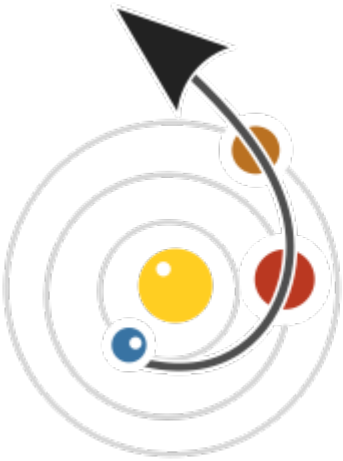
poliastro Documentation

Release 0.12.dev0

Juan Luis Cano Rodríguez

Dec 25, 2018

Contents



poliastro

Astrodynamics in Python

poliastro is an open source (MIT) collection of Python functions useful in Astrodynamics and Orbital Mechanics, focusing on interplanetary applications. It provides a simple and intuitive API and handles physical quantities with units.

View source code of poliastro!

Some of its awesome features are:

- Analytical and numerical orbit propagation
- Conversion between position and velocity vectors and classical orbital elements
- Coordinate frame transformations
- Hohmann and bielliptic maneuvers computation
- Trajectory plotting
- Initial orbit determination (Lambert problem)
- Planetary ephemerides (using SPICE kernels via Astropy)
- Computation of Near-Earth Objects (NEOs)

And more to come!

poliastro is developed by an open, international community. Release announcements and general discussion take place on our mailing list and chat.

The source code, issue tracker and wiki are hosted on GitHub, and all contributions and feedback are more than welcome. You can test poliastro in your browser using binder, a cloud Jupyter notebook server: See benchmarks for the performance analysis of poliastro.

poliastro works on recent versions of Python and is released under the MIT license, hence allowing commercial use of the library.

```
import matplotlib.pyplot as plt
plt.ion()

from poliastro.examples import molniya
from poliastro.plotting import plot

plot(molniya)
```

Contents

CHAPTER 1

Success stories

"My team and I used Poliastro for our final project in our Summer App Space program. This module helped us in plotting asteroids by using the data provided to us. It was very challenging finding a module that can take orbits from the orbital elements, plot planets, and multiple ones. This module helped us because we were able to understand the code as most of us were beginners and make some changes the way we wanted our project to turn out. We made small changes such as taking out the axis and creating a function that will create animations. I am happy we used Poliastro because it helped us directs us in a direction where we were satisfied of our final product."

—Nayeli Ju (2017)

"We are a group of students at University of Illinois at Urbana-Champaign, United States. We are currently working on a student AIAA/AAS satellite competition to design a satellite perform some science missions on asteroid (469219) 2016 HO3. We are using your poliastro python package in designing and visualizing the trajectory from GEO into asteroid's orbit. Thank you for your work on poliastro, especially the APIs that are very clear and informational, which helps us significantly."

—Yufeng Luo (University of Illinois at Urbana-Champaign, United States, 2017)

"We, at the Institute of Space and Planetary Astrophysics (ISPA, University of Karachi), are using Poliastro as part of Space Flight Dynamics Text Book development program. The idea is to develop a book suitable for undergrad students which will not only cover theoretical background but will also focus on some computational tools. We chose Poliastro as one of the packages because it was very well written and provided results with good accuracy. It is especially useful in covering some key topics like the Lambert's problem. We support the use of Poliastro and open source software because they are easily accessible to students (without any charges, unlike some other tools). A great plus point for Poliastro is that it is Python based and Python is now becoming a very important tool in areas related to Space Sciences and Technologies."

—Prof. Jawed iqbal, Syed Faisal ur Rahman (ISPA, University of Karachi, 2016)

CHAPTER 2

Contents

2.1 About poliastro

2.1.1 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 a 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.

2.1.2 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.

2.1.3 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.
- beyond: A young flight dynamics library written in Python with a focus on developing "a simple API for space observations". Some parts overlap with poliastro, but it also introduces many interesting features, and the examples look promising. Worth checking!
- SpiceyPy: This Python library wraps the SPICE Toolkit, a huge software collection developed by NASA which offers advanced astrodynamics functionality. Among all the wrappers available on the Internet, at the time of writing this is the most advanced and well-maintained one, although there are others.

2.1.4 Future ideas

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

- 3D plotting of orbits
- Continuous thrust maneuvers
- Tisserand graphs
- Porkchop plots

2.1.5 Note of the original author

I am Juan Luis Cano Rodríguez (two names and two surnames, it's the Spanish way!), an Aerospace Engineer with a passion for Astrodynamics and the Open Source world. Before poliastro started to be a truly community project, I started it 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!*

2.2 Getting started

2.2.1 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
- pytest, for running the tests from the package

poliastro is usually tested on Linux, Windows and OS X on Python 3.5 and 3.6 against latest NumPy.

2.2.2 Installation

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

```
$ conda install poliastro --channel conda-forge
```

Note: We encourage users to use conda and the conda-forge packages for convenience, especially when developing on Windows.

If the installation fails for any reason, please open an issue in the issue tracker.

Alternative installation methods

If you don't want to use conda you can install poliastro from PyPI using pip:

```
$ pip install numpy  # Run this one first for pip 9 and older!
$ pip install poliastro
```

Finally, you can also install the latest development version of poliastro directly from GitHub:

```
$ pip install https://github.com/poliastro/poliastro/archive/master.zip
```

This is useful if there is some feature that you want to try, but we did not release it yet as a stable version. Although you might find some unpolished details, these development installations should work without problems. If you find any, please open an issue in the issue tracker.

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.

Using poliastro on JupyterLab

After the release of plotly 3.0, plotting orbits using poliastro is easier than ever.

You have to install 2 (two) extensions of JupyterLab to make your experience smooth.

$ jupyter labextension install @jupyter-widgets/jupyterlab-manager

$ jupyter labextension install @jupyterlab/plotly-extension

And as the documentation of JupyterLab Extensions states:

"In order to install JupyterLab extensions, you need to have Node.js version 4 or later installed."

2.2.3 Testing

If installed correctly, the tests can be run using pytest:

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

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

2.3 User guide

2.3.1 Defining the orbit: `Orbit` objects

The core of poliastro are the *Orbit* 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
import matplotlib.pyplot as plt
plt.ion()  # To immediately show plots

from astropy import units as u

from poliastro.bodies import Earth, Mars, Sun
from poliastro.twobody import Orbit

plt.style.use("seaborn")  # Recommended
```

From position and velocity

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

```
# Data from Curtis, example 4.3
r = [-6045, -3490, 2500] * u.km
v = [-3.457, 6.618, 2.533] * u.km / u.s

ss = Orbit.from_vectors(Earth, r, v)
```

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

- Defining vectorial physical quantities using Astropy units is very easy. The list is automatically converted to a `astropy.units.Quantity`, which is actually a subclass of NumPy arrays.

- If we display the orbit we just created, we get a string with the radius of pericenter, radius of apocenter, inclination, reference frame and attractor:

```
>>> ss
7283 x 10293 km x 153.2 deg (GCRS) orbit around Earth ()
```

- If no time is specified, then a default value is assigned:

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

- The reference frame of the orbit will be one pseudo-inertial frame around the attractor. You can retrieve it using the *frame* property:

```
>>> ss.frame
<GCRS Frame (obstime=J2000.000, obsgeoloc=(0., 0., 0.) m, obsgeovel=(0., 0., 0.)␣
↪m / s)>
```

Note: At the moment, there is no explicit way to set the reference system of an orbit. This is the focus of our next releases, so we will likely introduce changes in the near future. Please subscribe to this issue to receive updates in your inbox.

Intermezzo: quick visualization of the orbit

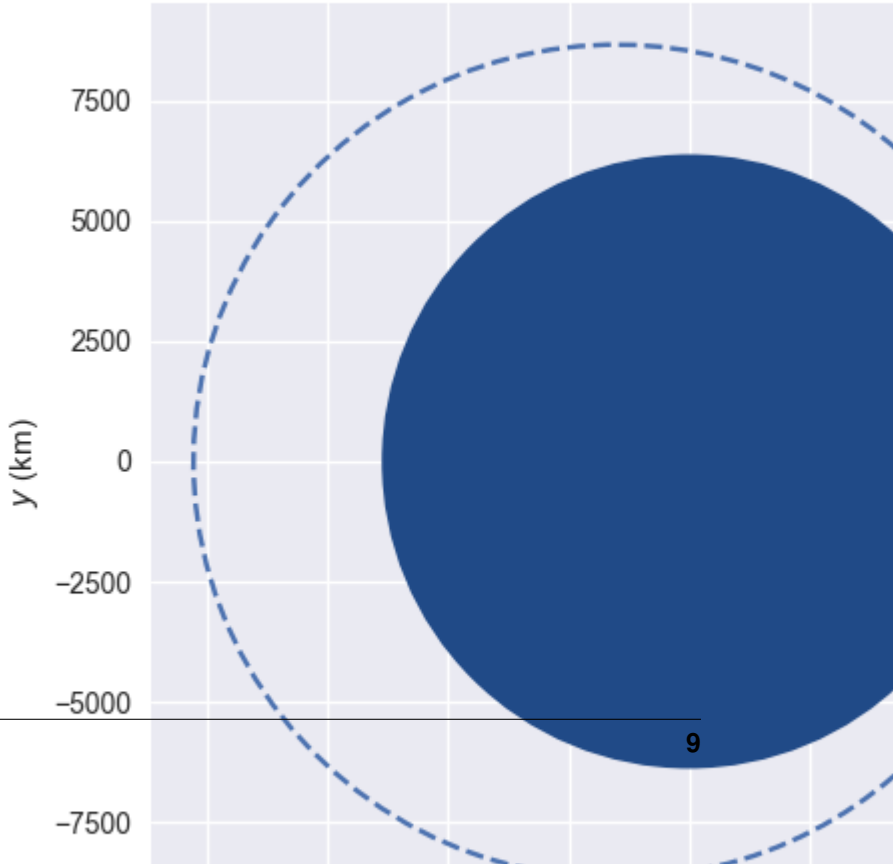
If we're working on interactive mode (for example, using the wonderful IPython notebook) we can immediately plot the current state:

```
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 \(90 \mathrm{^\circ}\) 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.



Warning: Be aware that, outside the Jupyter notebook (i.e. a normal Python interpreter or program) you might need to call `plt.show()` after

the plotting commands or `plt.ion()` before them or they won't show. Check out the Matplotlib FAQ for more information.

From classical orbital elements

We can also define a *Orbit* 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 *classical orbital elements*:

- Semimajor axis \(a\).
- Eccentricity \(e\).
- Inclination \(i\).
- Right ascension of the ascending node \(\Omega\).
- Argument of pericenter \(\omega\).
- True anomaly \(\nu\).

In this case, we'd use the method *from_classical()*:

```
# Data for␣
↪Mars at J2000 from JPL HORIZONS
a = 1.523679 * u.AU
ecc = 0.093315 * u.one
inc = 1.85 * u.deg
raan = 49.562 * u.deg
argp = 286.537 * u.deg
nu = 23.33 * u.deg

ss = Orbit.from_classical(Sun,
↪ a, ecc, inc, raan, argp, nu)
```

Notice that whether we create a `Orbit` from \(r\) and \(v\) or from elements we can access many mathematical properties individually using the *state* property of *Orbit* objects:

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

To see a complete list of properties, check out the *poliastro.twobody.orbit.Orbit* class on the API reference.

2.3.2 Moving forward in time: propagation

Now that we have defined an orbit, we might be interested in computing how is it going to evolve in the future. In the context of orbital mechanics, this process is known as **propagation**, and can be performed with the `propagate` method of *`Orbit`* objects:

```
>>> from poliastro.examples import iss
>>> iss
6772 x 6790 km x 51.6 deg (GCRS) orbit around Earth ()
>>> iss.epoch
<Time object: scale='utc' format='iso' value=2013-03-18 12:00:00.000>
>>> iss.nu.to(u.deg)
<Quantity 46.595804677061956 deg>
>>> iss.n.to(u.deg / u.min)
<Quantity 3.887010576192155 deg / min>
```

Using the *`propagate()`* method we can now retrieve the position of the ISS after some time:

```
>>> iss_30m = iss.propagate(30 * u.min)
>>> iss_30m.epoch  # Notice we advanced the epoch!
<Time object: scale='utc' format='iso' value=2013-03-18 12:30:00.000>
>>> iss_30m.nu.to(u.deg)
<Quantity 163.1409357544868 deg>
```

For more advanced propagation options, check out the *`poliastro.twobody.propagation`* module.

2.3.3 Studying non-keplerian orbits: perturbations

Apart from the Keplerian propagators, poliastro also allows the user to define custom perturbation accelerations to study non Keplerian orbits, thanks to Cowell's method:

```
>>> from poliastro.twobody.propagation import cowell
>>> from numba import njit
>>> r0 = [-2384.46, 5729.01, 3050.46] * u.km
>>> v0 = [-7.36138, -2.98997, 1.64354] * u.km / u.s
>>> initial = Orbit.from_vectors(Earth, r0, v0)
>>> @njit
... def accel(t0, state, k):
...     """Constant acceleration aligned with the velocity. """
...     v_vec = state[3:]
...     norm_v = (v_vec * v_vec).sum() ** .5
...     return 1e-5 * v_vec / norm_v
...
>>> initial.propagate(3 * u.day, method=cowell, ad=accel)
18255 x 21848 km x 28.0 deg (GCRS) orbit around Earth ()
```

Some natural perturbations are available in poliastro to be used directly in this way. For instance, let us examine the effect of J2 perturbation:

```
>>> from poliastro.core.perturbations import J2_perturbation
>>> tof = (48.0 * u.h).to(u.s)
>>> final = initial.propagate(tof, method=cowell, ad=J2_perturbation, J2=Earth.J2.
↪value, R=Earth.R.to(u.km).value)
```

The J2 perturbation changes the orbit parameters (from Curtis example 12.2):

```
>>> ((final.raan - initial.raan) / tof).to(u.deg / u.h)
<Quantity -0.17232668 deg / h>
>>> ((final.argp - initial.argp) / tof).to(u.deg / u.h)
<Quantity 0.28220397 deg / h>
```

For more available perturbation options, see the `poliastro.twobody.perturbations` module.

