1 Success stories

2 Contents

2.1 About poliastro
2.2 Getting started
2.3 User guide
2.4 Jupyter notebooks
2.5 High level API
2.6 Core level API
2.7 What’s new
2.8 References

Python Module Index
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.

```python
from poliastro.examples import molniya
molniya.plot()
```
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)
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!

- **SpicePy**: 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
- Plotly, for interactive orbit plotting
- matplotlib, for static 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, 3.6 and 3.7 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[jupyter] pytest
```

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 three extensions of JupyterLab to make your experience smooth:
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.”

If you face any further issues, you can refer to the installation guide by Plotly.

### 2.2.3 Testing

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

```bash
$ python -c "import poliastro.testing; poliastro.testing.test()"
```

```
platform linux -- Python 3.7.1, pytest-4.2.0, py-1.7.0, pluggy-0.8.1
rootdir: /home/juanlu/.miniconda36/envs/_test37/lib/python3.7/site-packages/poliastro, __inifile: collected 747 items
[...] ======== 738 passed, 3 skipped, 5 xfailed, 1 xpased, 13 warnings in 392.12 seconds_
```  

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:

```python
from astropy import units as u
from poliastro.bodies import Earth, Mars, Sun
from poliastro.twobody import Orbit
```

**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

\[
\begin{align*}
  \mathbf{r} &= [-6045, -3490, 2500] \, \text{u.km} \\
  \mathbf{v} &= [-3.457, 6.618, 2.533] \, \text{u.km / u.s}
\end{align*}
\]

\[
\text{ss} = \text{Orbit.from_vectors}(\text{Earth}, \mathbf{r}, \mathbf{v})
\]

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 an \emph{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:

\[
\begin{align*}
  \text{ss} &= 7283 \times 10293 \text{ km} \times 153.2 \text{ deg (GCRS) orbit around Earth (})
\end{align*}
\]

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

\[
\begin{align*}
  \text{ss.epoch} &= \text{<Time object: scale='utc' format='jyear_str' value=J2000.000>} \\
  \text{ss.epoch.iso} &= '2000-01-01 12:00:00.000'
\end{align*}
\]

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

\[
\begin{align*}
  \text{ss.frame} &= \text{<GCRS Frame (obstime=J2000.000, obsgeoloc=(0., 0., 0.) m, obsgeovel=(0., 0., 0.) m / s)>}
\end{align*}
\]

\textbf{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.

\textbf{Intermezzo: quick visualization of the orbit}

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

\[
\text{ss.plot()}
\]

This plot is made in the so called \emph{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° in the direction of the orbit.

The dotted line represents the \emph{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.

**Note:** This visualization uses Plotly under the hood and works best in a Jupyter notebook. To use the old interface based on matplotlib, which might be more useful for batch jobs and publication-quality plots, check out the `poliastro.plotting.static.StaticOrbitPlotter`.

---

### From classical orbital elements

We can also define an `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()`:

```python
# 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 an `Orbit` from \((r)\) and \((v)\) or from elements we can access many mathematical properties of the orbit:

```python
>>> ss.period.to(u.day)
<Quantity 686.9713888628166 d>
```

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 it is 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:

```python
>>> 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:

```python
>>> 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:

```python
>>> 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)
```

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):

```python
>>> ((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:

```python
>>> 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)
```

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

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

```python
>>> ss_i = 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:

```python
>>> 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 \texttt{apply_maneuver()}:

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

\subsection{More advanced plotting: OrbitPlotter objects}

We previously saw the \texttt{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 \texttt{OrbitPlotter} objects in the \texttt{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:

```python
from poliastro.plotting import OrbitPlotter

op = OrbitPlotter2D()
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:
Fig. 1: Plot of a Hohmann transfer.
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`:

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

And finally, retrieve the planet orbit:

```python
>>> 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:

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

For instance, this is a simplified version of the example **Going to Mars with Python using poliastro**, where the orbit of the Mars Science Laboratory mission (rover Curiosity) is determined:

```python
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>

![Mars Science Laboratory orbit](image)

**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:

```python
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')
```

![Fig. 3: Apophis asteroid orbit compared to Earth orbit.](image)

2.3. User guide
2.4 Jupyter notebooks

2.4.1 Analyzing the Parker Solar Probe flybys

1. Modulus of the exit velocity, some features of Orbit #2

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.

```python
from astropy import units as u

T_ref = 150 * u.day
T_ref
```

```
150 d
```

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

k = Sun.k
k
```

```
1.3271244 × 10^20 m^3 s^{-2}
```

```python
import numpy as np

T = 2\pi \sqrt{\frac{a^3}{\mu}} \Rightarrow a = \sqrt{\frac{\mu T^2}{4\pi^2}}
```

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

```
0.55249526 AU
```

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

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

```
-8.0283755 × 10^8 \frac{1}{kg}
```

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

```python
from poliastro.threebody.flybys import compute_flyby
```

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.

```python
from poliastro.threebody.flybys import compute_flyby
```
[21]: V = Orbit.from_body_ephem(Venus, epoch=flyby_1_time).v

[21]: [648499.74, 2695078.4, 1171563.7] \text{ km}\\^2 \\

[22]: h = 2548 * u.km

[23]: d_flyby_1 = Venus.R + h
d_flyby_1.to(u.km)

[23]: 8599.8 \text{ km}

[24]: V_2_v_, delta_ = compute_flyby(v1_pre, V, Venus.k, d_flyby_1)

[25]: norm(V_2_v_)

[25]: 27.755339 \text{ km/s}

4. Optimization

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

[26]: 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:

[27]: import matplotlib.pyplot as plt

[28]: theta_range = np.linspace(0, 2 * np.pi)
plt.plot(theta_range, [func(theta) for theta in theta_range])
plt.axhline(0, color="k", linestyle="dashed");

[29]: func(0)
from scipy.optimize import brentq

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

38.598709 °

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

279.3477 °

5. Exit orbit

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

%norm(V_2_v_a)

28.967364 km/s

%norm(V_2_v_b)

28.967364 km/s

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)

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)

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)
    ss_dest_rv = ss_orig_rv.transform_to(dest_frame)
    ss_dest_rv.representation_type = CartesianRepresentation
    ss_dest = Orbit.from_coords(ss_orig.attractor, ss_dest_rv, plane=plane)
    return ss_dest

change_plane(ss_1_a, Planes.EARTH_ECLIPTIC)
0 x 1 AU x 3.5 deg (HeliocentricEclipticJ2000) orbit around Sun () at epoch 2018-09-28 00:00:00.000 (TDB)

change_plane(ss_1_b, Planes.EARTH_ECLIPTIC)
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.

ss_1_a.period.to(u.day)
150 d

ss_1_a.a
82652115 km

And, finally, we plot the solution:

from poliastro.plotting import OrbitPlotter2D
frame = OrbitPlotter2D()
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")
/home/juanlu/Development/poliastro/poliastro-library/src/poliastro/twobody/propagation.py:230: UserWarning: Frame <class 'astropy.coordinates.builtin_frames.icrs.ICRS'> does not support 'obstime', time values were not returned
2.4.2 Catch that asteroid!

```python
from astropy import units as u
from astropy.time import Time

from astropy.utils.data import conf
conf.dataurl
conf.remote_timeout
conf.remote_timeout = 10000

First, we need to increase the timeout time to allow the download of data occur properly
```

Then, we do the rest of the imports and create our initial orbits.

```python
from astropy.coordinates import solar_system_ephemeris
solar_system_ephemeris.set("jpl")

from poliastro.bodies import *
from poliastro.twobody import Orbit
from poliastro.plotting import OrbitPlotter2D
from poliastro.plotting.misc import plot_solar_system

EPOCH = Time("2017-09-01 12:05:50", scale="tdb")

earth = Orbit.from_body_ephem(Earth, EPOCH)
earth

earth.plot(label=Earth)

Frame <class 'astropy.coordinates.builtin_frames.icrs.ICRS'> does not support 'obstime', time values were not returned
```

```python
from poliastro.neos import neows
```

2.4. Jupyter notebooks
```python
florence = neows.orbit_from_name("Florence")
florence
```

1 x 3 AU x 22.1 deg (HeliocentricEclipticJ2000) orbit around Sun () at epoch 2458600.5 (TDB)

Two problems: the epoch is not the one we desire, and the inclination is with respect to the ecliptic!

