000e+04, -7.05890200e-11, -0.00000000e+00] km>, <Quantity [-8.97717523e-16, -3.32749489e+00, -0.00000000e+00] km / s>)
```

## 1.3.3 More advanced plotting: OrbitPlotter objects

We previously saw the `poliastro.plotting.plot()` function to easily plot orbits. Now we’d like to plot several orbits in one graph (for example, the maneuver we computed in the previous section). For this purpose, we have `OrbitPlotter` objects in the `plotting` module.

These objects hold the perifocal plane of the first `State` we plot in them, projecting any further trajectories on this plane. This allows to easily visualize in two dimensions:

```python
from poliastro.plotting import OrbitPlotter
op = OrbitPlotter()
ss_a, ss_f = ss_i.apply_maneuver(hoh, intermediate=True)
op.plot(ss_i, label="Initial orbit")
op.plot(ss_a, label="Transfer orbit")
op.plot(ss_f, label="Final orbit")
```

Which produces this beautiful plot:

## 1.3.4 Where are the planets? Computing ephemerides

New in version 0.3.0.

Thanks to the awesome `jplephem` package, poliastro can now read Satellite Planet Kernel (SPK) files, part of NASA’s SPICE toolkit. This means that we can query the position and velocity of the planets of the Solar System.

The first time we import `poliastro.ephem` we will get a warning indicating that no SPK files are present:

```python
>>> import poliastro.ephem
No SPICE kernels found under ~/.poliastro. Please download them manually or using
  poliastro download-spk [-d NAME]
```
Fig. 1.1: Plot of a Hohmann transfer.
This is because poliastro does not download any data when installed: SPK files weight several MiB and that would slow the download process. Instead, we are requested to download them from NASA website or use the builtin command-line utility:

```
$ poliastro download-spk --name de421
```

No SPICE kernels found under ~/.poliastro. Please download them manually or using 

```
poliastro download-spk [-d NAME]
```

to provide a default kernel, else pass a custom one as an argument to `planet_ephem`.

If no --name argument is provided, de430 will be downloaded. Alternatively, we can use `poliastro.ephem.download_kernel()` from a Python session:

```
>>> from poliastro import ephem
>>> ephem.download_kernel("de421")
File de421.bsp already exists under /home/juanlu/.poliastro
```

In this case, the name argument is required.

Once we have downloaded an SPK file we can already compute the position and velocity vectors of the planets with the `poliastro.ephem.planet_ephem()` function. All we need is the body we are querying and an `astropy.time.Time` scalar or vector variable:

```
>>> from astropy import time
>>> epoch = time.Time("2015-05-09 10:43")
>>> from poliastro import ephem
>>> r, v = ephem.planet_ephem(ephem.EARTH, epoch)
>>> r
<Quantity [-9.99802065e+07, -1.03447226e+08, -4.48696791e+07] km>
>>> v
<Quantity [ 1880007.6848216 , -1579126.15900176, -684591.24441181] km / s>
```

**Note:** The position and velocity vectors are given with respect to the Solar System Barycenter in the International Standard Reference Frame, which has the Equator as the fundamental plane.

### 1.3.5 Traveling through space: solving the Lambert problem

The determination of an orbit given two position vectors and the time of flight is known in celestial mechanics as Lambert’s problem, also known as two point boundary value problem. This contrasts with Kepler’s problem or propagation, which is rather an initial value problem.

The module `poliastro.iod` allows as to solve Lambert’s problem, provided the main attractor’s gravitational constant, the two position vectors and the time of flight. As you can imagine, being able to compute the positions of the planets as we saw in the previous section is the perfect complement to this feature!

For instance, this is a simplified version of the example *Going to Mars with Python using poliastro*, where the orbit of the Mars Science Laboratory mission (rover Curiosity) is determined:
```python
>>> from astropy import time
>>> date_launch = time.Time('2011-11-26 15:02', scale='utc')
>>> date_arrival = time.Time('2012-08-06 05:17', scale='utc')
>>> tof = date_arrival - date_launch
>>> from poliastro import ephem
>>> r0, _ = ephem.planet_ephem(ephem.EARTH, date_launch)
>>> r, _ = ephem.planet_ephem(ephem.MARS, date_arrival)
>>> from poliastro import iod
>>> from poliastro.bodies import Sun
>>> v0, v = iod.lambert(Sun.k, r0, r, tof)
>>> v0
<Quantity [-29.29150998, 14.53326521, 5.41691336] km / s>
>>> v
<Quantity [ 17.6154992 , -10.99830723, -4.20796062] km / s>
```

*Per Python ad astra ;)*

## 1.4 What’s new

### 1.4.1 New in poliastro 0.4.0

**Warning:** This version is not released yet.

This is a new major release, focused on improving stability and code quality. New angle conversion and modified equinoctial elements functions were added and an important backwards incompatible change was introduced related to classical orbital elements.

**New features:**

- **Angle conversion functions:** Finally brought back from poliastro 0.1, new functions were added to convert between true \(\nu\), eccentric \(E\) and mean \(M\) anomaly, see #45.
- **Equinoctial elements:** Now it’s possible to convert between classical and equinoctial elements, as well as from/to position and velocity vectors, see #61.
- **Numerical propagation:** A new propagator using SciPy Dormand & Prince 8(5,3) integrator was added, see #64.

**Other highlights:**

- **MIT license:** The project has been relicensed to a more popular license. poliastro remains commercial-friendly through a permissive, OSI-approved license.
- **Python 3.5 and NumPy 1.10 compatibility.** poliastro retains compatibility with legacy Python (Python 2) and NumPy 1.9. *Next version will be Python 3 only.*

**Bugs fixed:**

- **Issue #62:** Conversion between coe and rv is not transitive
- **Issue #69:** Incorrect plotting of certain closed orbits

**Backward incompatible changes:**

- **Creation of orbits from classical elements:** poliastro has switched to the *semilatus rectum* \(p\) instead of the semimajor axis \(a\) to define State objects, and the function has been renamed to from_classical(). Please update your programs accordingly.
Fig. 1.2: Mars Science Laboratory orbit.
• Removed specific angular momentum \( h \) property to avoid a name clash with the fourth modified equinoctial element, use \( \text{norm}(ss.h\_vec) \) instead.

### 1.4.2 New in poliastro 0.3.1

This is a new minor release, with some bug fixes backported from the main development branch.

Bugs fixed:

• Fixed installation problem in Python 2.
  
  • Issue #49: Fix velocity units in \texttt{ephem}.
  
  • Issue #50: Fixed \texttt{ZeroDivisionError} when propagating with time zero.

### 1.4.3 New in poliastro 0.3.0

This is a new major release, focused on switching to a pure Python codebase. Lambert problem solving and ephemerides computation came back, and a couple of bugs were fixed.

New features:

• **Pure Python codebase**: Forget about Fortran linking problems and nightmares on Windows, because now poliastro is a pure Python package. A new dependency, numba, was introduced to accelerate the algorithms, but poliastro will use it only if it is installed.

• **Lambert problem solving**: New module \texttt{iod} to determine an orbit given two position vectors and the time of flight.

• PR #42: **Planetary ephemerides computation**: New module \texttt{ephem} with functions to deal with SPK files and compute position and velocity vectors of the planets.

• PR #38: New method \texttt{parabolic()} to create parabolic orbits.

• New conda package: visit poliastro binstar channel!

• New organization and logo.

Bugs fixed:

• Issue #19: Fixed plotting region for parabolic orbits.

• Issue #37: Fixed creation of parabolic orbits.

### 1.4.4 New in poliastro 0.2.1

This is a bugfix release, no new features were introduced since 0.2.0.

• Fixed #35 (failing tests with recent astropy versions), thanks to Sam Dupree for the bug report.

• Updated for recent Sphinx versions.

### 1.4.5 New in poliastro 0.2

• **Totally refactored code** to provide a more pythonic API (see PR #14 and wiki for further information) heavily inspired by Plyades by Helge Eichhorn.

  – Mandatory use of \texttt{physical units} through \texttt{astropy.units}.
– Object-oriented approach: State and Maneuver classes.
– Vector quantities: results not only have magnitude now, but also direction (see for example maneuvers).

• Easy plotting of orbits in two dimensions using matplotlib.
• Module example with sample data to start testing the library.

These features were removed temporarily not to block the release and will see the light again in poliastro 0.3:

• Conversion between anomalies.

• Ephemerides calculations, will look into Skyfield and the JPL ephemerides prepared by Brandon Rhodes (see issue #4).

• Lambert problem solving.
• Perturbation analysis.

1.5 References

Nanos gigantum humeris insidentes.

1.5.1 Books and papers

Several books and articles are mentioned across the documentation and the source code itself. Here is the complete list in no particular order:


1.5.2 Software

poliastro wouldn’t be possible without the tremendous, often unpaid and unrecognised effort of thousands of volunteers who devote a significant part of their lives to provide the best software money can buy, for free. This is a list of direct poliastro dependencies with a citeable resource, which doesn’t account for the fact that I have used and enjoyed free (as in freedom) operative systems, compilers, text editors, IDEs and browsers for my whole academic life.


1.6 API Reference

1.6.1 poliastro.twobody package

1.6.2 poliastro.twobody.propagation module

1.6.3 poliastro.twobody.angles module

1.6.4 poliastro.maneuver module

1.6.5 poliastro.bodies module

1.6.6 poliastro.plotting module

1.6.7 poliastro.iod module

1.6.8 poliastro.ephem module

1.6.9 poliastro.stumpff module

1.6.10 poliastro.util module