2.3.4 Studying artificial perturbations: thrust

In addition to natural perturbations, poliastro also has built-in artificial perturbations (thrusts) aimed at intentional change of some orbital elements. Let us simultaneously change eccentricity and inclination:

```
>>> from poliastro.twobody.thrust import change_inc_ecc
>>> from poliastro.twobody import Orbit
>>> from poliastro.bodies import Earth
>>> from poliastro.twobody.propagation import cowell
>>> from astropy import units as u
>>> from astropy.time import Time
>>> ecc_0, ecc_f = 0.4, 0.0
>>> a = 42164
>>> inc_0, inc_f = 0.0, (20.0 * u.deg).to(u.rad).value
>>> argp = 0.0
>>> f = 2.4e-7
>>> k = Earth.k.to(u.km**3 / u.s**2).value
>>> s0 = Orbit.from_classical(Earth, a * u.km, ecc_0 * u.one, inc_0 * u.deg, 0 * u.
↪deg, argp * u.deg, 0 * u.deg, epoch=Time(0, format='jd', scale='tdb'))
>>> a_d, _, _, t_f = change_inc_ecc(s0, ecc_f, inc_f, f)
>>> sf = s0.propagate(t_f * u.s, method=cowell, ad=a_d, rtol=1e-8)
```

The thrust changes orbit parameters as desired (within errors):

```
>>> sf.inc, sf.ecc
(<Quantity 0.34719734 rad>, <Quantity 0.00894513>)
```

For more available perturbation options, see the `poliastro.twobody.thrust` module.

2.3.5 Changing the orbit: `Maneuver` objects

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

Each `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 *impulse()* method or instantiating it directly.

```
from poliastro.maneuver import Maneuver

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 *hohmann()* and *bielliptic()* for Hohmann and bielliptic transfers respectively. Both return the corresponding `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:

```
>>> ss_i = Orbit.circular(Earth, alt=700 * u.km)
>>> ss_i
7078 x 7078 km x 0.0 deg (GCRS) orbit around Earth ()
>>> 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:

```
>>> 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(val.decompose([u.km, u.s]) for val in hoh[1])
(<Quantity 15729.741535747102 s>, <Quantity [ 0.        , 1.41999995, 0.        ] km /
↪ s>)
```

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

```
>>> ss_f = ss_i.apply_maneuver(hoh)
>>> ss_f
36000 x 36000 km x 0.0 deg (GCRS) orbit around Earth ()
```

2.3.6 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 `Orbit` we plot in them, projecting any further trajectories on this plane. This allows to easily visualize in two dimensions:

```
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:

2.3.7 Where are the planets? Computing ephemerides

New in version 0.3.0.

Thanks to Astropy and jplephem, 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 method `get_body_ephem()` will return a planetary orbit using low precision ephemerides available in Astropy and an `astropy.time.Time`:

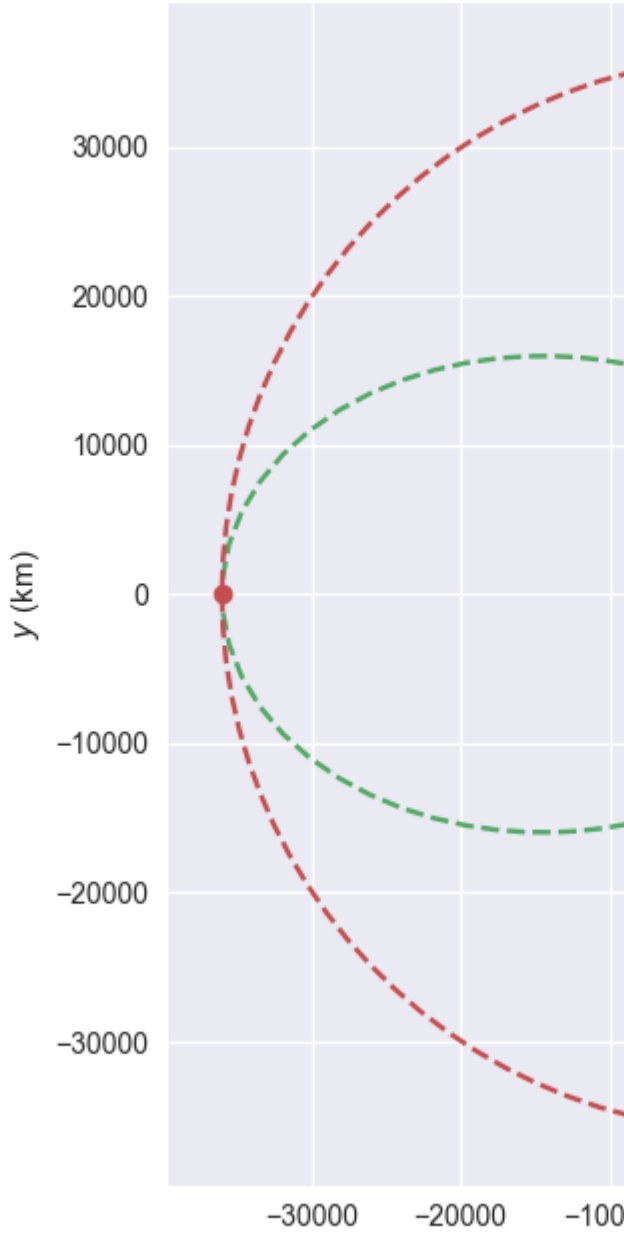


Fig. 1: Plot of a Hohmann transfer.

```
from astropy import time
epoch = time.Time("2015-05-09 10:43")  # UTC by default
```

And finally, retrieve the planet orbit:

```
>>> from poliastro import ephem
>>> Orbit.from_body_ephem(Earth, epoch)
1 x 1 AU x 23.4 deg (ICRS) orbit around Sun ()
```

This does not require any external download. If on the other hand we want to use higher precision ephemerides, we can tell Astropy to do so:

```
>>> from astropy.coordinates import solar_system_ephemeris
>>> solar_system_ephemeris.set("jpl")
Downloading http://naif.jpl.nasa.gov/pub/naif/generic_kernels/spk/planets/de430.bsp
|==========>------------------------------|  23M/119M (19.54%) ETA    59s22ss23
```

This in turn will download the ephemerides files from NASA and use them for future computations. For more information, check out Astropy documentation on ephemerides.

Note: The position and velocity vectors are given with respect to the Solar System Barycenter in the **International Celestial Reference Frame** (ICRF), which means approximately equatorial coordinates.

2.3.8 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 package *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:

```
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

ss0 = Orbit.from_body_ephem(Earth, date_launch)
ssf = Orbit.from_body_ephem(Mars, date_arrival)

from poliastro import iod
(v0, v), = iod.lambert(Sun.k, ss0.r, ssf.r, tof)
```

And these are the results:

```
>>> v0
<Quantity [-29.29150998, 14.53326521,  5.41691336] km / s>
>>> v
<Quantity [ 17.6154992 ,-10.99830723, -4.20796062] km / s>
```

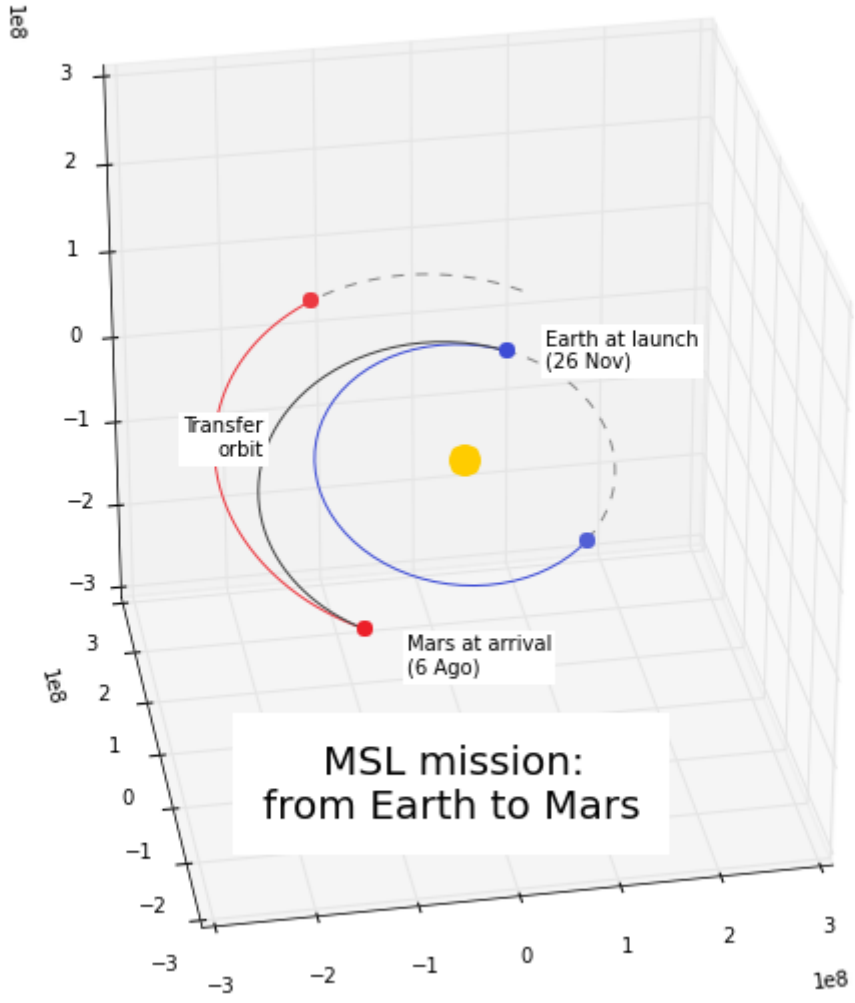


Fig. 2: Mars Science Laboratory orbit.

2.3.9 Working with NEOs

NEOs (Near Earth Objects) are asteroids and comets whose orbits are near to earth (obvious, isn't it?). More correctly, their perihelion (closest approach to the Sun) is less than 1.3 astronomical units (200 * 10^6 km). Currently, they are being an important subject of study for scientists around the world, due to their status as the relatively unchanged remains from the solar system formation process.

Because of that, a new module related to NEOs has been added to `poliastro` as part of SOCIS 2017 project.

For the moment, it is possible to search NEOs by name (also using wildcards), and get their orbits straight from NASA APIs, using `orbit_from_name()`. For example, we can get Apophis asteroid (99942 Apophis) orbit with one command, and plot it:

```
from poliastro.neos import neows

apophis_orbit = neows.orbit_from_name('apophis')  # Also '99942' or '99942 apophis'␣
↪works
earth_orbit =  Orbit.from_body_ephem(Earth)