```python
florence.rv()
```

```python
florence.epoch
```

```python
florence.epoch.iso
```

```python
'2019-04-27 00:00:00.000'
```

```python
florence.inc
```

```python
22.142394°
```

We first propagate:

```python
florence = florence.propagate(EPOCH)
florence.epoch.tdb.iso
```

```python
'2017-09-01 12:05:50.000'
```

And now we have to convert to the same frame that the planetary ephemerides are using to make consistent comparisons, which is ICRS:

```python
florence_icrs = florence.to_icrs()
florence_icrs.rv()
```

```python
(<Quantity [-1.46404761e+08, -5.35736589e+07, -2.05640225e+07] km>,
 <Quantity [7.33453685, 23.48471466, 24.12478169] km / s>)
```

Let us compute the distance between Florence and the Earth:

```python
from poliastro.util import norm
norm(florence_icrs.r - earth.r) - Earth.R
```

```python
6966586.7 km
```

This value is consistent with what ESA says! 7060 160 km

```python
abs(((norm(florence_icrs.r - earth.r) - Earth.R) - 7060160 * u.km) / (7060160 * u.km))
```

```python
0.013253715
```

```python
from IPython.display import HTML
HTML(
 """"""""""""""""
)""""""""""""""
```
And now we can plot!

```python
frame = plot_solar_system(outer=False, epoch=EPOCH)
frame.plot(florence_icrs, label="Florence")
```

The difference between doing it well and doing it wrong is clearly visible:

```python
frame = OrbitPlotter2D()
frame.plot(earth, label="Earth")
frame.plot(florence, label="Florence (Ecliptic)")
frame.plot(florence_icrs, label="Florence (ICRS)")
```

And now let’s do something more complicated: express our orbit with respect to the Earth! For that, we will use GCRS, with care of setting the correct observation time:

```python
from astropy.coordinates import GCRS, CartesianRepresentation
florence_heclip = florence.frame.realize_frame(florence.represent_as(CartesianRepresentation))
florence_gcrs_trans_cart = florence_heclip.transform_to(GCRS(obstime=EPOCH)).represent_as(CartesianRepresentation)
florence_hyper = Orbit.from_vectors(Earth, r=florence_gcrs_trans_cart.xyz, v=florence_gcrs_trans_cart.differentials['s'].d_xyz,
```
We now retrieve the ephemerides of the Moon, which are given directly in GCRS:

```python
[26]: moon = Orbit.from_body_ephem(Moon, EPOCH)
   moon
[26]: 367937 x 405209 km x 19.4 deg (GCRS) orbit around Earth () at epoch 2017-09-01 12:05 (TDB)
```

And now for the final plot:

```python
[28]: import matplotlib.pyplot as plt
   plt.ion()
   from poliastro.plotting.static import StaticOrbitPlotter
```

```python
[29]: frame = StaticOrbitPlotter()

   # This first plot sets the frame
   frame.plot(florence_hyper, label="Florence")

   # And then we add the Moon
   frame.plot(moon, label=Moon)

   plt.xlim(-1000000, 8000000)
   plt.ylim(-5000000, 5000000)
   plt.gcf().autofmt_xdate()
```
2.4.3 Comparing Hohmann and bielliptic transfers

```python
import numpy as np
import matplotlib.pyplot as plt
plt.ion()
from mpl_toolkits.axes_grid1.inset_locator import zoomed_inset_axes
from mpl_toolkits.axes_grid1.inset_locator import mark_inset
from astropy import units as u
from poliastro.bodies import Earth
from poliastro.twobody import Orbit
from poliastro.manuever import Maneuver

ZOOM = True
R = np.linspace(2, 75, num=100)
Rstar = [15.58, 40, 60, 100, 200, np.inf]

hohmann_data = np.zeros_like(R)
bielliptic_data = np.zeros((len(R), len(Rstar)))
ss_i = Orbit.circular(Earth, 1.8 * u.km)
r_i = ss_i.a
v_i = np.sqrt(ss_i.v.dot(ss_i.v))
for ii, r in enumerate(R):
    pass
```

Per Python ad astra!

2.4. Jupyter notebooks
r_f = r * r_i
man = Maneuver.hohmann(ss_i, r_f)
hohmann_data[ii] = (man.get_total_cost() / v_i).decompose().value

for jj, rstar in enumerate(Rstar):
    r_b = rstar * r_i
    man = Maneuver.bielliptic(ss_i, r_b, r_f)
    bielliptic_data[ii, jj] = (man.get_total_cost() / v_i).decompose().value

idx_max = np.argmax(hohmann_data)
ylims = (0.35, 0.6)

/home/juanlu/.miniconda36/envs/poliastro37/lib/python3.7/site-packages/astropy/units/quantity.py:461: RuntimeWarning:
invalid value encountered in true_divide

[3]: fig, ax = plt.subplots(figsize=(8, 6))

l, = ax.plot(R, hohmann_data, lw=2)
for jj in range(len(Rstar)):
    ax.plot(R, bielliptic_data[:, jj], color=l.get_color())
ax.vlines([11.94, R[idx_max]], *ylims, color='0.6')

if ZOOM:
    ax_zoom = zoomed_inset_axes(ax, 4, loc=4, axes_kwargs={'facecolor': '0.97'})
    for jj in range(len(Rstar)):
        ax_zoom.plot(R, bielliptic_data[:, jj], color=l.get_color())
    ax_zoom.vlines([11.94, R[idx_max]], *ylims, color='0.6')
    ax_zoom.set_xlim(11.0, 16.0)
    ax_zoom.set_ylim(0.52, 0.545)
    ax_zoom.set_xticks([])
    ax_zoom.set_yticks([])
    ax_zoom.grid(False)
    ax_zoom.set_title("4x zoom")
    mark_inset(ax, ax_zoom, loc1=1, loc2=3, fc="none", ec='0.3')

ax.set_xlabel("R")
ax.set_ylabel("Relative change in velocity")
ax.set_ylim(*ylims)
ax.set_xlim(2, 75)
ax.set_title("Hohmann vs bielliptic transfers")
fig.savefig("hohmann-bielliptic-transfers.png")
2.4.4 Customising static orbit plots

The default styling for plots works pretty well however sometimes you may need to change things. The following will show you how to change the style of your plots and have different types of lines and dots

This is the default plot we will start with:

[1]: from astropy.time import Time
    import matplotlib.pyplot as plt

    plt.ion()
    from poliastro.plotting.static import StaticOrbitPlotter

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

    [2]: epoch = Time("2018-08-17 12:05:50", scale="tdb")

    plotter = StaticOrbitPlotter()
    plotter.plot(Orbit.from_body_ephem(Earth, epoch), label="Earth")
    plotter.plot(Orbit.from_body_epphem(Mars, epoch), label="Mars")
    plotter.plot(Orbit.from_body_ephem(Jupiter, epoch), label="Jupiter");
```python
[3]: epoch = Time("2018-08-17 12:05:50", scale="tdb")

plotter = StaticOrbitPlotter()
earth_plots = plotter.plot(Orbit.from_body_ephem(Earth, epoch), label=Earth)

earth_plots[0].set_linestyle("-")  # solid line
earth_plots[0].set_linewidth(0.5)
earth_plots[1].set_marker("H")    # Hexagon
earth_plots[1].set_markersize(15)

mars_plots = plotter.plot(Orbit.from_body_ephem(Mars, epoch), label=Mars)
jupiter_plots = plotter.plot(Orbit.from_body_ephem(Jupiter, epoch), label=Jupiter)
```
Here we get hold of the lines list from the `OrbitPlotter.plot` method this is a list of lines. The first is the orbit line. The second is the current position marker. With the matplotlib lines objects we can start changing the style. First we make the line solid but thin line. Then we change the current position marker to a large hexagon.

More details of the style options for the markers can be found here: https://matplotlib.org/2.0.2/api/markers_api.html#module-matplotlib.markers More details of the style options on lines can be found here: https://matplotlib.org/2.0.2/api/lines_api.html However make sure that you use the set methods rather than just changing the attributes as the methods will force a re-draw of the plot.

Next we will make some changes to the other two orbits.