op = OrbitPlotter()
op.plot(earth_orbit, label='Earth')
op.plot(apophis_orbit, label='Apophis')
```

Per Python ad astra ;)

2.4 Jupyter notebooks

2.5 References

Nanos gigantum humeris insidentes.

2.5.1 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:

- Vallado, David A., and Wayne D. McClain. *Fundamentals of astrodynamics and applications*. Vol. 12. Springer Science & Business Media, 2001.
- Curtis, Howard. *Orbital mechanics for engineering students*. Butterworth-Heinemann, 2013.
- Bate, Roger R., Donald D. Mueller, William W. Saylor, and Jerry E. White. *Fundamentals of astrodynamics: (dover books on physics)*. Dover publications, 2013.
- Battin, Richard H. *An introduction to the mathematics and methods of astrodynamics*. Aiaa, 1999.
- Edelbaum, Theodore N. "Propulsion requirements for controllable satellites." *ARS Journal* 31, no. 8 (1961): 1079-1089.
- Walker, M. J. H., B. Ireland, and Joyce Owens. "A set modified equinoctial orbit elements." *Celestial Mechanics* 36.4 (1985): 409-419.

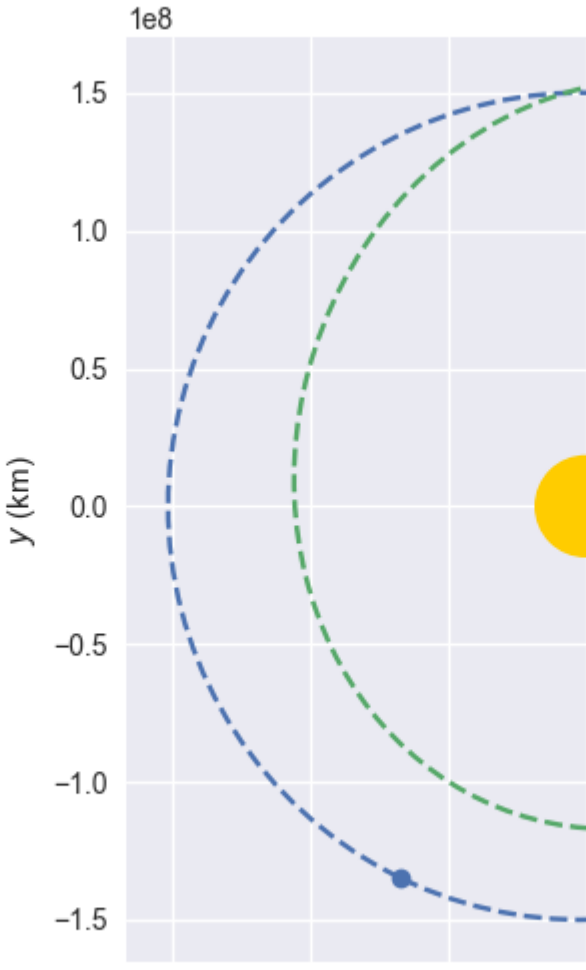


Fig. 3: Apophis asteroid orbit compared to Earth orbit.

2.5.2 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.

- Van Der Walt, Stefan, S. Chris Colbert, and Gael Varoquaux. "The NumPy array: a structure for efficient numerical computation." *Computing in Science & Engineering* 13, no. 2 (2011): 22-30. DOI:10.1109/MCSE.2011.37
- Jones, Eric, Travis Oliphant, and Pearu Peterson. "SciPy: Open Source Scientific Tools for Python", 2001-, http://www.scipy.org/ [Online; accessed 2015-12-12].
- Hunter, John D. "Matplotlib: A 2D graphics environment." *Computing in science and engineering* 9, no. 3 (2007): 90-95. DOI:10.1109/MCSE.2007.55
- Pérez, Fernando, and Brian E. Granger. "IPython: a system for interactive scientific computing." *Computing in Science & Engineering* 9, no. 3 (2007): 21-29. DOI:10.1109/MCSE.2007.53
- Robitaille, Thomas P., Erik J. Tollerud, Perry Greenfield, Michael Droettboom, Erik Bray, Tom Aldcroft, Matt Davis et al. "Astropy: A community Python package for astronomy." *Astronomy & Astrophysics* 558 (2013): A33. DOI:10.1051/0004-6361/201322068

2.6 API Reference

2.6.1 High level API

poliastro.twobody package

poliastro.twobody.angles module

Angles and anomalies.

poliastro.twobody.angles.**D_to_nu**(*D*)
True anomaly from parabolic eccentric anomaly.

Parameters **D** (*Quantity*) – Eccentric anomaly.

Returns **nu** – True anomaly.

Return type Quantity

Notes

Taken from Farnocchia, Davide, Davide Bracali Cioci, and Andrea Milani. "Robust resolution of Kepler's equation in all eccentricity regimes." Celestial Mechanics and Dynamical Astronomy 116, no. 1 (2013): 21-34.

poliastro.twobody.angles.**nu_to_D**(*nu*)
Parabolic eccentric anomaly from true anomaly.

Parameters **nu** (*Quantity*) – True anomaly.

Returns **D** – Hyperbolic eccentric anomaly.

Return type Quantity

Notes

Taken from Farnocchia, Davide, Davide Bracali Cioci, and Andrea Milani. "Robust resolution of Kepler's equation in all eccentricity regimes." Celestial Mechanics and Dynamical Astronomy 116, no. 1 (2013): 21-34.

poliastro.twobody.angles.**nu_to_E**(*nu*, *ecc*)
Eccentric anomaly from true anomaly.

New in version 0.4.0.

Parameters

- **nu** (*Quantity*) – True anomaly.
- **ecc** (*Quantity*) – Eccentricity.

Returns **E** – Eccentric anomaly.

Return type Quantity

poliastro.twobody.angles.**nu_to_F**(*nu*, *ecc*)
Hyperbolic eccentric anomaly from true anomaly.

Parameters

- **nu** (*Quantity*) – True anomaly.
- **ecc** (*Quantity*) – Eccentricity (>1).

Returns **F** – Hyperbolic eccentric anomaly.

Return type Quantity

Note: Taken from Curtis, H. (2013). *Orbital mechanics for engineering students*. 167

---

poliastro.twobody.angles.**E_to_nu**(*E*, *ecc*)
True anomaly from eccentric anomaly.

New in version 0.4.0.

Parameters

- **E** (*Quantity*) – Eccentric anomaly.
- **ecc** (*Quantity*) – Eccentricity.

Returns **nu** – True anomaly.

Return type Quantity

poliastro.twobody.angles.**F_to_nu**(*F*, *ecc*)
True anomaly from hyperbolic eccentric anomaly.

Parameters

- **F** (*Quantity*) – Hyperbolic eccentric anomaly.
- **ecc** (*Quantity*) – Eccentricity (>1).

Returns **nu** – True anomaly.

Return type Quantity

poliastro.twobody.angles.**M_to_E**(*M*, *ecc*)
Eccentric anomaly from mean anomaly.

New in version 0.4.0.

Parameters

- **M** (*Quantity*) – Mean anomaly.
- **ecc** (*Quantity*) – Eccentricity.

Returns **E** – Eccentric anomaly.

Return type Quantity

poliastro.twobody.angles.**M_to_F**(*M*, *ecc*)
Hyperbolic eccentric anomaly from mean anomaly.

Parameters

- **M** (*Quantity*) – Mean anomaly.
- **ecc** (*Quantity*) – Eccentricity (>1).

Returns **F** – Hyperbolic eccentric anomaly.

Return type Quantity

poliastro.twobody.angles.**M_to_D**(*M*, *ecc*)
Parabolic eccentric anomaly from mean anomaly.

Parameters

- **M** (*Quantity*) – Mean anomaly.
- **ecc** (*Quantity*) – Eccentricity (>1).

Returns **D** – Parabolic eccentric anomaly.

Return type Quantity

poliastro.twobody.angles.**E_to_M**(*E*, *ecc*)
Mean anomaly from eccentric anomaly.

New in version 0.4.0.

Parameters

- **E** (*Quantity*) – Eccentric anomaly.
- **ecc** (*Quantity*) – Eccentricity.

Returns **M** – Mean anomaly.

Return type Quantity

poliastro.twobody.angles.**F_to_M**(*F*, *ecc*)
Mean anomaly from eccentric anomaly.

Parameters

- **F** (*Quantity*) – Hyperbolic eccentric anomaly.
- **ecc** (*Quantity*) – Eccentricity (>1).

Returns **M** – Mean anomaly.

Return type Quantity

poliastro.twobody.angles.**D_to_M**(*D*, *ecc*)
Mean anomaly from eccentric anomaly.

Parameters

- **D** (*Quantity*) – Parabolic eccentric anomaly.
- **ecc** (*Quantity*) – Eccentricity.

Returns **M** – Mean anomaly.

Return type Quantity

poliastro.twobody.angles.**M_to_nu**(*M*, *ecc*, *delta=0.01*)
True anomaly from mean anomaly.

New in version 0.4.0.

Parameters

- **M** (*Quantity*) – Mean anomaly.
- **ecc** (*Quantity*) – Eccentricity.
- **delta** (*float (optional)*) – threshold of near-parabolic regime definition (from Davide Farnocchia et al)

Returns **nu** – True anomaly.

Return type Quantity

Examples

```
>>> M_to_nu(30.0 * u.deg, 0.06 * u.one)
<Quantity 33.67328493 deg>
```

poliastro.twobody.angles.**nu_to_M**(*nu*, *ecc*, *delta=0.01*)
Mean anomaly from true anomaly.

New in version 0.4.0.

Parameters

- **nu** (*Quantity*) – True anomaly.
- **ecc** (*Quantity*) – Eccentricity.
- **delta** (*float (optional)*) – threshold of near-parabolic regime definition (from Davide Farnocchia et al)

Returns **M** – Mean anomaly.

Return type Quantity

poliastro.twobody.angles.**fp_angle**(*nu*, *ecc*)
Flight path angle.

New in version 0.4.0.

Parameters

- **nu** (*Quantity*) – True anomaly.
- **ecc** (*Quantity*) – Eccentricity.

Note: Algorithm taken from Vallado 2007, pp. 113.

poliastro.twobody.classical module

Functions to define orbits from classical orbital elements.

class poliastro.twobody.classical.**ClassicalState**(*attractor*, *p*, *ecc*, *inc*, *raan*, *argp*, *nu*)
State defined by its classical orbital elements.

p
Semilatus rectum.

ecc
Eccentricity.

inc
Inclination.

raan
Right ascension of the ascending node.

argp
Argument of the perigee.

nu
True anomaly.

to_vectors()
Converts to position and velocity vector representation.

to_classical()
Converts to classical orbital elements representation.

to_equinoctial()
Converts to modified equinoctial elements representation.

poliastro.twobody.decorators module

Decorators.

poliastro.twobody.decorators.**state_from_vector**(*func*)
Changes signature to receive Orbit instead of state array.

Examples

```
>>> from poliastro.twobody.decorators import state_from_vector
>>> @state_from_vector
... def func(_, ss):
...     return ss.r, ss.v
...
>>> func(0.0, [1, 2, 3, -1, -2, -3], 1.0)
(<Quantity [1., 2., 3.] km>, <Quantity [-1., -2., -3.] km / s>)
```

Notes

Functions decorated with this will have poor performance.

poliastro.twobody.equinoctial module

Functions to define orbits from modified equinoctial orbital elements.

poliastro.twobody.equinoctial.**mee2coe**(*p*, *f*, *g*, *h*, *k*, *L*)
Converts from modified equinoctial orbital elements to classical orbital elements.

The definition of the modified equinoctial orbital elements is taken from [Walker, 1985].

Note: The conversion is always safe because arctan2 works also for 0, 0 arguments.

---

class poliastro.twobody.equinoctial.**ModifiedEquinoctialState**(*attractor*, *p*, *f*, *g*, *h*, *k*, *L*)

p
Semimajor axis.

f
Second modified equinoctial element.

g
Third modified equinoctial element.

h
Fourth modified equinoctial element.

k
Fifth modified equinoctial element.

L
True longitude.

to_classical()
Converts to classical orbital elements representation.

poliastro.twobody.orbit module

exception poliastro.twobody.orbit.**TimeScaleWarning**

class poliastro.twobody.orbit.**Orbit**(*state*, *epoch*, *plane*)
Position and velocity of a body with respect to an attractor at a given time (epoch).

Regardless of how the Orbit is created, the implicit reference system is an inertial one. For the specific case of the Solar System, this can be assumed to be the International Celestial Reference System or ICRS.

state
Position and velocity or orbital elements.

epoch
Epoch of the orbit.

plane
Fundamental plane of the frame.

frame
Reference frame of the orbit.

New in version 0.11.0.

classmethod from_vectors(*attractor*, *r*, *v*, *epoch=<Time object: scale='tt' format='jyear_str' value=J2000.000>*, *plane=<Planes.EARTH_EQUATOR: 'Earth mean Equator and Equinox of epoch (J2000.0)'>*)
Return *Orbit* from position and velocity vectors.

Parameters

- **attractor** (Body) – Main attractor.
- **r** (*Quantity*) – Position vector wrt attractor center.
- **v** (*Quantity*) – Velocity vector.
- **epoch** (*Time, optional*) – Epoch, default to J2000.
- **plane** (Planes) – Fundamental plane of the frame.

classmethod from_classical(*attractor*, *a*, *ecc*, *inc*, *raan*, *argp*, *nu*, *epoch=<Time object: scale='tt' format='jyear_str' value=J2000.000>*, *plane=<Planes.EARTH_EQUATOR: 'Earth mean Equator and Equinox of epoch (J2000.0)'>*)
Return *Orbit* from classical orbital elements.

Parameters

- **attractor** (Body) – Main attractor.
- **a** (*Quantity*) – Semi-major axis.
- **ecc** (*Quantity*) – Eccentricity.
- **inc** (*Quantity*) – Inclination
- **raan** (*Quantity*) – Right ascension of the ascending node.
- **argp** (*Quantity*) – Argument of the pericenter.
- **nu** (*Quantity*) – True anomaly.
- **epoch** (*Time, optional*) – Epoch, default to J2000.
- **plane** (Planes) – Fundamental plane of the frame.

classmethod from_equinoctial(*attractor*, *p*, *f*, *g*, *h*, *k*, *L*, *epoch=<Time object: scale='tt' format='jyear_str' value=J2000.000>*, *plane=<Planes.EARTH_EQUATOR: 'Earth mean Equator and Equinox of epoch (J2000.0)'>*)
Return *Orbit* from modified equinoctial elements.

Parameters

- **attractor** (Body) – Main attractor.
- **p** (*Quantity*) – Semilatus rectum.
- **f** (*Quantity*) – Second modified equinoctial element.
- **g** (*Quantity*) – Third modified equinoctial element.
- **h** (*Quantity*) – Fourth modified equinoctial element.
- **k** (*Quantity*) – Fifth modified equinoctial element.

- **L** (*Quantity*) – True longitude.
- **epoch** (*Time, optional*) – Epoch, default to J2000.
- **plane** (Planes) – Fundamental plane of the frame.

classmethod from_body_ephem(*body*, *epoch=None*)
Return osculating *Orbit* of a body at a given time.

classmethod circular(*attractor*, *alt*, *inc=<Quantity 0. deg>*, *raan=<Quantity 0. deg>*, *arglat=<Quantity 0. deg>*, *epoch=<Time object: scale='tt' format='jyear_str' value=J2000.000>*, *plane=<Planes.EARTH_EQUATOR: 'Earth mean Equator and Equinox of epoch (J2000.0)'>*)
Return circular *Orbit*.

Parameters

- **attractor** (Body) – Main attractor.
- **alt** (*Quantity*) – Altitude over surface.
- **inc** (*Quantity, optional*) – Inclination, default to 0 deg (equatorial orbit).
- **raan** (*Quantity, optional*) – Right ascension of the ascending node, default to 0 deg.
- **arglat** (*Quantity, optional*) – Argument of latitude, default to 0 deg.
- **epoch** (*Time, optional*) – Epoch, default to J2000.
- **plane** (Planes) – Fundamental plane of the frame.

classmethod parabolic(*attractor*, *p*, *inc*, *raan*, *argp*, *nu*, *epoch=<Time object: scale='tt' format='jyear_str' value=J2000.000>*, *plane=<Planes.EARTH_EQUATOR: 'Earth mean Equator and Equinox of epoch (J2000.0)'>*)
Return parabolic *Orbit*.

Parameters

- **attractor** (Body) – Main attractor.
- **p** (*Quantity*) – Semilatus rectum or parameter.
- **inc** (*Quantity, optional*) – Inclination.
- **raan** (*Quantity*) – Right ascension of the ascending node.
- **argp** (*Quantity*) – Argument of the pericenter.
- **nu** (*Quantity*) – True anomaly.
- **epoch** (*Time, optional*) – Epoch, default to J2000.
- **plane** (Planes) – Fundamental plane of the frame.

represent_as(*representation*)
Converts the orbit to a specific representation.

New in version 0.11.0.

Parameters **representation** (*BaseRepresentation*) – Representation object to use. It must be a class, not an instance.

Examples

```
>>> from poliastro.examples import iss
>>> from astropy.coordinates import CartesianRepresentation,␣
↪SphericalRepresentation
>>> iss.represent_as(CartesianRepresentation)
<CartesianRepresentation (x, y, z) in km
    (859.07256, -4137.20368, 5295.56871)
 (has differentials w.r.t.: 's')>
>>> iss.represent_as(CartesianRepresentation).xyz
<Quantity [  859.07256, -4137.20368,  5295.56871] km>
>>> iss.represent_as(CartesianRepresentation).differentials['s']
<CartesianDifferential (d_x, d_y, d_z) in km / s
    (7.37289205, 2.08223573, 0.43999979)>
>>> iss.represent_as(CartesianRepresentation).differentials['s'].d_xyz
<Quantity [7.37289205, 2.08223573, 0.43999979] km / s>
>>> iss.represent_as(SphericalRepresentation)
<SphericalRepresentation (lon, lat, distance) in (rad, rad, km)
    (4.91712525, 0.89732339, 6774.76995296)
 (has differentials w.r.t.: 's')>
```

to_icrs()
Creates a new Orbit object with its coordinates transformed to ICRS.

Notice that, strictly speaking, the center of ICRS is the Solar System Barycenter and not the Sun, and therefore these orbits cannot be propagated in the context of the two body problem. Therefore, this function exists merely for practical purposes.

New in version 0.11.0.

propagate(*value*, *method=<function mean_motion>*, *rtol=1e-10*, ***kwargs*)
Propagates an orbit.

If value is true anomaly, propagate orbit to this anomaly and return the result. Otherwise, if time is provided, propagate this *Orbit* some *time* and return the result.

Parameters

- **value** (*Multiple options*) – True anomaly values or time values. If given an angle, it will always propagate forward.
- **rtol** (*float, optional*) – Relative tolerance for the propagation algorithm, default to 1e-10.
- **method** (*function, optional*) – Method used for propagation
- ****kwargs** – parameters used in perturbation models

sample(*values=None*, *method=<function mean_motion>*)
Samples an orbit to some specified time values.

New in version 0.8.0.

Parameters

- **values** (*Multiple options*) – Number of interval points (default to 100), True anomaly values, Time values.
- **method** (*function, optional*) – Method used for propagation

Returns **positions** – Array of x, y, z positions, with proper times as the frame attributes if supported.

Return type BaseCoordinateFrame

Notes

When specifying a number of points, the initial and final position is present twice inside the result (first and last row). This is more useful for plotting.

Examples