```python
[4]: epoch = Time("2018-08-17 12:05:50", scale="tdb")

plotter = StaticOrbitPlotter()

earth_plots = plotter.plot(Orbit.from_body_ephem(Earth, epoch), label=Earth)
earth_plots[0].set_linenstyle("-")  # solid line
earth_plots[0].set_linewidth(0.5)
earth_plots[1].set_marker("H")  # Hexagon
earth_plots[1].set_markersize(15)

mars_plots = plotter.plot(Orbit.from_body_ephem(Mars, epoch), label=Mars)
mars_plots[0].set_dashes([0, 1, 0, 1, 1, 0])
mars_plots[0].set_linewidth(2)
mars_plots[1].set_marker("D")  # Diamond
mars_plots[1].set_markersize(15)
mars_plots[1].set_fillstyle("none")
# make sure this is set if you use fillstyle 'none'
mars_plots[1].set_markeredgewidth(1)
```

(continues on next page)
You can also change the style of the plot using the matplotlib axis which can be acquired from the `OrbitPlotter()`

See the following example that creates a grid, adds a title, and makes the background transparent. To make the changes clearer it goes back to the initial example.

```python
[5]: epoch = Time("2018-08-17 12:05:50", scale="tdb")

fig, ax = plt.subplots()

ax.grid(True)
ax.set_title("Earth, Mars, and Jupiter")
ax.set_facecolor("None")

plotter = StaticOrbitPlotter(ax)

plotter.plot(Orbit.from_body_ephem(Earth, epoch), label=Earth)
plotter.plot(Orbit.from_body_ephem(Mars, epoch), label=Mars)
plotter.plot(Orbit.from_body_ephem(Jupiter, epoch), label=Jupiter)```
2.4.5 New Horizons launch and trajectory


[1]: from astropy import time
from astropy import units as u

from poliastro.bodies import Sun, Earth, Jupiter
from poliastro.twobody import Orbit
from poliastro.plotting import OrbitPlotter2D
from poliastro import iod
from poliastro.util import norm

Parking orbit

Quoting from “New Horizons Mission Design”:

It was first inserted into an elliptical Earth parking orbit of perigee altitude 165 km and apogee altitude 215 km. [Emphasis mine]

[2]: r_p = Earth.R + 165 * u.km
r_a = Earth.R + 215 * u.km

a_parking = (r_p + r_a) / 2
ecc_parking = 1 - r_p / a_parking

parking = Orbit.from_classical(  
    Earth,  
    a_parking,  
    ecc_parking,  
    0 * u.deg,  
    0 * u.deg,  
    0 * u.deg,

(continues on next page)
Hyperbolic exit

Hyperbolic excess velocity:

\[ v_\infty^2 = \frac{\mu}{-a} = 2\varepsilon = C_3 \]

Relation between orbital velocity \( v \), local escape velocity \( v_e \) and hyperbolic excess velocity \( v_\infty \):

\[ v^2 = v_e^2 + v_\infty^2 \]

**Option a): Insert \( C_3 \) from report, check \( v_e \) at parking perigee**

```python
[3]: C_3_A = 157.6561 * u.km ** 2 / u.s ** 2  # Designed
    a_exit = -(Earth.k / C_3_A).to(u.km)
    ecc_exit = 1 - r_p / a_exit
    exit = Orbit.from_classical(
        Earth,
        a_exit,
        ecc_exit,
        0 * u.deg,
        0 * u.deg,
        0 * u.deg,
        0 * u.deg,  # We don't mind
        time.Time("2006-01-19", scale="utc"),
    )
    norm(exit.v).to(u.km / u.s)

[3]: 16.718069 km s
```

Quoting “New Horizons Mission Design”:

After a short coast in the parking orbit, the spacecraft was then injected into the desired heliocentric orbit by the Centaur second stage and Star 48B third stage. At the Star 48B burnout, the New Horizons spacecraft reached the highest Earth departure speed, estimated at 16.2 km/s, becoming the fastest spacecraft ever launched from Earth. [Emphasis mine]
So it stays within the same order of magnitude. Which is reasonable, because real life burns are not instantaneous.

```python
fig = FigureWidget()
op = OrbitPlotter2D(figure=fig)

op.plot(parking)
op.plot(exit)

fig.layout.xaxis.range = -8000, 8000
fig.layout.yaxis.range = -20000, 20000

fig
```

**Option b): Compute $v_\infty$ using the Jupyter flyby**

According to Wikipedia, the closest approach occurred at 05:43:40 UTC. We can use this data to compute the solution of the Lambert problem between the Earth and Jupiter.

```python
nh_date = time.Time("2006-01-19 19:00", scale="utc")
nh_flyby_date = time.Time("2007-02-28 05:43:40", scale="utc")
nh_tof = nh_flyby_date - nh_date

nh_earth = Orbit.from_body_ephem(Earth, nh_date)
nh_r_0, v_earth = nh_earth.rv()

nh_jup = Orbit.from_body_ephem(Jupiter, nh_flyby_date)
nh_r_f, v_jup = nh_jup.rv()

(nh_v_0, nh_v_f), = iod.lambert(Sun.k, nh_r_0, nh_r_f, nh_tof)
```

(continues on next page)
Input time was converted to scale='tdb' with value 2007-02-28 05:44:45.185. Use → Time(..., scale='tdb') instead.

The hyperbolic excess velocity is measured with respect to the Earth:

\[ C_3_{\text{lambert}} = (\text{norm}(n_{h0} - v_{\text{earth}})).\text{to}(\text{u.km / u.s})^2 \]

\[ \text{Relative error of } 0.51\% \]

Which again, stays within the same order of magnitude of the figure given to the Guo & Farquhar report.

**From Earth to Jupiter**

\[ nh = \text{Orbit.from_vectors}(\text{Sun}, \text{n}_{h0}.\text{to}(\text{u.km}), \text{n}_{v0}.\text{to}(\text{u.km / u.s}), \text{n}_{\text{date}}) \]

\[ \text{op} = \text{OrbitPlotter2D()} \]

\[ \text{op.plot}(\text{n}_{\text{jup}}, \text{label}='\text{Jupiter}') \]

\[ \text{op.plot}(\text{n}_{\text{earth}}, \text{label}='\text{Earth}') \]

\[ \text{op.plot}(\text{nh}, \text{label}='\text{New Horizons}') \]

\[ /\text{home/juanlu/Development/poliastro/poliastro-library/src/poliastro/twobody/} \]

\[ \text{propagation.py:230: UserWarning: \text{Frame <class 'astropy.coordinates.builtin_frames.icrs.ICRS'} does not support \text{obstime}, time values were not returned} \]

\[ \text{FigureWidget({}} \]

\[ \quad \text{'}data': [\{} \text{'}hoverinfo': '\text{none}', \text{'}line': [\} \text{'}color': 'rgb(31, 119, 180)', \text{'}dash':... \]

### 2.4.6 Going to Mars with Python using poliastro

This is an example on how to use poliastro, a little library I’ve been working on to use in my Astrodynamics lessons. It features conversion between classical orbital elements and position vectors, propagation of Keplerian orbits, initial orbit determination using the solution of the Lambert’s problem and orbit plotting.

In this example we’re going to draw the trajectory of the mission Mars Science Laboratory (MSL), which carried the rover Curiosity to the surface of Mars in a period of something less than 9 months.