```
>>> from astropy import units as u
>>> from poliastro.examples import iss
>>> iss.sample()  # doctest: +ELLIPSIS
<GCRS Coordinate ...>
>>> iss.sample(10)  # doctest: +ELLIPSIS
<GCRS Coordinate ...>
>>> iss.sample([0, 180] * u.deg)  # doctest: +ELLIPSIS
<GCRS Coordinate ...>
>>> iss.sample([0, 10, 20] * u.minute)  # doctest: +ELLIPSIS
<GCRS Coordinate ...>
>>> iss.sample([iss.epoch + iss.period / 2])  # doctest: +ELLIPSIS
<GCRS Coordinate ...>
```

apply_maneuver(*maneuver*, *intermediate=False*)

Returns resulting *Orbit* after applying maneuver to self.

Optionally return intermediate states (default to False).

Parameters

- **maneuver** (Maneuver) – Maneuver to apply.
- **intermediate** (*bool, optional*) – Return intermediate states, default to False.

poliastro.twobody.propagation module

Propagation algorithms

poliastro.twobody.propagation.**func_twobody**(*t0*, *u_*, *k*, *ad*, *ad_kwargs*)

Differential equation for the initial value two body problem.

This function follows Cowell's formulation.

Parameters

- **t0** (*float*) – Time.
- **u** (*ndarray*) – Six component state vector [x, y, z, vx, vy, vz] (km, km/s).
- **k** (*float*) – Standard gravitational parameter.
- **ad** (*function(t0, u, k)*) – Non Keplerian acceleration (km/s2).
- **ad_kwargs** (*optional*) – perturbation parameters passed to ad

poliastro.twobody.propagation.**cowell**(*orbit*, *tof*, *rtol=1e-11*, ***, *ad=None*, ***ad_kwargs*)

Propagates orbit using Cowell's formulation.

Parameters

- **orbit** (Orbit) – the Orbit object to propagate.

- **ad** (*function(t0, u, k), optional*) – Non Keplerian acceleration (km/s2), default to None.
- **tof** (*Multiple options*) – Time to propagate, float (s), Times to propagate, array of float (s).
- **rtol** (*float, optional*) – Maximum relative error permitted, default to 1e-10.

Raises RuntimeError – If the algorithm didn't converge.

Note: This method uses a Dormand & Prince method of order 8(5,3) available in the poliastro.integrators module. If multiple tofs are provided, the method propagates to the maximum value and calculates the other values via dense output

---

poliastro.twobody.propagation.**propagate**(*orbit*, *time_of_flight*, *, *method=<function mean_motion>*, *rtol=1e-10*, ***kwargs*)

Propagate an orbit some time and return the result.

poliastro.twobody.rv module

Functions to define orbits from position and velocity vectors.

class poliastro.twobody.rv.**RVState**(*attractor*, *r*, *v*)

State defined by its position and velocity vectors.

r

Position vector.

v

Velocity vector.

to_vectors()

Converts to position and velocity vector representation.

to_classical()

Converts to classical orbital elements representation.

poliastro.twobody.thrust package

poliastro.twobody.thrust.change_a_inc.**change_a_inc**(*k*, *a_0*, *a_f*, *inc_0*, *inc_f*, *f*)

Guidance law from the Edelbaum/Kéchichian theory, optimal transfer between circular inclined orbits (a_0, i_0) –> (a_f, i_f), ecc = 0.

Parameters

- **k** (*float*) – Gravitational parameter.
- **a_0** (*float*) – Initial semimajor axis.
- **a_f** (*float*) – Final semimajor axis.
- **inc_0** (*float*) – Initial inclination.
- **inc_f** (*float*) – Final inclination.
- **f** (*float*) – Magnitude of constant acceleration

Notes

Edelbaum theory, reformulated by Kéchichian.

References

- Edelbaum, T. N. "Propulsion Requirements for Controllable Satellites", 1961.
- Kéchichian, J. A. "Reformulation of Edelbaum's Low-Thrust Transfer Problem Using Optimal Control Theory", 1997.

Argument of perigee change, with formulas developed by Pollard.

References

- Pollard, J. E. "Simplified Approach for Assessment of Low-Thrust Elliptical Orbit Transfers", 1997.
- Pollard, J. E. "Evaluation of Low-Thrust Orbital Maneuvers", 1998.

poliastro.twobody.thrust.change_argp.**change_argp**(*k*, *a*, *ecc*, *argp_0*, *argp_f*, *f*)

Guidance law from the model. Thrust is aligned with an inertially fixed direction perpendicular to the semimajor axis of the orbit.

Parameters **f** (*float*) – Magnitude of constant acceleration

Quasi optimal eccentricity-only change, with formulas developed by Pollard.

References

- Pollard, J. E. "Simplified Approach for Assessment of Low-Thrust Elliptical Orbit Transfers", 1997.

poliastro.twobody.thrust.change_ecc_quasioptimal.**change_ecc_quasioptimal**(*ss_0*, *ecc_f*, *f*)

Guidance law from the model. Thrust is aligned with an inertially fixed direction perpendicular to the semimajor axis of the orbit.

Parameters

- **ss_0** (Orbit) – Initial orbit, containing all the information.
- **ecc_f** (*float*) – Final eccentricity.
- **f** (*float*) – Magnitude of constant acceleration

Simultaneous eccentricity and inclination changes.

References

- Pollard, J. E. "Simplified Analysis of Low-Thrust Orbital Maneuvers", 2000.

poliastro.twobody.thrust.change_inc_ecc.**change_inc_ecc**(*ss_0*, *ecc_f*, *inc_f*, *f*)

Guidance law from the model. Thrust is aligned with an inertially fixed direction perpendicular to the semimajor axis of the orbit.

Parameters

- **ss_0** (Orbit) – Initial orbit, containing all the information.

- **ecc_f** (*float*) – Final eccentricity.
- **inc_f** (*float*) – Final inclination.
- **f** (*float*) – Magnitude of constant acceleration.

poliastro.iod package

poliastro.iod.izzo module

Izzo's algorithm for Lambert's problem

poliastro.iod.izzo.**lambert**(*k*, *r0*, *r*, *tof*, *M=0*, *numiter=35*, *rtol=1e-08*)

Solves the Lambert problem using the Izzo algorithm.

New in version 0.5.0.

Parameters

- **k** (*Quantity*) – Gravitational constant of main attractor (km^3 / s^2).
- **r0** (*Quantity*) – Initial position (km).
- **r** (*Quantity*) – Final position (km).
- **tof** (*Quantity*) – Time of flight (s).
- **M** (*int, optional*) – Number of full revolutions, default to 0.
- **numiter** (*int, optional*) – Maximum number of iterations, default to 35.
- **rtol** (*float, optional*) – Relative tolerance of the algorithm, default to 1e-8.

Yields **v0, v** (*tuple*) – Pair of velocity solutions.

poliastro.iod.vallado module

Initial orbit determination.

poliastro.iod.vallado.**lambert**(*k*, *r0*, *r*, *tof*, *short=True*, *numiter=35*, *rtol=1e-08*)

Solves the Lambert problem.

New in version 0.3.0.

Parameters

- **k** (*Quantity*) – Gravitational constant of main attractor (km^3 / s^2).
- **r0** (*Quantity*) – Initial position (km).
- **r** (*Quantity*) – Final position (km).
- **tof** (*Quantity*) – Time of flight (s).
- **short** (*boolean, optional*) – Find out the short path, default to True. If False, find long path.
- **numiter** (*int, optional*) – Maximum number of iterations, default to 35.
- **rtol** (*float, optional*) – Relative tolerance of the algorithm, default to 1e-8.

Raises RuntimeError – If it was not possible to compute the orbit.

Note: This uses the universal variable approach found in Battin, Mueller & White with the bisection iteration suggested by Vallado. Multiple revolutions not supported.

poliastro.neos package

Code related to NEOs.

Functions related to NEOs and different NASA APIs. All of them are coded as part of SOCIS 2017 proposal.

Notes

The orbits returned by the functions in this package are in the *HeliocentricEclipticJ2000* frame.

poliastro.neos.dastcom5 module

NEOs orbit from DASTCOM5 database.

poliastro.neos.dastcom5.**asteroid_db**()
Return complete DASTCOM5 asteroid database.

Returns **database** – Database with custom dtype.

Return type numpy.ndarray

poliastro.neos.dastcom5.**comet_db**()
Return complete DASTCOM5 comet database.

Returns **database** – Database with custom dtype.

Return type numpy.ndarray

poliastro.neos.dastcom5.**orbit_from_name**(*name*)
Return *Orbit* given a name.

Retrieve info from JPL DASTCOM5 database.

Parameters **name** (*str*) – NEO name.

Returns **orbit** – NEO orbits.

Return type list (*Orbit*)

poliastro.neos.dastcom5.**orbit_from_record**(*record*)
Return *Orbit* given a record.

Retrieve info from JPL DASTCOM5 database.

Parameters **record** (*int*) – Object record.

Returns **orbit** – NEO orbit.

Return type *Orbit*

poliastro.neos.dastcom5.**record_from_name**(*name*)
Search *dastcom.idx* and return logical records that match a given string.

Body name, SPK-ID, or alternative designations can be used.

Parameters **name** (*str*) – Body name.

Returns **records** – DASTCOM5 database logical records matching str.

Return type list (int)

poliastro.neos.dastcom5.**string_record_from_name**(*name*)
Search *dastcom.idx* and return body full record.

Search DASTCOM5 index and return body records that match string, containing logical record, name, alternative designations, SPK-ID, etc.

Parameters **name** (*str*) – Body name.

Returns **lines** – Body records

Return type list(str)

poliastro.neos.dastcom5.**read_headers**()
Read *DASTCOM5* headers and return asteroid and comet headers.

Headers are two numpy arrays with custom dtype.

Returns **ast_header, com_header** – DASTCOM5 headers.

Return type tuple (numpy.ndarray)

poliastro.neos.dastcom5.**read_record**(*record*)
Read *DASTCOM5* record and return body data.

Body data consists of numpy array with custom dtype.

Parameters **record** (*int*) – Body record.

Returns **body_data** – Body information.

Return type numpy.ndarray

poliastro.neos.dastcom5.**download_dastcom5**()
Downloads DASTCOM5 database.

Downloads and unzip DASTCOM5 file in default poliastro path (~/.poliastro).

poliastro.neos.dastcom5.**entire_db**()
Return complete DASTCOM5 database.

Merge asteroid and comet databases, only with fields related to orbital data, discarding the rest.

Returns **database** – Database with custom dtype.

Return type numpy.ndarray

dastcom5 parameters

Dastcom5 parameters

avail:

- *a* if it is available for asteroids.
- *c* if it is available for comets.
- *[number]+* since which version of DASTCOM5 is available.

avail	Label	Definition
ac/3+	EPOCH	Time of osc. orbital elements solution, JD (CT,TDB)
ac/3+	CALEPO	Time of osc. orbital elements solution, YYYYDDMM.ffff
ac/3+	MA	Mean anomaly at EPOCH, deg (elliptical & hyperbolic cases “9.999999E99” if not available)
ac/3+	W	Argument of periapsis at EPOCH, J2000 ecliptic, deg.
ac/3+	OM	Longitude of ascending node at EPOCH, J2000 ecliptic,deg.
ac/3+	IN	Inclination angle at EPOCH wrt J2000 ecliptic, deg.
ac/3+	EC	Eccentricity at EPOCH
ac/3+	A	Semi-major axis at EPOCH, au
ac/3+	QR	Perihelion distance at EPOCH, au
ac/3+	TP	Perihelion date for QR at EPOCH, JD (CT,TDB)
ac/3+	TPCAL	Perihelion date for QR at EPOCH, format YYYYMMDD.fff
ac/5+	TPFRAC	Decimal (fractional) part of TP for extended precision
ac/4+	SOLDAT	Date orbit solution was performed, JD (CT,TDB)
ac/4+	SRC(01)	Square root covariance vector. Vector-stored upper- triangular matrix with order {EC,QR,TP,OM,W,IN,{ ESTI
ac/3+	H	Absolute visual magnitude (IAU H-G system) (99=unknown)
ac/3+	G	Mag. slope parm. (IAU H-G)(99=unknown & 0.15 not assumed)
c/3+	M1	Total absolute magnitude, mag.
c/3+	M2	Nuclear absolute magnitue, mag.
c/4+	K1	Total absolute magnitude scaling factor
c/4+	K2	Nuclear absolute magnitude scaling factor
c/4+	PHCOF	Phase coefficient for K2= 5
ac/3+	A1	Non-grav. accel., radial component, [s:10^-8 au/day^2]
ac/3+	A2	Non-grav. accel., transverse component,[s:10^-8 au/day^2
ac/4+	A3	Non-grav. accel., normal component, [s:10^-8 au/day^2]
c/4+	DT	Non-grav. lag/delay parameter, days
ac/5+	R0	Non-grav. model constant, normalizing distance, au
ac/5+	ALN	Non-grav. model constant, normalizing factor
ac/5+	NM	Non-grav. model constant, exponent m
ac/5+	NN	Non-grav. model constant, exponent n
ac/5+	NK	Non-grav. model constant, exponent k
c/4+	S0	Center-of-light estimated offset at 1 au, km
c/5+	TCL	Center-of-light start-time offset, d since “ref.time”
a /5+	LGK	Surface thermal conductivity log_10(k), (W/m/K)
ac/5+	RHO	Bulk density, kg/m^3
ac/5+	AMRAT	Solar pressure model, area/mass ratio, m^2/kg
c/5+	AJ1	Jet 1 acceleration, au/d^2
c/5+	AJ2	Jet 2 acceleration, au/d^2
c/5+	ET1	Thrust angle, colatitude of jet 1, deg.
c/5+	ET2	Thrust angle, colatitude of jet 2, deg.
c/5+	DTH	Jet model diurnal lag angle, deg. (delta_theta)
ac/5+	ALF	Spin pole orientation, RA, deg.
ac/5+	DEL	Spin pole orientation, DEC, deg.
ac/5+	SPHLM3	Earth gravity sph. harm. model limit, Earth radii
ac/5+	SPHLM5	Jupiter grav. sph. harm. model limit, Jupiter radii
ac/3+	RP	Object rotational period, hrs
ac/3+	GM	Object mass parameter, km^3/s^2
ac/3+	RAD	Object mean radius, km
ac/5+	EXTNT1	Triaxial ellipsoid, axis 1/largest equat. extent, km
ac/5+	EXTNT2	Triaxial ellipsoid, axis 2/smallest equat. extent, km
ac/5+	EXTNT3	Triaxial ellipsoid, axis 3/polar extent, km

avail	Label	Definition
ac/4+	MOID	Earth MOID at EPOCH time, au; '99' if not computed
ac/3+	ALBEDO	Geometric visual albedo, 99 if unknown
a /3+	BVCI	B-V color index, mag., 99 if unknown
a /5+	UBCI	U-B color index, mag., 99 if unknown
a /5+	IRCI	I-R color index, mag., 99 if unknown
ac/4+	RMSW	RMS of weighted optical residuals, arcsec
ac/5+	RMSU	RMS of unweighted optical residuals, arcsec
ac/5+	RMSN	RMS of normalized optical residuals
ac/5+	RMSNT	RMS of all normalized residuals
a /5+	RMSH	RMS of abs. visual magnitude (H) residuals, mag.
c/5+	RMSMT	RMS of MT estimate residuals, mag.
c/5+	RMSMN	RMS of MN estimate residuals, mag.
ac/3+	NO	Logical record-number of this object in DASTCOM
ac/4+	NOBS	Number of observations of all types used in orbit soln.
ac/4+	OBSFRST	Start-date of observations used in fit, YYYYMMDD
ac/4+	OBSLAST	Stop-date of observations used in fit, YYYYMMDD
ac/5+	PRELTV	Planet relativity "bit-switch" byte: bits 0-7 are set to 1 if relativity for corresponding planet was computed, 0 if
ac/5+	SPHMX3	Earth grav. model max. degree; 0=point-mass, 2= J2 only, 3= up to J3 zonal, 22= 2x2 field, 33=3x3 field, etc.
ac/5+	SPHMX5	Jupiter grav. max. deg.; 0=point-mass, 2= J2 only, 3= up to J3 zonal, 22= 2x2 field, 33=3x3 field, etc.
ac/5+	JGSEP	Galilean satellites used as sep. perturbers; 0=no 1=yes
ac/5+	TWOBOD	Two-body orbit model flag; 0=no 1=yes
ac/5+	NSATS	Number of satellites; 99 if unknown.
ac/4+	UPARM	Orbit condition code; 99 if not computed
ac/4+	LSRC	Length of square-root cov. vector SRC (# elements used)
c/3+	IPYR	Perihelion year (i.e., 1976, 2012, 2018, etc.)
ac/3+	NDEL	Number of radar delay measurements used in orbit soln.
ac/3+	NDOP	Number of radar Doppler measurements used in orbit soln.
c/5+	NOBSMT	Number of magnitude measurements used in total mag. soln.
c/5+	NOBSMN	Number of magnitude measurements used in nuc. mag. soln.
c/3+	COMNUM	IAU comet number (parsed from DESIG)
ac/3+	EQUNOX	Equinox of orbital elements ('1950' or '2000')
ac/4+	PENAM	Planetary ephemeris ID/name
ac/3+	SBNAM	Small-body perturber ephemeris ID/name
a /3	SPTYPT	Tholen spectral type
a /4+	SPTYPS	SMASS-II spectral type
ac/3+	DARC	Data arc span (year-year, OR integer # of days)
a /3+	COMNT1	Asteroid comment line #1
a /3+	COMNT2	Asteroid comment line #2
c/3+	COMNT3	Comet comment line #1
c/3+	COMNT4	Comet comment line #2
ac/3+	DESIG	Object designation
ac/4+	ESTL	Dynamic parameter estimation list. Last symbol set to '+' if list is too long for field; check object record comm
ac/3+	IREF	Solution reference/ID/name
ac/3+	NAME	Object name

poliastro.neos.neows module

NEOs orbit from NEOWS and JPL SBDB

poliastro.neos.neows.**orbit_from_spk_id**(*spk_id*, *api_key=None*)

Return *Orbit* given a SPK-ID.

Retrieve info from NASA NeoWS API, and therefore it only works with NEAs (Near Earth Asteroids).

Parameters

- **spk_id** (*str*) – SPK-ID number, which is given to each body by JPL.
- **api_key** (*str*) – NASA OPEN APIs key (default: *DEMO_KEY*)

Returns **orbit** – NEA orbit.

Return type *Orbit*

poliastro.neos.neows.**spk_id_from_name**(*name*)

Return SPK-ID number given a small-body name.

Retrieve and parse HTML from JPL Small Body Database to get SPK-ID.

Parameters **name** (*str*) – Small-body object name. Wildcards "*" and/or "?" can be used.

Returns **spk_id** – SPK-ID number.

Return type str

poliastro.neos.neows.**orbit_from_name**(*name*, *api_key=None*)

Return *Orbit* given a name.

Retrieve info from NASA NeoWS API, and therefore it only works with NEAs (Near Earth Asteroids).

Parameters

- **name** (*str*) – NEA name.
- **api_key** (*str*) – NASA OPEN APIs key (default: *DEMO_KEY*)

Returns **orbit** – NEA orbit.

Return type *Orbit*

poliastro.bodies module

Bodies of the Solar System.

Contains some predefined bodies of the Solar System:

- Sun ()
- Earth ()
- Moon ()
- Mercury ()
- Venus ()
- Mars ()
- Jupiter ()
- Saturn ()
- Uranus ()
- Neptune ()
- Pluto ()

and a way to define new bodies (*Body* class).

Data references can be found in *constants*

class poliastro.bodies.**Body**(*parent*, *k*, *name*, *symbol=None*, *R=<Quantity 0. km>*, ***kwargs*)
Class to represent a generic body.

__init__(*parent*, *k*, *name*, *symbol=None*, *R=<Quantity 0. km>*, ***kwargs*)
Constructor.

Parameters

- **parent** (*Body*) – Central body.
- **k** (*Quantity*) – Standard gravitational parameter.
- **name** (*str*) – Name of the body.
- **symbol** (*str, optional*) – Symbol for the body.
- **R** (*Quantity, optional*) – Radius of the body.

poliastro.constants module

Astronomical and physics constants.

This module complements constants defined in *astropy.constants*, with gravitational paremeters and radii.

Note that *GM_jupiter* and *GM_neptune* are both referred to the whole planetary system gravitational parameter.

Unless otherwise specified, gravitational and mass parameters were obtained from:

- Luzum, Brian et al. "The IAU 2009 System of Astronomical Constants: The Report of the IAU Working Group on Numerical Standards for Fundamental Astronomy." Celestial Mechanics and Dynamical Astronomy 110.4 (2011): 293–304. Crossref. Web. DOI: 10.1007/s10569-011-9352-4

radii were obtained from:

- Archinal, B. A. et al. "Report of the IAU Working Group on Cartographic Coordinates and Rotational Elements: 2009." Celestial Mechanics and Dynamical Astronomy 109.2 (2010): 101–135. Crossref. Web. DOI: 10.1007/s10569-010-9320-4

J2 for the Sun was obtained from:

- https://hal.archives-ouvertes.fr/hal-00433235/document (New values of gravitational moments J2 and J4 deduced from helioseismology, Redouane Mecheri et al)

poliastro.coordinates module

Functions related to coordinate systems and transformations.

This module complements astropy.coordinates.

poliastro.coordinates.**body_centered_to_icrs**(*r*, *v*, *source_body*, *epoch=<Time object: scale='tt' format='jyear_str' value=J2000.000>*, *rotate_meridian=False*)
Converts position and velocity body-centered frame to ICRS.

Parameters

- **r** (*Quantity*) – Position vector in a body-centered reference frame.
- **v** (*Quantity*) – Velocity vector in a body-centered reference frame.

- **source_body** (Body) – Source body.
- **epoch** (*Time, optional*) – Epoch, default to J2000.
- **rotate_meridian** (*bool, optional*) – Whether to apply the rotation of the meridian too, default to False.

Returns **r, v** – Position and velocity vectors in ICRS.

Return type tuple (Quantity)

poliastro.coordinates.**icrs_to_body_centered**(*r*, *v*, *target_body*, *epoch=<Time object: scale='tt' format='jyear_str' value=J2000.000>*, *rotate_meridian=False*)

Converts position and velocity in ICRS to body-centered frame.

Parameters

- **r** (*Quantity*) – Position vector in ICRS.
- **v** (*Quantity*) – Velocity vector in ICRS.
- **target_body** (Body) – Target body.
- **epoch** (*Time, optional*) – Epoch, default to J2000.
- **rotate_meridian** (*bool, optional*) – Whether to apply the rotation of the meridian too, default to False.

Returns **r, v** – Position and velocity vectors in a body-centered reference frame.

Return type tuple (Quantity)

poliastro.coordinates.**inertial_body_centered_to_pqw**(*r*, *v*, *source_body*)

Converts position and velocity from inertial body-centered frame to perifocal frame.

Parameters

- **r** (*Quantity*) – Position vector in a inertial body-centered reference frame.
- **v** (*Quantity*) – Velocity vector in a inertial body-centered reference frame.
- **source_body** (Body) – Source body.

Returns **r_pqw, v_pqw** – Position and velocity vectors in ICRS.

Return type tuple (Quantity)

poliastro.coordinates.**transform**(*orbit*, *frame_orig*, *frame_dest*)

Transforms Orbit from one frame to another.

Parameters

- **orbit** (*Orbit*) – Orbit to transform
- **frame_orig** (*BaseCoordinateFrame*) – Initial frame
- **frame_dest** (*BaseCoordinateFrame*) – Final frame

Returns **orbit** – Orbit in the new frame

Return type Orbit

poliastro.cli module

Command line functions.

poliastro.examples module

Example data.

poliastro.examples.**iss = 6772 x 6790 km x 51.6 deg (GCRS) orbit around Earth () at epoch 2(**
ISS orbit example

Taken from Plyades (c) 2012 Helge Eichhorn (MIT License)

poliastro.examples.**molniya = 6650 x 46550 km x 63.4 deg (GCRS) orbit around Earth () at epo**
Molniya orbit example

poliastro.examples.**soyuz_gto = 6628 x 42328 km x 6.0 deg (GCRS) orbit around Earth () at ep**
Soyuz geostationary transfer orbit (GTO) example

Taken from Soyuz User's Manual, issue 2 revision 0

poliastro.examples.**churi = 1 x 6 AU x 7.0 deg (HCRS) orbit around Sun () at epoch 2015-11-0**
Comet 67P/Churyumov–Gerasimenko orbit example

poliastro.frames module

Coordinate frames definitions.

class poliastro.frames.**Planes**
An enumeration.

class poliastro.frames.**HeliocentricEclipticJ2000**(**args*, *copy=True*, *representation_type=None*, *differential_type=None*, ***kwargs*)
Heliocentric ecliptic coordinates. These origin of the coordinates are the center of the sun, with the x axis pointing in the direction of the mean equinox of J2000 and the xy-plane in the plane of the ecliptic of J2000 (according to the IAU 1976/1980 obliquity model).

class poliastro.frames.**HCRS**(**args*, *copy=True*, *representation_type=None*, *differential_type=None*, ***kwargs*)

class poliastro.frames.**MercuryICRS**(**args*, *copy=True*, *representation_type=None*, *differential_type=None*, ***kwargs*)

class poliastro.frames.**VenusICRS**(**args*, *copy=True*, *representation_type=None*, *differential_type=None*, ***kwargs*)

class poliastro.frames.**MarsICRS**(**args*, *copy=True*, *representation_type=None*, *differential_type=None*, ***kwargs*)

class poliastro.frames.**JupiterICRS**(**args*, *copy=True*, *representation_type=None*, *differential_type=None*, ***kwargs*)

class poliastro.frames.**SaturnICRS**(**args*, *copy=True*, *representation_type=None*, *differential_type=None*, ***kwargs*)

class poliastro.frames.**UranusICRS**(**args*, *copy=True*, *representation_type=None*, *differential_type=None*, ***kwargs*)

class poliastro.frames.**NeptuneICRS**(**args*, *copy=True*, *representation_type=None*, *differential_type=None*, ***kwargs*)

class poliastro.frames.**PlutoICRS**(**args*, *copy=True*, *representation_type=None*, *differential_type=None*, ***kwargs*)

poliastro.frames.**get_frame**(*attractor*, *plane*, *obstime=<Time object: scale='tt' format='jyear_str' value=J2000.000>*)
Returns an appropriate reference frame from an attractor and a plane.

Available planes are Earth equator (parallel to GCRS) and Earth ecliptic. The fundamental direction of both is the equinox of epoch (J2000). An obstime is needed to properly locate the attractor.

Parameters

- **attractor** (Body) – Body that serves as the center of the frame.
- **plane** (Planes) – Fundamental plane of the frame.
- **obstime** (*Time*) – Time of the frame.

poliastro.maneuver module

Orbital maneuvers.

class poliastro.maneuver.**Maneuver**(**impulses*)

Class to represent a Maneuver.

Each Maneuver consists on a list of impulses \(\Delta v_i\) (changes in velocity) each one applied at a certain instant \(t_i\). You can access them directly indexing the Maneuver object itself.

```
>>> man = Maneuver((0 * u.s, [1, 0, 0] * u.km / u.s),
... (10 * u.s, [1, 0, 0] * u.km / u.s))
>>> man[0]
(<Quantity 0. s>, <Quantity [1., 0., 0.] km / s>)
>>> man.impulses[1]
(<Quantity 10. s>, <Quantity [1., 0., 0.] km / s>)
```

__init__(**impulses*)

Constructor.

Parameters **impulses** (*list*) – List of pairs (delta_time, delta_velocity)

Notes

TODO: Fix docstring, *args convention

classmethod impulse(*dv*)

Single impulse at current time.

classmethod hohmann(*orbit_i*, *r_f*)

Compute a Hohmann transfer between two circular orbits.

classmethod bielliptic(*orbit_i*, *r_b*, *r_f*)

Compute a bielliptic transfer between two circular orbits.

get_total_time()

Returns total time of the maneuver.

get_total_cost()

Returns total cost of the maneuver.

poliastro.threebody package

poliastro.threebody.restricted module

Circular Restricted 3-Body Problem (CR3BP)

Includes the computation of Lagrange points

poliastro.threebody.restricted.**lagrange_points**(*r12*, *m1*, *m2*)
Computes the Lagrangian points of CR3BP.

Computes the Lagrangian points of CR3BP given the distance between two bodies and their masses. It uses the formulation found in Eq. (2.204) of Curtis, Howard. 'Orbital mechanics for engineering students'. Elsevier, 3rd Edition.

Parameters

- **r12** (*Quantity*) – Distance between the two bodies
- **m1** (*Quantity*) – Mass of the main body
- **m2** (*Quantity*) – Mass of the secondary body

Returns Distance of the Lagrangian points to the main body, projected on the axis main body - secondary body

Return type Quantity

poliastro.threebody.restricted.**lagrange_points_vec**(*m1*, *r1*, *m2*, *r2*, *n*)
Computes the five Lagrange points in the CR3BP.

Returns the positions in the same frame of reference as *r1* and *r2* for the five Lagrangian points.

Parameters

- **m1** (*Quantity*) – Mass of the main body. This body is the one with the biggest mass.
- **r1** (*Quantity*) – Position of the main body.
- **m2** (*Quantity*) – Mass of the secondary body.
- **r2** (*Quantity*) – Position of the secondary body.
- **n** (*Quantity*) – Normal vector to the orbital plane.

Returns Position of the Lagrange points: [L1, L2, L3, L4, L5] The positions are of type ~astropy.units.Quantity

Return type list

poliastro.threebody.flybys module

poliastro.threebody.flybys.**compute_flyby**(*v_spacecraft*, *v_body*, *k*, *r_p*, *theta=<Quantity 0. deg>*)
Computes outbound velocity after a flyby.

Parameters

- **v_spacecraft** (*Quantity*) – Velocity of the spacecraft, relative to the attractor of the body.
- **v_body** (*Quantity*) – Velocity of the body, relative to its attractor.
- **k** (*Quantity*) – Standard gravitational parameter of the body.
- **r_p** (*Quantity*) – Radius of periapsis, measured from the center of the body.
- **theta** (*Quantity, optional*) – Aim angle of the B vector, default to 0.

Returns

- **v_spacecraft_out** (*~astropy.units.Quantity*) – Outbound velocity of the spacecraft.
- **delta** (*~astropy.units.Quantity*) – Turn angle.

poliastro.patched_conics module

Patched Conics computations

Contains methods to compute interplanetary trajectories approximating the three body problem with Patched Conics.

poliastro.patched_conics.**compute_soi**(*body*, *a=None*)
Approximated radius of the Laplace Sphere of Influence (SOI) for a body.

Parameters

- **body** (*~poliastro.bodies.Body*) – Astronomical body which the SOI's radius is computed for.
- **a** (*float, optional*) – Semimajor axis of the body's orbit, default to None (will be computed from ephemerides).

Returns Approximated radius of the Sphere of Influence (SOI) [m]

Return type astropy.units.quantity.Quantity

poliastro.plotting module

Plotting utilities.

poliastro.plotting.**plot**(*state*, *label=None*, *color=None*, *dark=False*)
Plots an *Orbit* in 2D.

For more advanced tuning, use the *OrbitPlotter* class.

poliastro.plotting.**plot3d**(*orbit*, ***, *label=None*, *color=None*, *dark=False*)
Plots an *Orbit* in 3D.

For more advanced tuning, use the *OrbitPlotter3D* class.

class poliastro.plotting.**OrbitPlotter**(*ax=None*, *num_points=150*, *dark=False*)
OrbitPlotter class.

This class holds the perifocal plane of the first *Orbit* plotted in it using *plot()*, so all following plots will be projected on that plane. Alternatively, you can call *set_frame()* to set the frame before plotting.

__init__(*ax=None*, *num_points=150*, *dark=False*)
Constructor.

Parameters

- **ax** (*Axes*) – Axes in which to plot. If not given, new ones will be created.
- **num_points** (*int, optional*) – Number of points to use in plots, default to 150.
- **dark** (*bool, optional*) – If set as True, plots the orbit in Dark mode.

set_frame(*p_vec*, *q_vec*, *w_vec*)
Sets perifocal frame.

Raises ValueError – If the vectors are not a set of mutually orthogonal unit vectors.

plot_trajectory(*trajectory*, ***, *label=None*, *color=None*)
Plots a precomputed trajectory.

Parameters **trajectory** (*BaseRepresentation, BaseCoordinateFrame*) – Trajectory to plot.

set_attractor(*attractor*)
Sets plotting attractor.

Parameters **attractor** (Body) – Central body.

plot(*orbit*, *label=None*, *color=None*, *method=<function mean_motion>*)
Plots state and osculating orbit in their plane.

class poliastro.plotting.**OrbitPlotter3D**(*dark=False*)
OrbitPlotter3D class.

__init__(*dark=False*)
Initialize self. See help(type(self)) for accurate signature.

class poliastro.plotting.**OrbitPlotter2D**
OrbitPlotter2D class.

New in version 0.9.0.

__init__()
Initialize self. See help(type(self)) for accurate signature.

poliastro.plotting.**plot_solar_system**(*outer=True*, *epoch=None*)
Plots the whole solar system in one single call.

New in version 0.9.0.

Parameters

- **outer** (*bool, optional*) – Whether to print the outer Solar System, default to True.
- **epoch** (*Time, optional*) – Epoch value of the plot, default to J2000.

poliastro.util module

Function helpers.

poliastro.util.**circular_velocity**(*k*, *a*)
Compute circular velocity for a given body (k) and semimajor axis (a).

poliastro.util.**rotate**(*vector*, *angle*, *axis='z'*)
Rotates a vector around axis a right-handed positive angle.

This is just a convenience function around `astropy.coordinates.matrix_utilities.rotation_matrix()`.

Parameters

- **vector** (*Quantity*) – Dimension 3 vector.
- **angle** (*Quantity*) – Angle of rotation.
- **axis** (*str, optional*) – Either 'x', 'y' or 'z'.

Note: This performs a so-called *active* or *alibi* transformation: rotates the vector while the coordinate system remains unchanged. To do the opposite operation (*passive* or *alias* transformation) call the function as `rotate(vec, ax, -angle, unit)` or use the convenience function *transform()*, see[1].

---

[1] http://en.wikipedia.org/wiki/Rotation_matrix#Ambiguities

References

poliastro.util.**transform**(*vector*, *angle*, *axis='z'*)
Rotates a coordinate system around axis a positive right-handed angle.

Note: This is a convenience function, equivalent to `rotate(vec, -angle, axis, unit)`. Refer to the documentation of *`rotate()`* for further information.

---

poliastro.util.**norm**(*vec*)
Norm of a Quantity vector that respects units.

Parameters **vec** (*Quantity*) – Vector with units.

poliastro.util.**time_range**(*start*, ***, *periods=50*, *spacing=None*, *end=None*, *format=None*, *scale=None*)
Generates range of astronomical times.

New in version 0.8.0.

Parameters

- **periods** (*int, optional*) – Number of periods, default to 50.
- **spacing** (*Time or Quantity, optional*) – Spacing between periods, optional.
- **end** (*Time or equivalent, optional*) – End date.

Returns Array of time values.

Return type Time

2.6.2 Core API

poliastro.core.angles module

poliastro.core.elements module

poliastro.core.elements.**rv_pqw**
Returns r and v vectors in perifocal frame.

poliastro.core.elements.**coe2rv**
Converts from classical orbital elements to vectors.

Parameters

- **k** (*float*) – Standard gravitational parameter (km^3 / s^2).
- **p** (*float*) – Semi-latus rectum or parameter (km).
- **ecc** (*float*) – Eccentricity.
- **inc** (*float*) – Inclination (rad).
- **omega** (*float*) – Longitude of ascending node (rad).
- **argp** (*float*) – Argument of perigee (rad).
- **nu** (*float*) – True anomaly (rad).

poliastro.core.elements.**coe2mee**
Converts from classical orbital elements to modified equinoctial orbital elements.

The definition of the modified equinoctial orbital elements is taken from [Walker, 1985].

Parameters

- **k** (*float*) – Standard gravitational parameter (km^3 / s^2).
- **p** (*float*) – Semi-latus rectum or parameter (km).
- **ecc** (*float*) – Eccentricity.
- **inc** (*float*) – Inclination (rad).
- **omega** (*float*) – Longitude of ascending node (rad).
- **argp** (*float*) – Argument of perigee (rad).
- **nu** (*float*) – True anomaly (rad).

Note: The conversion equations are taken directly from the original paper.

---

poliastro.core.elements.**rv2coe**
Converts from vectors to classical orbital elements.

Parameters

- **k** (*float*) – Standard gravitational parameter (km^3 / s^2).
- **r** (*array*) – Position vector (km).
- **v** (*array*) – Velocity vector (km / s).
- **tol** (*float, optional*) – Tolerance for eccentricity and inclination checks, default to 1e-8.

poliastro.core.hyper module

Utility hypergeometric functions.

poliastro.core.hyper.**hyp2f1b**
Hypergeometric function 2F1(3, 1, 5/2, x), see [Battin].

poliastro.core.iod module

poliastro.core.perturbations module

poliastro.core.perturbations.**J2_perturbation**
Calculates J2_perturbation acceleration (km/s2)

New in version 0.9.0.

Parameters

- **t0** (*float*) – Current time (s)
- **state** (*numpy.ndarray*) – Six component state vector [x, y, z, vx, vy, vz] (km, km/s).
- **k** (*float*) – gravitational constant, (km^3/s^2)
- **J2** (*float*) – oblateness factor

- **R** (*float*) – attractor radius

Notes

The J2 accounts for the oblateness of the attractor. The formula is given in Howard Curtis, (12.30)

poliastro.core.perturbations.**J3_perturbation**
Calculates J3_perturbation acceleration (km/s2)

Parameters

- **t0** (*float*) – Current time (s)
- **state** (*numpy.ndarray*) – Six component state vector [x, y, z, vx, vy, vz] (km, km/s).
- **k** (*float*) – gravitational constant, (km^3/s^2)
- **J3** (*float*) – oblateness factor
- **R** (*float*) – attractor radius

Notes

The J3 accounts for the oblateness of the attractor. The formula is given in Howard Curtis, problem 12.8 This perturbation has not been fully validated, see https://github.com/poliastro/poliastro/pull/398

poliastro.core.perturbations.**atmospheric_drag**
Calculates atmospheric drag acceleration (km/s2)

New in version 0.9.0.

Parameters

- **t0** (*float*) – Current time (s)
- **state** (*numpy.ndarray*) – Six component state vector [x, y, z, vx, vy, vz] (km, km/s).
- **k** (*float*) – gravitational constant, (km^3/s^2)
- **C_D** (*float*) – dimensionless drag coefficient ()
- **A** (*float*) – frontal area of the spacecraft (km^2)
- **m** (*float*) – mass of the spacecraft (kg)
- **H0** (*float*) – atmospheric scale height, (km)
- **rho0** (*float*) – the exponent density pre-factor, (kg / m^3)

Notes

This function provides the acceleration due to atmospheric drag. We follow Howard Curtis, section 12.4 the atmospheric density model is rho(H) = rho0 x exp(-H / H0)

poliastro.core.perturbations.**shadow_function**

Determines whether the satellite is in attractor's shadow, uses algorithm 12.3 from Howard Curtis

Parameters

- **r_sat** (*numpy.ndarray*) – position of the satellite in the frame of attractor (km)
- **r_sun** (*numpy.ndarray*) – position of star in the frame of attractor (km)

- **R** (*float*) – radius of body (attractor) that creates shadow (km)

poliastro.core.perturbations.**third_body**(*t0*, *state*, *k*, *k_third*, *third_body*)
Calculates 3rd body acceleration (km/s2)

Parameters

- **t0** (*float*) – Current time (s)
- **state** (*numpy.ndarray*) – Six component state vector [x, y, z, vx, vy, vz] (km, km/s).
- **k** (*float*) – gravitational constant, (km^3/s^2)
- **third_body** (*a callable object returning the position of 3rd body*) – third body that causes the perturbation

poliastro.core.perturbations.**radiation_pressure**(*t0*, *state*, *k*, *R*, *C_R*, *A*, *m*, *Wdivc_s*, *star*)
Calculates radiation pressure acceleration (km/s2)

Parameters

- **t0** (*float*) – Current time (s)
- **state** (*numpy.ndarray*) – Six component state vector [x, y, z, vx, vy, vz] (km, km/s).
- **k** (*float*) – gravitational constant, (km^3/s^2)
- **R** (*float*) – radius of the attractor
- **C_R** (*float*) – dimensionless radiation pressure coefficient, 1 < C_R < 2 ()
- **A** (*float*) – effective spacecraft area (km^2)
- **m** (*float*) – mass of the spacecraft (kg)
- **Wdivc_s** (*float*) – total star emitted power divided by the speed of light (W * s / km)
- **star** (*a callable object returning the position of star in attractor frame*) – star position

Notes

This function provides the acceleration due to star light pressure. We follow Howard Curtis, section 12.9

poliastro.core.thrust package

poliastro.core.thrust.change_a_inc.**extra_quantities**
Extra quantities given by the Edelbaum (a, i) model.

poliastro.core.thrust.change_a_inc.**beta**
Compute yaw angle (β) as a function of time and the problem parameters.

poliastro.core.thrust.change_a_inc.**beta_0**
Compute initial yaw angle (β) as a function of the problem parameters.

poliastro.core.thrust.change_a_inc.**compute_parameters**
Compute parameters of the model.

poliastro.core.thrust.change_a_inc.**delta_V**
Compute required increment of velocity.

poliastro.core.thrust.change_argp.**delta_V**
Compute required increment of velocity.

poliastro.core.thrust.change_argp.**extra_quantities**
Extra quantities given by the model.

poliastro.core.thrust.change_ecc_quasioptimal.**delta_V**
Compute required increment of velocity.

poliastro.core.thrust.change_ecc_quasioptimal.**extra_quantities**
Extra quantities given by the model.

poliastro.core.thrust.change_inc_ecc.**delta_V**
Compute required increment of velocity.

poliastro.core.thrust.change_inc_ecc.**extra_quantities**
Extra quantities given by the model.

poliastro.core.propagation module

poliastro.core.propagation.**mean_motion**
Propagates orbit using mean motion

New in version 0.9.0.

Parameters

- **orbit** (Orbit) – the Orbit object to propagate.
- **tof** (*float*) – Time of flight (s).

Notes

This method takes initial $\vec{r}, \vec{v}$, calculates classical orbit parameters, increases mean anomaly and performs inverse transformation to get final $\vec{r}, \vec{v}$ The logic is based on formulae (4), (6) and (7) from http://dx.doi.org/10.1007/s10569-013-9476-9

poliastro.core.stumpff module

Stumpff functions.

poliastro.core.stumpff.**c2**
Second Stumpff function.

For positive arguments:

$$c_2(\psi) = \frac{1 - \cos\sqrt{\psi}}{\psi}$$

poliastro.core.stumpff.**c3**
Third Stumpff function.

For positive arguments:

$$c_3(\psi) = \frac{\sqrt{\psi} - \sin\sqrt{\psi}}{\sqrt{\psi^3}}$$

poliastro.core.util module

Function helpers.

poliastro.core.util.**circular_velocity**
Compute circular velocity for a given body (k) and semimajor axis (a).

poliastro.core.util.**rotate**
Rotates the coordinate system around axis x, y or z a CCW angle.

Parameters

- **vec** (*ndarray*) – Dimension 3 vector.
- **angle** (*float*) – Angle of rotation (rad).
- **axis** (*int*) – Axis to be rotated.

Notes

This performs a so-called active or alibi transformation: rotates the vector while the coordinate system remains unchanged. To do the opposite operation (passive or alias transformation) call the function as *rotate(vec, ax, -angle)* or use the convenience function *transform*, see *[1]_*.

References

poliastro.core.util.**transform**
Rotates a coordinate system around axis a positive right-handed angle.

Notes

This is a convenience function, equivalent to *rotate(vec, ax, -angle)*. Refer to the documentation of that function for further information.

poliastro.core.util.**norm**
Norm of a 3d vector.

poliastro.core.util.**cross**
Computes cross product between two vectors

2.7 What's new

2.7.1 poliastro 0.11.0 - 2018-09-21

This short cycle release brought some new features related to the three body problem, as well as important changes related to how reference frames are handled in poliastro.

Highlights

- **Support for Python 3.7** has been added to the library, now that all the depdendencies are easily available there. Currently supported versions of Python are 3.5, 3.6 and 3.7.

New features

- **Lagrange points**: The new experimental module *poliastro.threebody.restricted* contains functions to compute the Lagrange points in the circular restricted three body problem (CR3BP). It has been validated only approximately, so use it at your own risk.