**Note:** This is a very simplistic analysis which doesn’t take into account many important factors of the mission, but can serve as an starting point for more serious computations (and as a side effect produces a beautiful plot at the end).

First of all, we import the necessary modules. Apart from poliastro we will make use of astropy to deal with physical units and time definitions and jplephem to compute the positions and velocities of the planets.
We need a binary file from NASA called *SPICE kernel* to compute the position and velocities of the planets. Astropy downloads it for us:

```python
from astropy.coordinates import solar_system_ephemeris, get_body_barycentric_posvel
solar_system_ephemeris.set("jpl")
```

The initial data was gathered from Wikipedia: the date of the launch was on **November 26, 2011 at 15:02 UTC** and landing was on **August 6, 2012 at 05:17 UTC**. We compute then the time of flight, which is exactly what it sounds. It is a crucial parameter of the mission.

```python
# Initial data
N = 50

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
tof.to(u.h)
```

Once we have the vector of times we can use `get_body_barycentric_posvel` to compute the array of positions and velocities of the Earth and Mars.

```python
times_vector = time_range(date_launch, end=date_arrival, periods=N)
times_vector[:5]
```

```python
<Time object: scale='utc' format='iso' value=['2011-11-26 15:02:00.000' '2011-12-01 19:14:33.082'
 '2011-12-06 23:27:06.163' '2011-12-12 03:39:39.245'
 '2011-12-17 07:52:12.327']>
```

```python
rr_earth, vv_earth = get_body_barycentric_posvel("earth", times_vector)
```

```python
rr_earth[:3]
```

```python
<CartesianRepresentation (x, y, z) in km
 [(64600643.37167563, 1.21424866e+08, 52640047.33041222),
 (52175250.21264037, 1.26254284e+08, 54733247.42732787),
 (39319701.40598051, 1.30036609e+08, 56373071.6065251 )]>
```

```python
vv_earth[:3]
```
The positions and velocities are in the International Celestial Reference Frame (ICRS), which has the Earth equator as the fundamental plane.

To compute the transfer orbit, we have the useful function `lambert`: according to a theorem with the same name, the transfer orbit between two points in space only depends on those two points and the time it takes to go from one to the other. We have the starting and final position and we have the time of flight: there we go!

```python
# Compute the transfer orbit!
r0 = rr_earth[0].xyz
rf = rr_mars[-1].xyz
(va, vb), = iod.lambert(Sun.k, r0, rf, tof)
ss0_trans = Orbit.from_vectors(Sun, r0, va, date_launch)
ssf_trans = Orbit.from_vectors(Sun, rf, vb, date_arrival)
```

```python
from poliastro.plotting import OrbitPlotter3D
from poliastro.bodies import Earth, Mars
from plotly.graph_objs import FigureWidget
```

```python
# I like color
color_earth0 = "#3d4cd5"
color_earthf = "#525fd5"
color_mars0 = "#ec3941"
color_marsf = "#ec1f28"
color_sun = "#ffcc00"
color_orbit = "#888888"
color_trans = "#444444"

fig = FigureWidget()
frame = OrbitPlotter3D(figure=fig)
```

(continues on next page)
```python
frame.set_attractor(Sun)
frame.plot_trajectory(rr_earth, label=Earth, color=color_earth0)
frame.plot_trajectory(rr_mars, label=Mars, color=color_marsf)
frame.plot_trajectory(
    propagate(ss0_trans, time.TimeDelta(times_vector - ss0_trans.epoch)),
    label="MSL trajectory",
    color=color_trans,
)
frame.set_view(30 * u.deg, 260 * u.deg, distance=3 * u.km)
fig.layout.title = "MSL Mission: from Earth to Mars"
fig
```

This line opens a new browser tab and saves the resulting image:

```python
#frame.savefig("msl3d.png", title="MSL Mission: from Earth to Mars")
```

Not bad! Let's celebrate with some music!

```python
from IPython.display import YouTubeVideo
YouTubeVideo('zSgiXGELjbc')
```
2.4.7 Going to Jupiter with Python using Jupyter and poliastro

```python
import numpy as np
import astropy.units as u
from astropy.time import Time
from astropy.coordinates import solar_system_ephemeris
from poliastro.bodies import Sun, Earth, Jupiter
from poliastro.twobody import Orbit
from poliastro.maneuver import Maneuver
from poliastro.iod import izzo
from poliastro.plotting import OrbitPlotter2D
from poliastro.util import norm

solar_system_ephemeris.set("jpl")
```

```python
# Initial data
# Links and sources: https://github.com/poliastro/poliastro/wiki/EuroPython:-Per-
→Python-ad-Astra

date_launch = Time("2011-08-05 16:25", scale="utc")
C_3 = 31.1 * u.km ** 2 / u.s ** 2
date_flyby = Time("2013-10-09 19:21", scale="utc")
date_arrival = Time("2016-07-05 03:18", scale="utc")
```

```python
# Initial state of the Earth
ss_e0 = Orbit.from_body_ephem(Earth, date_launch)
r_e0, v_e0 = ss_e0.rv()

→py:379: TimeScaleWarning:
Input time was converted to scale='tdb' with value 2011-08-05 16:26:06.183. Use
→Time(..., scale='tdb') instead.
```

```python
r_e0
```

```plaintext
1.0246553 \times 10^8, -1.023135 \times 10^8, -44353346] \text{ km}
```

```python
v_e0
```

```plaintext
[1847708.5, 1594323.4, 691089.12] \text{ km} / \text{ s}
```

```python
# State of the Earth the day of the flyby
ss_efly = Orbit.from_body_ephem(Earth, date_flyby)
r_efly, v_efly = ss_efly.rv()

→py:379: TimeScaleWarning:
Input time was converted to scale='tdb' with value 2013-10-09 19:22:07.182. Use
→Time(..., scale='tdb') instead.
```
# Assume that the insertion velocity is tangential to that of the Earth

dv = C_3 ** 0.5 * v_e0 / norm(v_e0)
man = Maneuver.impulse(dv)

# Inner Cruise 1
icl = ss_e0.apply_maneuver(man)
icl.rv()

\[
\text{(<Quantity [1.02465527e+08, -1.02313505e+08, -4.43533465e+07] km>,}
\text{<Quantity [2198705.82621214, 1897186.74383867, 822370.88977492] km / d>)}
\]

icl.period.to(u.year)

\[
2.1515474 \text{ yr}
\]

op = OrbitPlotter2D()
op.plot(ss_e0)
op.plot(icl)

Frame <class 'astropy.coordinates.builtin_frames.icrs.ICRS'> does not support 'obstime', time values were not returned

FigureWidget({
    'data': [{'hoverinfo': 'none',
       'line': {'color': 'rgb(31, 119, 180)', 'dash':...}

# We propagate until the aphelion
ss_aph = icl.propagate(icl.period / 2)
ss_aph.epoch

\[
<\text{Time object: scale='tdb' format='iso' value=2012-09-01 14:40}>
\]

# Let's compute the Lambert solution to do the flyby of the Earth
time_of_flight = date_flyby - ss_aph.epoch
time_of_flight

\[
<\text{TimeDelta object: scale='tai' format='jd' value=403.1958969055241}>
\]

(v_aph, v_fly), = izzo.lambert(Sun.k, ss_aph.r, ss_efly.r, time_of_flight)

# Check the delta-V
norm(v_aph - ss_aph.v) # Too high!

\[
1.079866 \text{ km/s}
\]

ss_aph_post = Orbit.from_vectors(Sun, ss_aph.r, v_aph, epoch=ss_aph.epoch)
ss_junofly = Orbit.from_vectors(Sun, r_efly, v_fly, epoch=date_flyby)

op = OrbitPlotter2D()
op.plot(ss_e0, label="Earth")
op.plot_trajectory(icl.sample(50, max_anomaly=180 * u.deg), label="Inner Cruise 1")
op.plot(ss_aph_post, label="Back to Earth")
Frame `<class 'astropy.coordinates.builtin_frames.icrs.ICRS'>` does not support `<code>`'obstime'`<code>`, time values were not returned

```python
FigureWidget({
    'data': [{'hoverinfo': 'none',
              'line': {'color': 'rgb(31, 119, 180)', 'dash':...
```

```python
# And now, go to Jupiter!
ss_j = Orbit.from_body_ephem(Jupiter, date_arrival)
r_, v_j = ss_j.rv()
/home/juanlu/Development/poliastro/poliastro-library/src/poliastro/twobody/orbit.py:379: TimeScaleWarning:
Input time was converted to scale='tdb' with value 2016-07-05 03:19:08.184. Use
```
```
```
```python
(v_flypre, v_oip), = izzo.lambert(Sun.k, r_efly, r_, date_arrival - date_flyby)
```
```
```python
ss_oip = Orbit.from_vectors(Sun, r_, v_oip, epoch=date_flyby)
```
```
```python
op = OrbitPlotter2D()
```
```
```python
# Plotting approximation, suggestions welcome
op.plot(ss_e0, label="Earth")
op.plot_trajectory(ic1.sample(50, max_anomaly=180 * u.deg), label="Inner Cruise 1")
op.plot_trajectory(
    ss_aph_post.sample(50, min_anomaly=180 * u.deg, max_anomaly=400 * u.deg),
    label="Back to Earth",
)
op.plot_trajectory(
    ss_oip.sample(50, min_anomaly=10 * u.deg, max_anomaly=180 * u.deg),
    label="Jupiter Orbit Insertion Phase",
)
op.plot(ss_j, label="Jupiter")
```
```
```python
FigureWidget({
    'data': [{'hoverinfo': 'none',
              'line': {'color': 'rgb(31, 119, 180)', 'dash':...
```

## 2.4.8 Plotting in 3D

```python
import numpy as np
from poliastro.examples import *
from poliastro.plotting import *
```
```
```python
churi.plot(use_3d=True)
```
WARNING: ErfaWarning: ERFA function "taiutc" yielded 3 of "dubious year (Note 4)"
→[astropy._erfa.core]

FigureWidget({
    'data': [{'line': {'color': 'rgb(31, 119, 180)', 'dash': 'dash', 'width': 5}, ...

[3]:
frame = OrbitPlotter3D()
frame.plot(churi)
frame.plot(Orbit.from_body_ephem(Earth))
/home/juanlu/Development/poliastro/poliastro-library/src/poliastro/twobody/
→propagation.py:230: UserWarning:
Frame <class 'astropy.coordinates.builtin_frames.icrs.ICRS'> does not support
→'obstime', time values were not returned

FigureWidget({
    'data': [{'line': {'color': 'rgb(31, 119, 180)', 'dash': 'dash', 'width': 5}, ...

[4]:
frame = OrbitPlotter3D()
frame.plot(molniya)
frame.plot(Orbit.from_body_ephem(Earth))
~/Development/poliastro/poliastro-library/src/poliastro/plotting/_base.py in
→plot(self, orbit, label, color)
  134     color = next(self._color_cycle)
  135
→ 136     self._set_attractor(orbit.attractor)
  137
  138     label = generate_label(orbit, label)
~/Development/poliastro/poliastro-library/src/poliastro/plotting/_base.py in
→_set_attractor(self, attractor)
  44     elif attractor is not self._attractor:
  45         raise NotImplementedError(
→ 46             "Attractor has already been set to {}.").format(self._attractor,
  47             --name)
  48
NotImplementedError: Attractor has already been set to Earth.

[5]:
frame = OrbitPlotter3D()
frame.plot(molniya)
frame.plot(iss)
from poliastro.neos import neows
from poliastro.examples import iss
eros = neows.orbit_from_name("eros")

frame = OrbitPlotter3D()
frame.plot(Orbit.from_body_ephem(Earth), label=Earth)
frame.plot(eros, label="eros")

Frame <class 'astropy.coordinates.builtin_frames.icrs.ICRS'> does not support 'obstime', time values were not returned

from astropy.coordinates import get_body_barycentric_posvel
from poliastro.util import time_range

date_launch = time.Time("2011-11-26 15:02", scale="utc")
date_arrival = time.Time("2012-08-06 05:17", scale="utc")
rr_earth, _ = get_body_barycentric_posvel("earth", time_range(date_launch, end=date_arrival, periods=50))

frame = OrbitPlotter3D()
frame.set_attractor(Sun)
frame.plot(Orbit.from_body_ephem(Earth), label=Earth)
frame.plot_trajectory(rr_earth, label=Earth)
frame.plot(eros, label="eros")
frame.plot_trajectory(rr_earth, label=Earth)

FigureWidget({
    'data': [{
        'line': {
            'color': 'rgb(31, 119, 180)',
            'dash': 'dash',
            'width': 5,
            ...}}]})
2.4.9 Cowell’s formulation

For cases where we only study the gravitational forces, solving the Kepler’s equation is enough to propagate the orbit forward in time. However, when we want to take perturbations that deviate from Keplerian forces into account, we need a more complex method to solve our initial value problem: one of them is **Cowell’s formulation**.

In this formulation we write the two body differential equation separating the Keplerian and the perturbation accelerations:

\[ \ddot{x} = -\frac{\mu}{|x|^3} + \mathcal{C}_d \]

For an in-depth exploration of this topic, still to be integrated in poliastro, check out https://github.com/Juanlu001/pfc-uc3m

An earlier version of this notebook allowed for more flexibility and interactivity, but was considerably more complex. Future versions of poliastro and plotly might bring back part of that functionality, depending on user feedback. You can still download the older version here.

**First example**

Let’s setup a very simple example with constant acceleration to visualize the effects on the orbit.

```python
[1]: import numpy as np
from astropy import units as u
from astropy import time
from poliastro.bodies import Earth
from poliastro.twobody import Orbit
from poliastro.twobody.propagation import propagate
from poliastro.examples import iss
from poliastro.twobody.propagation import cowell
from poliastro.plotting import OrbitPlotter3D
from poliastro.util import norm

To provide an acceleration depending on an extra parameter, we can use **closures** like this one:

```python
[2]: accel = 2e-5

```python
[3]: def constant_accel_factory(accel):
    def constant_accel(t0, u, k):
        v = u[3:]
        norm_v = (v[0]**2 + v[1]**2 + v[2]**2)**.5
        return accel * v / norm_v
    return constant_accel

```python
[4]: times = np.linspace(0, 10 * iss.period, 500)
times
[4]: [0, 111.36212, 222.72424, ..., 55346.973, 55458.335, 55569.697] s
```
And we plot the results:

Error checking

```python
[7]: def state_to_vector(ss):
    r, v = ss.rv()
    x, y, z = r.to(u.km).value
    vx, vy, vz = v.to(u.km / u.s).value
    return np.array([x, y, z, vx, vy, vz])
[8]: k = Earth.k.to(u.km ** 3 / u.s ** 2).value
[9]: rtol = 1e-13
full_periods = 2
[10]: u0 = state_to_vector(iss)
tf = ((2 * full_periods + 1) * iss.period / 2)
u0, tf
[10]: (array([ 8.59072560e+02, -4.13720368e+03, 5.29556871e+03, 7.37289205e+00, 2.08223573e+00, 4.39999794e-01]), <Quantity 13892.42425291 s>)
[11]: iss_f_kep = iss.propagate(tf, rtol=1e-18)
[12]: r, v = cowell(iss.attractor.k, iss.r, iss.v, [tf] * u.s, rtol=rtol)
[13]: iss_f_num = Orbit.from_vectors(Earth, r[0], v[0], iss.epoch + tf)
[14]: iss_f_num.r, iss_f_kep.r
[14]: (<Quantity [ -835.92108005, 4151.60692532, -5303.60427969] km>, <Quantity [ -835.92108005, 4151.60692532, -5303.60427969] km>)
[15]: assert np.allclose(iss_f_num.r, iss_f_kep.r, rtol=rtol, atol=1e-08 * u.km)
assert np.allclose(iss_f_num.v, iss_f_kep.v, rtol=rtol, atol=1e-08 * u.km / u.s)
```
Numerical validation