- **Flybys**: New functions to compute the exit velocity and turn angle have been added to the new module *poliastro.threebody.flybys*. The B-plane aim point can be specified and the result will be returned in the correct reference frame. This feature was motivated by the Parker Solar Probe mission, and you can read an example on how to analyze parts of its trajectory using poliastro.
- **Reference frames**: We addded experimental support for reference frames in poliastro objects. So far, the *Orbit* objects were in some assumed reference frame that could not be controlled, leading to some confusion by people that wanted some specific coordinates. Now, **the reference frame is made out explicit**, and there is also the possibility to make a limited set of transformations. This framework will be further developed in the next release and transformations to arbitrary frames will be allowed. Check out the *poliastro.frames* module for more information.

Bugs fixed

- Issue #450: Angles function of safe API have wrong docstrings

Do you want to help with the remaining ones? Check the current list here! https://github.com/poliastro/poliastro/issues?q=is%3Aopen+is%3Aissue+label%3Abug

Backwards incompatible changes

- The `poliastro.twobody.Orbit.sample()` method returns one single object again that contains the positions and the corresponding times.

Contributors

This is the complete list of the people that contributed to this release, with a + sign indicating first contribution.

- Juan Luis Cano
- Nikita Astrakhantsev
- Shreyas Bapat
- Daniel Lubián+
- Wil Selwood+

2.7.2 poliastro 0.10.0 - 2018-07-21

This major release brings important changes from the code perspective (including a major change in the structure of the library), several performance improvements and a new infrastructure for running timing benchmarks, as well as some new features and bug fixes.

Highlights

- **Major change in the structure of poliastro codebase**: We separated the high level, units safe functions from the low level, fast ones, with the subsequent improvement in code quality. With this change we effectively communicate where "core" algorithms should go, make easier for future contributors to add numerical functions, and improved the overall quality of the library.
- **Upgrade to new SciPy ODE solvers**: We wrote our own version of Dormand-Prince 8(5,3) based on the new IVP framework in SciPy 1.0 to take advantage of event detection, dense output and other fancy features. In particular, the *`sample()`* method now uses dense output when available, therefore removing the need to propagate the orbit repeatedly.
- **New infrastructure for benchmarks**: We started publishing timing benchmarks results using Airspeed Velocity, a Python framework for writing, running, studying and publishing benchmarks. Besides, we bought a dedicated machine to run them with as much precision as we can. Please check them out and consider adding new benchmarks as well!
- **Several performance improvements**: Now that we are tracking performance, we dedicated some time during this release to fix some performance regressions that appeared in propagation, improving the behavior near parabolic orbits, and accelerating (even more!) the Izzo algorithm for the Lambert problem as well as some poliastro utilities.
- **New Continuous Integration infrastructure**: We started to use CircleCI for the Linux tests, the coverage measurements and the documentation builds. This service has faster machines and better support for workflows, which significantly reduced the build times and completely removed the timeouts that were affecting us in Travis CI.
- **Plotly backends now stable**: We fixed some outstanding issues with the 2D Plotly backend so now it's no longer experimental. We also started refactoring some parts of the plotting module and prepared the ground for the new interactive widgets that Plotly 3.0 brings.

New features

- **New continuous thrust/low thrust guidance laws**: We brought some continuous thrust guidance laws for orbital maneuvers that have analytical solution, such as orbit raising combined with inclination change, eccentricity change and so forth. This is based on the Master Thesis of Juan Luis Cano, "Study of analytical solutions for low-thrust trajectories", which provided complete validation for all of these laws and which can be found on GitHub.
- **More natural perturbations**: We finished adding the most common orbital perturbations, namely Solar radiation pressure and J3 perturbation. We could not reach agreement with the paper for the latter, so if you are considering using it please read the discussion in the original pull request and consider lending us a hand to validate it properly!
- **New dark mode for matplotlib plots**: We added a `dark` parameter to *`OrbitPlotter`* objects so the background is black. Handy for astronomical purposes!

Bugs fixed:

Besides some installation issues due to the evolution of dependencies, these code bugs were fixed:

- Issue #345: Bodies had incorrect aspect ratio in OrbitPlotter2D
- Issue #369: Orbit objects cannot be unpickled
- Issue #382: Orbit.from_body_ephem returns wrong orbit for the Moon
- Issue #385: Sun Incorrectly plotted in plot_solar_system

Backward incompatible changes

- Some functions have been moved to :py:mod‘:poliastro.core‘.

Contributors

This is the complete list of the people that contributed to this release, with a + sign indicating first contribution.

- Juan Luis Cano
- Nikita Astrakhantsev
- Shreyas Bapat
- jmerskine1+

2.7.3 poliastro 0.9.1 - 2018-05-11

This is a minor release that fixes one single issue:

- Issue #369: Orbit objects cannot be unpickled

Thanks to Joan Fort Alsina for reporting.

2.7.4 poliastro 0.9.0 - 2018-04-25

This major release received lots of improvements in the 2D plotting code and propagation functions, introduced the new perturbation framework and paved the way for the Python in Astronomy 2018 workshop and the Google Summer of Code 2018 program.

New features

- **New experimental 2D Plotly backend**: A new *OrbitPlotter2D* class was introduced that uses Plotly instead of matplotlib for the rendering. There are still some issues that should be resolved when we take advantage of the latest Plotly version, hence the “experimental” nature.
- **New propagators**: A new Keplerian propagator `mean_motion()` was introduced that has better convergence properties than `kepler()`, so now the user can choose.
- **New perturbation functions**: A new module `poliastro.twobody.perturbations` was introduced that contains perturbation accelerations that can be readily used with *cowell()*. So far we implemented J2 and atmospheric drag effects, and we will add more during the summer. Check out the User Guide for examples!
- **Support for different propagators in sampling**: With the introduction of new propagators and perturbation accelerations, now the user can easily sample over a period of time using any of them. We are eager to see what experiments you come up with!
- **Easy plotting of the Solar System**: A new function *plot_solar_system()* was added to easily visualize our inner or complete Solar System in 2D plots.

Other highlights

- **poliastro participates in Google Summer of Code thanks to OpenAstronomy!** More information in the poliastro blog.

- **poliastro will be presented at the Python in Astronomy 2018 workshop** to be held at Center for Computational Astrophysics at the Flatiron Institute in New York, USA. You can read more details about the event here.

New contributors

This is the complete list of the people that contributed to this release, with a + sign indicating first contribution.

- Juan Luis Cano
- Pablo Galindo+
- Matt Ettus+
- Shreyas Bapat+
- Ritiek Malhotra+
- Nikita Astrakhantsev+

Bugs fixed:

- Issue #294: Default steps 2D plots were too visible

Backward incompatible changes

- Now the `poliastro.twobody.Orbit.sample()` method returns a tuple of (times, positions).
- All the propagator methods changed their signature and now accept `Orbit` objects.

2.7.5 poliastro 0.8.0 - 2017-11-18

This is a new major release, focused on bringing 3D plotting functions and preparing the material for the Open Source Cubesat Workshop.

New features

- **Sampling method** for `Orbit` objects that returns an array of positions. This was already done in the plotting functions and will help providing other applications, such as exporting an Orbit to other formats.
- **3D plotting functions**: finally poliastro features a new high level object, *`poliastro.plotting.OrbitPlotter3D`*, that uses Plotly to represent orbit and trajectories in 3D. The venerable notebook about the trajectory of rover Curiosity has been updated accordingly.
- **Propagation to a certain date**: now apart from specifying the total elapsed time for propagation or time of flight, we can directly specify a target date in *`poliastro.twobody.orbit.Orbit.propagate()`*.
- **Hyperbolic anomaly conversion**: we implemented the conversion of hyperbolic to mean and true anomaly to complement the existing eccentric anomaly functions and improve the handling of hyperbolic orbits in *`poliastro.twobody.angles`*.

Other highlights

- **poliastro is now an Astropy affiliated package**, which gives the project a privileged position in the Python ecosystem. Thank you, Astropy core developers! You can read the evaluation here.
- **poliastro will be presented at the first Open Source Cubesat Workshop** to be held at the European Space Operations Centre in Darmstadt, Germany. You can read the full program of the event here.

New contributors

This is the complete list of the people that contributed to this release, with a + sign indicating first contribution.

- Juan Luis Cano
- Antonio Hidalgo
- mattrossman+
- Roshan Jossey+

Bugs fixed:

- Issue #275: Converting from true to mean anomaly fails for hyperbolic orbits

Backward incompatible changes

- The `ephem` module has been removed in favor of the `astropy.coordinates.get_body_barycentric_posvel` function.

2.7.6 poliastro 0.7.0 - 2017-09-15

This is a new major release, which adds new packages and modules, besides fixing several issues.

New features:

- **NEOS package**: a new package has been added to poliastro, *`neos`* package. It provides several ways of getting NEOs (Near Earth Objects) data from NASA databases, online and offline.
- **New patched conics module**. New module containing a function to compute the radius of the Sphere of Influence (SOI).
- **Use Astropy for body ephemerides**. Instead of downloading the SPK files ourselves, now we use Astropy builtin capabilities. This also allows the user to select a builtin ephemerides that does not require external downloads. See #131 for details.
- **Coordinates and frames modules**: new modules containing transformations between ICRS and body-centered frame, and perifocal to body_centered, *`coordinates`* as well as Heliocentric coordinate frame in *`frames`* based on Astropy for NEOs.
- **Pip packaging**: troublesome dependencies have been released in wheel format, so poliastro can now be installed using pip from all platforms.
- **Legend plotting**: now label and epoch are in a figure legend, which ends with the ambiguity of the epochs when having several plots in the same figure.

Other highlights:

- **Joined Open Astronomy**: we are now part of Open Astronomy, a collaboration between open source astronomy and astrophysics projects to share resources, ideas, and to improve code.
- **New constants module**: poliastro has now a `constants` module, with GMs and radii of solar system bodies.
- **Added Jupyter examples**: poliastro examples are now available in the documentation as Jupyter notebooks, thanks to nbsphinx.
- **New Code of Conduct**: poliastro community now has a Code of conduct.
- **Documentation update**: documentation has been updated with new installation ways, propagation and NEOs examples, "refactored" code and images, improved contribution guidelines and intersphinx extension.
- **New success stories**: two new success stories have been added to documentation.
- **Bodies now have a parent**. It is now possible to specify the attractor of a body.
- **Relative definition of Bodies**. Now it is possible to define Body parameters with respect to another body, and also add any number of properties in a simple way.

New contributors

Thanks to the generous SOCIS grant from the European Space Agency, Antonio Hidalgo has devoted three months developing poliastro full time and gained write acces to the repository.

This is the complete list of the people that contributed to this release, with a + sign indicating first contribution.

- Juan Luis Cano
- MiguelHB+
- Antonio Hidalgo+
- Zac Miller+
- Fran Navarro+
- Pablo Rodríguez Robles+

Bugs fixed:

- Issue #205: Bug when plotting orbits with different epochs.
- Issue #128: Missing ephemerides if no files on import time.
- Issue #131: Slightly incorrect ephemerides results due to improper time scale.
- Issue #130: Wrong attractor size when plotting different orbits.

Backward incompatible changes:

- **Non-osculating orbits**: removed support for non-osculating orbits. `plotting.plot()` calls containing `osculating` parameter should be replaced.

2.7.7 poliastro 0.6.0 - 2017-02-12

This major release was focused on refactoring some internal core parts and improving the propagation functionality.

Highlights:

- **Support Python 3.6**. See #144.
- **Introduced ‘‘Orbit‘‘ objects** to replace `State` ones. The latter has been simplified, reducing some functionality, now their API has been moved to the former. See the User Guide and the examples for updated explanations. See #135.
- **Allow propagation functions to receive a callback**. This paves the way for better plotting and storage of results. See #140.

2.7.8 poliastro 0.5.0 - 2016-03-06

This is a new major release, focused on expanding the initial orbit determination capabilities and solving some infrastructure challenges.

New features:

- **Izzo's algorithm for the Lambert problem**: Thanks to this algorithm multirevolution solutions are also returned. The old algorithm is kept on a separate module.

Other highlights:

- **Documentation on Read the Docs**: You can now browse previous releases of the package and easily switch between released and development versions.
- **Mailing list**: poliastro now has a mailing list hosted on groups.io. Come and join!
- **Clarified scope**: poliastro will now be focused on interplanetary applications, leaving other features to the new python-astrodynamics project.

Bugs fixed:

- Issue #110: Bug when plotting State with non canonical units

Backward incompatible changes:

- **Drop Legacy Python**: poliastro 0.5.x and later will support only Python 3.x. We recommend our potential users to create dedicated virtual environments using conda or virtualenv or to contact the developers to fund Python 2 support.
- **Change ‘‘lambert‘‘ function API**: The functions for solving Lambert's problem are now _generators_, even in the single revolution case. Check out the User Guide for specific examples.
- **Creation of orbits from classical elements**: poliastro has reverted the switch to the *semilatus rectum* \(p\) instead of the semimajor axis \(a\) made in 0.4.0, so \(a\) must be used again. This change is definitive.

2.7.9 poliastro 0.4.2 - 2015-12-24

Fixed packaging problems.

2.7.10 poliastro 0.4.0 - 2015-12-13

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.
- Removed specific angular momentum \(h\) property to avoid a name clash with the fourth modified equinoctial element, use `norm(ss.h_vec)` instead.

2.7.11 poliastro 0.3.1 - 2015-06-30

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 `ephem`.
- Issue #50: Fixed `ZeroDivisionError` when propagating with time zero.

2.7.12 poliastro 0.3.0 - 2015-05-09

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 *iod* to determine an orbit given two position vectors and the time of flight.
- PR #42: **Planetary ephemerides computation**: New module `ephem` with functions to deal with SPK files and compute position and velocity vectors of the planets.
- PR #38: New method `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.

2.7.13 poliastro 0.2.1 - 2015-04-26

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.

2.7.14 poliastro 0.2 - 2014-08-16

- **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 **physical units** through `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.

2.8 Example Gallery - poliastro

Checkout the example gallery for a better insight of the poliastro package.

Note: Click *here* to download the full example code

2.8.1 Analyzing the Parker Solar Probe flybys

First, using the data available in the reports, we try to compute some of the properties of orbit #2. This is not enough to completely define the trajectory, but will give us information later on in the process.