According to [Edelbaum, 1961], a coplanar, semimajor axis change with tangent thrust is defined by:

\[ \frac{da}{a_0} = 2 \frac{F}{mV_0} dt, \quad \frac{\Delta V}{V_0} = 1 \frac{\Delta a}{2 a_0} \]

So let’s create a new circular orbit and perform the necessary checks, assuming constant mass and thrust (i.e. constant acceleration):

```python
[17]: ss = Orbit.circular(Earth, 500 * u.km)
tof = 20 * ss.period
ad = constant_accel_factory(1e-7)
r, v = cowell(ss.attractor.k, ss.r, ss.v, [tof] * u.s, ad=ad)
ss_final = Orbit.from_vectors(Earth, r[0], v[0], ss.epoch + tof)
```

```python
[18]: da_a0 = (ss_final.a - ss.a) / ss.a
da_a0
[18]: 2.989621 × 10^-6 km
```

```python
[19]: dv_v0 = abs(norm(ss_final.v) - norm(ss.v)) / norm(ss.v)
2 * dv_v0
[19]: 0.0029960538
```

```python
[20]: np.allclose(da_a0, 2 * dv_v0, rtol=1e-2)
[20]: True
```

This means we successfully validated the model against an extremely simple orbit transfer with approximate analytical solution. Notice that the final eccentricity, as originally noticed by Edelbaum, is nonzero:

```python
[21]: ss_final.ecc
[21]: 6.6621427 × 10^-6
```

References

- [Edelbaum, 1961] “Propulsion requirements for controllable satellites”

2.4.10 Revisiting Lambert’s problem in Python
Part 1: Reproducing the original figure

```python
x = np.linspace(-1, 2, num=1000)
M_list = 0, 1, 2, 3
ll_list = 1, 0.9, 0.7, 0, -0.7, -0.9, -1

fig, ax = plt.subplots(figsize=(10, 8))
ax.set_prop_cycle(cycler('linestyle', ['-','--']) *
              (cycler('color', ['black']) * len(ll_list)))

for M in M_list:
    for ll in ll_list:
        T_x0 = np.zeros_like(x)
        for ii in range(len(x)):
            y = iod._compute_y(x[ii], ll)
            T_x0[ii] = iod._tof_equation(x[ii], y, 0.0, ll, M)
        if M == 0 and ll == 1:
            T_x0[x > 0] = np.nan
        elif M > 0:
            # Mask meaningless solutions
            T_x0[x > 1] = np.nan
        l, = ax.plot(x, T_x0)

ax.set_ylim(0, 10)
ax.set_xticks((-1, 0, 1, 2))
ax.set_yticks((0, np.pi, 2 * np.pi, 3 * np.pi))
ax.set_yticklabels(('$0$', '$\pi$', '$2 \pi$', '$3 \pi$'))
ax.vlines(1, 0, 10)
ax.text(0.65, 4.0, "elliptic")
ax.text(1.16, 4.0, "hyperbolic")

ax.text(0.05, 1.5, "$M = 0$", bbox=dict(facecolor='white'))
ax.text(0.05, 5, "$M = 1$", bbox=dict(facecolor='white'))
ax.text(0.05, 8, "$M = 2$", bbox=dict(facecolor='white'))
ax.annotate("$\lambda = 1$", xy=(-0.3, 1), xytext=(-0.75, 0.25),
            arrowprops=dict(arrowstyle="simple", facecolor="black"))
ax.annotate("$\lambda = -1$", xy=(0.3, 2.5), xytext=(0.65, 2.75),
            arrowprops=dict(arrowstyle="simple", facecolor="black"))
ax.grid()
ax.set_xlabel("$x$")
ax.set_ylabel("$T$")
```

Part 2: Locating $T_{\text{min}}$

```python
for M in M_list:
    for ll in ll_list:
        x_T_min, T_min = iod._compute_T_min(ll, M, 10, 1e-8)
        ax.plot(x_T_min, T_min, "kx", mew=2)
```

![Diagram showing the locaiton of $T_{\text{min}}$ with various curves for different values of $M$.](image-url)
Part 3: Try out solution

```python
[5]: T_ref = 1
ll_ref = 0
(x_ref, _), = iod._find_xy(ll_ref, T_ref, 0, 10, 1e-8)
x_ref
```
```python
0.4334673453504257
```

```python
[6]: ax.plot(x_ref, T_ref, "o", mew=2, mec="red", mfc="none")
```
```
fig
```
Part 4: Run some examples

```python
from astropy import units as u
from poliastro.bodies import Earth

Single revolution

```python
k = Earth.k
r0 = [15945.34, 0.0, 0.0] * u.km
r = [12214.83399, 10249.46731, 0.0] * u.km
tof = 76.0 * u.min

expected_va = [2.058925, 2.915956, 0.0] * u.km / u.s
expected_vb = [-3.451569, 0.910301, 0.0] * u.km / u.s

(v0, v), = izzo.lambert(k, r0, r, tof)
```
Multiple revolutions

\[
\begin{align*}
\text{k} &= \text{Earth.k} \\
\text{r0} &= [22592.145603, -1599.915239, -19783.950506] \times \text{u.km} \\
\text{r} &= [1922.067697, 4054.157051, -8925.727465] \times \text{u.km} \\
\text{tof} &= 10 \times \text{u.h} \\
\text{expected}_\text{va} &= [2.000652697, 0.387688615, -2.666947760] \times \text{u.km} / \text{u.s} \\
\text{expected}_\text{vb} &= [-3.79246619, -1.77707641, 6.856814395] \times \text{u.km} / \text{u.s} \\
\text{expected}_\text{va}_\text{l} &= [0.50335770, 0.61869408, -1.57176904] \times \text{u.km} / \text{u.s} \\
\text{expected}_\text{vb}_\text{l} &= [-4.18334626, -1.13262727, 6.13307091] \times \text{u.km} / \text{u.s} \\
\text{expected}_\text{va}_\text{r} &= [-2.45759553, 1.16945801, 0.43161258] \times \text{u.km} / \text{u.s} \\
\text{expected}_\text{vb}_\text{r} &= [-5.53841370, 0.01822220, 5.49641054] \times \text{u.km} / \text{u.s}
\end{align*}
\]

\[
\begin{align*}
\text{v0, v),} &= \text{izzo.lambert(k, r0, r, tof)} \\
\text{v}
\end{align*}
\]

\[
\begin{align*}
\text{v} &= [-3.7924662, -1.7770764, 6.8568144] \times \text{km} / \text{s}
\end{align*}
\]

\[
\begin{align*}
\text{v}_\text{l} &= [-4.1833463, -1.1326273, 6.1330709] \times \text{km} / \text{s} \\
\text{v}_\text{r} &= [-5.5384132, 0.018222134, 5.4964102] \times \text{km} / \text{s}
\end{align*}
\]

### 2.4.11 Studying Hohmann transfers

\[
\begin{align*}
\text{import numpy as np} \\
\text{import matplotlib.pyplot as plt} \\
\text{plt.ion()} \\
\text{plt.ion()}
\end{align*}
\]

(continues on next page)
from astropy import units as u
from poliastro.util import norm
from poliastro.bodies import Earth
from poliastro.twobody import Orbit
from poliastro.maneuver import Maneuver

Earth.k

3.9860044 × 10^14 m^3/s^2

ss_i = Orbit.circular(Earth, alt=800 * u.km)

7178 x 7178 km x 0.0 deg (GCRS) orbit around Earth () at epoch J2000.000 (TT)

r_i = ss_i.a.to(u.km)

7178.1366 km

v_i = norm(v_i_vec)

7.4518315 km/s

N = 1000
dv_a_vector = np.zeros(N) * u.km / u.s
dv_b_vector = dv_a_vector.copy()

for ii, r_f in enumerate(r_f_vector):
    man = Maneuver.hohmann(ss_i, r_f)
    (_, dv_a), (_, dv_b) = man.impulses
    dv_a_vector[ii] = norm(dv_a)
    dv_b_vector[ii] = norm(dv_b)

fig, ax = plt.subplots(figsize=(7, 7))

ax.plot((r_f_vector / r_i).value, (dv_a_vector / v_i).value, label="First impulse")
ax.plot((r_f_vector / r_i).value, (dv_b_vector / v_i).value, label="Second impulse")
ax.plot((r_f_vector / r_i).value, ((dv_a_vector + dv_b_vector) / v_i).value, label="Total cost")
ax.plot((r_f_vector / r_i).value, np.full(N, np.sqrt(2) - 1), 'k--')
ax.plot((r_f_vector / r_i).value, (1 / np.sqrt(r_f_vector / r_i)).value, 'k--')

ax.set_ylim(0, 0.7)
ax.set_xlabel("$R$")
ax.set_ylabel("$\Delta v_a / v_i$")
plt.legend()
fig.savefig("hohmann.png")

2.4. Jupyter notebooks 53
2.4.12 Using NEOS package

With the new poliastro version (0.7.0), a new package is included: NEOS package.

The docstrings of this package states the following:

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

So, first of all, an important question:

What are NEOs?

NEO stands for near-Earth object. The Center for NEO Studies (CNEOS) defines NEOs as comets and asteroids that have been nudged by the gravitational attraction of nearby planets into orbits that allow them to enter the Earth’s neighborhood.

And what does “near” exactly mean? In terms of orbital elements, asteroids and comets can be considered NEOs if their perihelion (orbit point which is nearest to the Sun) is less than 1.3 au = 1.945 * 108 km from the Sun.

```
[1]: from astropy import time
from poliastro.twobody.orbit import Orbit
from poliastro.bodies import Earth
from poliastro.plotting import OrbitPlotter2D
```
NeoWS module

This module makes requests to NASA NEO Webservice, so you’ll need an internet connection to run the next examples.

The simplest `neows` function is `orbit_from_name()`, which return an Orbit object given a name:

```
from poliastro.neos import neows
```

```
eros = neows.orbit_from_name("Eros")
frame = OrbitPlotter2D()
frame.plot(eros, label="Eros")
```

You can also search by IAU number or SPK-ID (there is a faster `neows.orbit_from_spk_id()` function in that case, although):

```
ganymed = neows.orbit_from_name("1036")  # Ganymed IAU number
amor = neows.orbit_from_name("2001221")  # Amor SPK-ID
eros = neows.orbit_from_spk_id("2000433")  # Eros SPK-ID
frame = OrbitPlotter2D()
frame.plot(ganymed, label="Ganymed")
frame.plot(amor, label="Amor")
frame.plot(eros, label="Eros")
```

Since `neows` relies on Small-Body Database browser to get the SPK-ID given a body name, you can use the wildcards from that browser: * and ?.

Keep it in mind that `orbit_from_name()` can only return one Orbit, so if several objects are found with that name, it will raise an error with the different bodies.

```
neows.orbit_from_name("*alley")
```

```
ValueError Traceback (most recent call last)
<ipython-input-5-2ff6e3b83a9b> in <module>
----> 1 neows.orbit_from_name("*alley")

~/Development/poliastro/poliastro-library/src/poliastro/neos/neows.py in orbit_from_name(name, api_key)
    139    140        """
-> 141    142        if spk_id is not None:
    143        return orbit_from_spk_id(spk_id, api_key)

~/Development/poliastro/poliastro-library/src/poliastro/neos/neows.py in orbit_from_spk_id(spk_id, api_key)
    108    109        for body in object_list[:obj_num]:
```

(continues on next page)
bodies += body.string + "\n"
raise ValueError(str(len(object_list)) + " different bodies found:\n" + bodies)

ValueError: 6 different bodies found:
903 Nealley (A918 RH)
2688 Halley (1982 HG1)
14182 Alley (1998 WG12)

Note that epoch is provided by the Web Service itself, so if you need orbit on another epoch, you have to propagate it:

```
[6]: eros.epoch.iso
[6]: '2019-04-27 00:00'

[7]: epoch = time.Time(2458000.0, scale="tdb", format="jd")
eros_november = eros.propagate(epoch)
eros_november.epoch.iso
[7]: '2017-09-03 12:00'
```

Given that we are using NASA APIs, there is a maximum number of requests. If you want to make many requests, it is recommended getting a NASA API key. You can use your API key adding the `api_key` parameter to the function:

```
[8]: neows.orbit_from_name("Toutatis", api_key="DEMO_KEY")
[8]: 1 x 4 AU x 0.4 deg (HeliocentricEclipticJ2000) orbit around Sun () at epoch 2455011.5 (TDB)
```

**DASTCOM5 module**

This module can also be used to get NEOs orbit, in the same way that `neows`, but it have some advantages (and some disadvantages).

It relies on DASTCOM5 database, a NASA/JPL maintained asteroid and comet database. This database has to be downloaded at least once in order to use this module. According to its README, it is updated typically a couple times per day, but potentially as frequently as once per hour, so you can download it whenever you want the more recently discovered bodies. This also means that, after downloading the file, you can use the database offline.

The file is a ~230 MB zip that you can manually download and unzip in `~/poliastro` or, more easily, you can use

```
dastcom5.download_dastcom5()
```

The main DASTCOM5 advantage over NeoWs is that you can use it to search not only NEOs, but any asteroid or comet. The easiest function is `orbit_from_name()`:

```
[9]: from poliastro.neos import dastcom5
```
atira = dastcom5.orbit_from_name("atira")[0]  # NEO
wikipedia = dastcom5.orbit_from_name("wikipedia")[0]  # Asteroid, but not NEO.

frame = OrbitPlotter2D()
frame.plot(atira, label="Atira (NEO)")
frame.plot(wikipedia, label="Wikipedia (asteroid)")

FigureWidget(
    'data': [{'hoverinfo': 'none',
              'line': {'color': 'rgb(31, 119, 180)', 'dash':...]

Keep in mind that this function returns a list of orbits matching your string. This is made on purpose given that there are comets which have several records in the database (one for each orbit determination in history) what allow plots like this one:

halleys = dastcom5.orbit_from_name("1P")

frame = OrbitPlotter2D()
frame.plot(halleys[0], label="Halley")
frame.plot(halleys[5], label="Halley")
frame.plot(halleys[10], label="Halley")
frame.plot(halleys[20], label="Halley")
frame.plot(halleys[-1], label="Halley")

FigureWidget(
    'data': [{'hoverinfo': 'none',
              'line': {'color': 'rgb(31, 119, 180)', 'dash':...]

While neows can only be used to get Orbit objects, dastcom5 can also provide asteroid and comet complete database. Once you have this, you can get specific data about one or more bodies. The complete databases are ndarrays, so if you want to know the entire list of available parameters, you can look at the dtype, and they are also explained in documentation API Reference:

ast_db = dastcom5.asteroid_db()
comet_db = dastcom5.comet_db()

ast_db.dtype.names[
    ...20 ...]

# They are more than 100, but that would be too much lines in this notebook :P
Asteroid and comet parameters are not exactly the same (although they are very close):

With these ndarrays you can classify asteroids and comets, sort them, get all their parameters, and whatever comes to your mind.

For example, NEOs can be grouped in several ways. One of the NEOs group is called Atiras, and is formed by NEOs whose orbits are contained entirely with the orbit of the Earth. They are a really little group, and we can try to plot all of these NEOs using asteroid_db():

Talking in orbital terms, Atiras have an aphelion distance, \( Q < 0.983 \) au and a semi-major axis, \( a < 1.0 \) au. Visiting documentation API Reference, you can see that DASTCOM5 provides semi-major axis, but doesn’t provide aphelion distance. You can get aphelion distance easily knowing perihelion distance (q, QR in DASTCOM5) and semi-major axis \( Q = 2a - q \), but there are probably many other ways.