```
from astropy import units as u

T_ref = 150 * u.day
print(T_ref)
```

Out:

```
150.0 d
```

```
from poliastro.bodies import Earth, Sun, Venus

k = Sun.k
print(k)
```

Out:

```
Name   = Heliocentric gravitational constant
  Value  = 1.32712442099e+20
  Uncertainty  = 10000000000.0
  Unit  = m3 / s2
  Reference = IAU 2009 system of astronomical constants
```

```
import numpy as np
```

$$T = 2\pi\sqrt{\frac{a^3}{\mu}} \Rightarrow a = \sqrt[3]{\frac{\mu T^2}{4\pi^2}}$$

```
a_ref = np.cbrt(k * T_ref**2 / (4 * np.pi**2)).to(u.km)
print(a_ref.to(u.au))
```

Out:

```
0.5524952607456924 AU
```

$$\varepsilon = -\frac{\mu}{r} + \frac{v^2}{2} = -\frac{\mu}{2a} \Rightarrow v = +\sqrt{\frac{2\mu}{r} - \frac{\mu}{a}}$$

```
energy_ref = (-k / (2 * a_ref)).to(u.J / u.kg)
print(energy_ref)
```

Out:

```
-802837548.527052 J / kg
```

```
from poliastro.twobody import Orbit
from poliastro.util import norm
from astropy.time import Time

flyby_1_time = Time("2018-09-28", scale="tdb")
print(flyby_1_time)
```

Out:

```
2018-09-28 00:00:00.000
```

```
r_mag_ref = norm(Orbit.from_body_ephem(Venus, epoch=flyby_1_time).r)
print(r_mag_ref.to(u.au))
```

Out:

```
0.7257313162319988 AU
```

```
v_mag_ref = np.sqrt(2 * k / r_mag_ref - k / a_ref)
print(v_mag_ref.to(u.km / u.s))
```

Out:

```
28.967363509562784 km / s
```

2. Lambert arc between #0 and #1

To compute the arrival velocity to Venus at flyby #1, we have the necessary data to solve the boundary value problem.

```
d_launch = Time("2018-08-11", scale="tdb")
print(d_launch)
```

Out:

```
2018-08-11 00:00:00.000
```

```
ss0 = Orbit.from_body_ephem(Earth, d_launch)
ss1 = Orbit.from_body_ephem(Venus, epoch=flyby_1_time)

tof = flyby_1_time - d_launch

from poliastro import iod

(v0, v1_pre), = iod.lambert(Sun.k, ss0.r, ss1.r, tof.to(u.s))
print(v0)
```

Out:

```
[ 9.59933726 11.29855172  2.92449333] km / s
```

```
print(v1_pre)
```

Out:

```
[-16.98082099  23.30752839   9.13129077] km / s
```

```
print(norm(v1_pre))
```

Out:

```
30.24846495185759 km / s
```

3. Flyby #1 around Venus

We compute a flyby using poliastro with the default value of the entry angle, just to discover that the results do not match what we expected.

```
from poliastro.threebody.flybys import compute_flyby

V = Orbit.from_body_ephem(Venus, epoch=flyby_1_time).v
print(V)
```

Out:

```
[ 648499.73735241 2695078.44750227 1171563.7170508 ] km / d
```

```
h = 2548 * u.km

d_flyby_1 = Venus.R + h
print(d_flyby_1.to(u.km))
```

Out:

```
8599.8 km
```

```
V_2_v_, delta_ = compute_flyby(v1_pre, V, Venus.k, d_flyby_1)

print(norm(V_2_v_))
```

Out:

```
27.75533877003213 km / s
```

4. Optimization

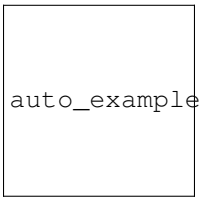
Now we will try to find the value of $theta$ that satisfies our requirements.

```
def func(theta):
    V_2_v, _ = compute_flyby(v1_pre, V, Venus.k, d_flyby_1, theta * u.rad)
    ss_1 = Orbit.from_vectors(Sun, ss1.r, V_2_v, epoch=flyby_1_time)
    return (ss_1.period - T_ref).to(u.day).value
```

There are two solutions:

```
import matplotlib.pyplot as plt
# %matplotlib inline

theta_range = np.linspace(0, 2 * np.pi)
plt.figure()
plt.plot(theta_range, [func(theta) for theta in theta_range])
plt.axhline(0, color='k', linestyle="dashed")
plt.show()
```



auto_examples/images/sphx_glr_plot_AnalyzingtheParkerSolarProbeflybys_001.png

```
print(func(0))
```

Out:

```
-9.142672330001131
```

```
print(func(1))
```

Out:

```
7.09811543934556
```

```
from scipy.optimize import brentq

theta_opt_a = brentq(func, 0, 1) * u.rad
print(theta_opt_a.to(u.deg))
```

Out:

```
38.59870925415651 deg
```

```
theta_opt_b = brentq(func, 4, 5) * u.rad
print(theta_opt_b.to(u.deg))
```

Out:

```
279.34770004025205 deg
```

```
V_2_v_a, delta_a = compute_flyby(v1_pre, V, Venus.k, d_flyby_1, theta_opt_a)
V_2_v_b, delta_b = compute_flyby(v1_pre, V, Venus.k, d_flyby_1, theta_opt_b)

print(norm(V_2_v_a))
```

Out:

```
28.96736350956281 km / s
```

```
print(norm(V_2_v_b))
```

Out:

```
28.96736350956282 km / s
```

5. Exit Orbit

And finally, we compute orbit #2 and check that the period is the expected one.

```
ss01 = Orbit.from_vectors(Sun, ss1.r, v1_pre, epoch=flyby_1_time)
print(ss01)
```

Out:

```
0 x 1 AU x 18.8 deg (HCRS) orbit around Sun () at epoch 2018-09-28 00:00:00.000 (TDB)
```

The two solutions have different inclinations, so we still have to find out which is the good one. We can do this by computing the inclination over the ecliptic - however, as the original data was in the International Celestial Reference Frame (ICRF), whose fundamental plane is parallel to the Earth equator of a reference epoch, we have change the plane to the Earth ecliptic, which is what the original reports use.

```
ss_1_a = Orbit.from_vectors(Sun, ss1.r, V_2_v_a, epoch=flyby_1_time)
print(ss_1_a)
```

Out:

```
0 x 1 AU x 25.0 deg (HCRS) orbit around Sun () at epoch 2018-09-28 00:00:00.000 (TDB)
```

```
ss_1_b = Orbit.from_vectors(Sun, ss1.r, V_2_v_b, epoch=flyby_1_time)
print(ss_1_b)
```

Out:

```
0 x 1 AU x 13.1 deg (HCRS) orbit around Sun () at epoch 2018-09-28 00:00:00.000 (TDB)
```

Let's define a function to do that quickly for us, using the get_frame function from poliastro.frames:

```
from astropy.coordinates import CartesianRepresentation
from poliastro.frames import Planes, get_frame

def change_plane(ss_orig, plane):
    """Changes the plane of the Orbit.

    """
    ss_orig_rv = ss_orig.frame.realize_frame(
        ss_orig.represent_as(CartesianRepresentation)
    )

    dest_frame = get_frame(ss_orig.attractor, plane, obstime=ss_orig.epoch)
```

(continues on next page)

(continued from previous page)

```
    ss_dest_rv = ss_orig_rv.transform_to(dest_frame)
    ss_dest_rv.representation_type = CartesianRepresentation

    ss_dest = Orbit.from_vectors(
        ss_orig.attractor,
        r=ss_dest_rv.data.xyz,
        v=ss_dest_rv.data.differentials['s'].d_xyz,
        epoch=ss_orig.epoch,
        plane=plane,
    )
    return ss_dest

print(change_plane(ss_1_a, Planes.EARTH_ECLIPTIC))
```

Out:

```
0 x 1 AU x 3.5 deg (HeliocentricEclipticJ2000) orbit around Sun () at epoch 2018-09-
↪28 00:00:00.000 (TDB)
```

```
print(change_plane(ss_1_b, Planes.EARTH_ECLIPTIC))
```

Out:

```
0 x 1 AU x 13.1 deg (HeliocentricEclipticJ2000) orbit around Sun () at epoch 2018-09-
↪28 00:00:00.000 (TDB)
```

Therefore, the correct option is the first one.

```
print(ss_1_a.period.to(u.day))
```

Out:

```
149.99999999999991 d
```

```
print(ss_1_a.a)
```

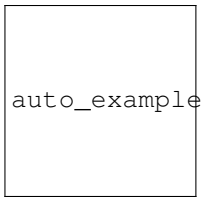
Out:

```
82652114.57939689 km
```

And, finally, we plot the solution:

```
from poliastro.plotting import OrbitPlotter
plt.figure()
frame = OrbitPlotter()

frame.plot(ss0, label=Earth)
frame.plot(ss1, label=Venus)
frame.plot(ss01, label="#0 to #1")
frame.plot(ss_1_a, label="#1 to #2");
```

- 
auto_examples/images/sphx_glr_plot_AnalyzingtheParkerSolarProbeflybys_002.png
- 
auto_examples/images/sphx_glr_plot_AnalyzingtheParkerSolarProbeflybys_003.png

Total running time of the script: (0 minutes 7.761 seconds)

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.

Python Module Index

p

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V