```
[13]: aphelion_condition = 2 * ast_db["A"] - ast_db["QR"] < 0.983
axis_condition = ast_db["A"] < 1.3
atiras = ast_db[aphelion_condition & axis_condition]
```

The number of Atira NEOs we use using this method is:

```
[14]: len(atiras)
[14]: 16
```

Which is consistent with the stats published by CNEOS

Now we’re gonna plot all of their orbits, with corresponding labels, just because we love plots :)

```
[15]: from poliastro.twobody.orbit import Orbit
from poliastro.bodies import Earth

earth = Orbit.from_body_ephem(Earth)

We only need to get the 16 orbits from these 16 ndarrays.
There are two ways:

- Gather all their orbital elements manually and use the Orbit.from_classical() function.
- Use the NO property (logical record number in DASTCOM5 database) and the dastcom5.orbit_from_record() function.

The second one seems easier and it is related to the current notebook, so we are going to use that one:

We are going to use ASTNAM property of DASTCOM5 database:

```
[16]: import matplotlib.pyplot as plt
plt.ion()

from poliastro.plotting.static import StaticOrbitPlotter

frame = StaticOrbitPlotter()

frame.plot(earth, label="Earth")

for record in atiras["NO"]:  
        ss = dastcom5.orbit_from_record(record).to_icrs()  
        frame.plot(ss, color="#666666")
```
If we needed also the names of each asteroid, we could do:

```python
[17]: frame = StaticOrbitPlotter()

frame.plot(earth, label="Earth")

for i in range(len(atiras)):
    record = atiras["NO"][i]
    label = atiras["ASTNAM"][:i].decode().strip()  # DASTCOM5 strings are binary
    ss = dastcom5.orbit_from_record(record).to_icrs()
    frame.plot(ss, label=label)
```
We knew beforehand that there are no Atira comets, only asteroids (comet orbits are usually more eccentric), but we could use the same method with com_db if we wanted.

Finally, another interesting function in dastcom5 is entire_db(), which is really similar to ast_db and com_db, but it returns a Pandas dataframe instead of a numpy ndarray. The dataframe has asteroids and comets in it, but in order to achieve that (and a more manageable dataframe), a lot of parameters were removed, and others were renamed:

```python
[18]: db = dastcom5.entire_db()
db.columns

Index(['NUMBER', 'NOBS', 'OBSFRST', 'OBSLAST', 'EPOCH', 'CALEPOCH', 'MA', 'W', 'OM', 'IN', 'EC', 'A', 'QR', 'TP', 'TPCAL', 'TPFRAC', 'SOLDAT', 'DESIG', 'IREF', 'NAME'],
dtype='object')
```

Also, in this function, DASTCOM5 data (specially strings) is ready to use (decoded and improved strings, etc):

```python
[19]: db[db.NAME == "Halley"]
```

# As you can see, Halley is the name of an asteroid too, did you know that?

```
<table>
<thead>
<tr>
<th>NUMBER</th>
<th>NOBS</th>
<th>OBSFRST</th>
<th>OBSLAST</th>
<th>EPOCH</th>
<th>CALEPOCH</th>
<th>MA</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>2687</td>
<td>2688</td>
<td>1732</td>
<td>19500814</td>
<td>20160827</td>
<td>2455701.5</td>
<td>20110520.0</td>
<td>22.786977</td>
</tr>
<tr>
<td>739644</td>
<td>900001</td>
<td>161</td>
<td>0</td>
<td>0</td>
<td>1633920.5</td>
<td>-2390607.0</td>
<td>0.165294</td>
</tr>
<tr>
<td>739645</td>
<td>900002</td>
<td>161</td>
<td>0</td>
<td>0</td>
<td>1661840.5</td>
<td>-1631115.0</td>
<td>0.031113</td>
</tr>
<tr>
<td>739646</td>
<td>900003</td>
<td>161</td>
<td>0</td>
<td>0</td>
<td>1689880.5</td>
<td>-860823.0</td>
<td>0.142118</td>
</tr>
<tr>
<td>739647</td>
<td>900004</td>
<td>161</td>
<td>0</td>
<td>0</td>
<td>1717320.5</td>
<td>-111008.0</td>
<td>359.963217</td>
</tr>
<tr>
<td>739648</td>
<td>900005</td>
<td>161</td>
<td>0</td>
<td>0</td>
<td>1745200.5</td>
<td>660206.0</td>
<td>0.142118</td>
</tr>
</tbody>
</table>
```

(continues on next page)
<table>
<thead>
<tr>
<th>Time</th>
<th>std. dev.</th>
<th>std. dev.</th>
<th>std. dev.</th>
<th>std. dev.</th>
<th>std. dev.</th>
<th>std. dev.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.01</td>
<td>0.001</td>
<td>0.001</td>
<td>0.001</td>
<td>0.001</td>
<td>0.001</td>
<td>0.001</td>
</tr>
<tr>
<td>0.02</td>
<td>0.002</td>
<td>0.002</td>
<td>0.002</td>
<td>0.002</td>
<td>0.002</td>
<td>0.002</td>
</tr>
<tr>
<td>0.03</td>
<td>0.003</td>
<td>0.003</td>
<td>0.003</td>
<td>0.003</td>
<td>0.003</td>
<td>0.003</td>
</tr>
<tr>
<td>0.04</td>
<td>0.004</td>
<td>0.004</td>
<td>0.004</td>
<td>0.004</td>
<td>0.004</td>
<td>0.004</td>
</tr>
<tr>
<td>0.05</td>
<td>0.005</td>
<td>0.005</td>
<td>0.005</td>
<td>0.005</td>
<td>0.005</td>
<td>0.005</td>
</tr>
</tbody>
</table>

(continues from previous page)

### 2.4. Jupyter notebooks

---

(continues on next page)
<table>
<thead>
<tr>
<th>TP</th>
<th>TPCAL</th>
<th>TPFRAC</th>
<th>SOLDAT</th>
<th>DESIG</th>
</tr>
</thead>
<tbody>
<tr>
<td>2687</td>
<td>2.455571e+06</td>
<td>2.011011e+07</td>
<td>0.309024</td>
<td>2.457850e+06</td>
</tr>
<tr>
<td>739644</td>
<td>1.633908e+06</td>
<td>-2.390525e+06</td>
<td>0.309024</td>
<td>2.457850e+06</td>
</tr>
<tr>
<td>739645</td>
<td>1.661838e+06</td>
<td>-1.110108e+06</td>
<td>0.309024</td>
<td>2.457850e+06</td>
</tr>
<tr>
<td>739646</td>
<td>1.745189e+06</td>
<td>6.601260e+05</td>
<td>0.309024</td>
<td>2.457850e+06</td>
</tr>
<tr>
<td>739647</td>
<td>1.772639e+06</td>
<td>1.410322e+06</td>
<td>0.309024</td>
<td>2.457850e+06</td>
</tr>
<tr>
<td>739648</td>
<td>1.800091e+06</td>
<td>2.180518e+06</td>
<td>0.309024</td>
<td>2.457850e+06</td>
</tr>
<tr>
<td>739649</td>
<td>1.828916e+06</td>
<td>2.950420e+06</td>
<td>0.309024</td>
<td>2.457850e+06</td>
</tr>
<tr>
<td>739650</td>
<td>1.857708e+06</td>
<td>3.740216e+06</td>
<td>0.309024</td>
<td>2.457850e+06</td>
</tr>
<tr>
<td>739651</td>
<td>1.885964e+06</td>
<td>4.510628e+06</td>
<td>0.309024</td>
<td>2.457850e+06</td>
</tr>
<tr>
<td>739652</td>
<td>1.914910e+06</td>
<td>5.300927e+06</td>
<td>0.309024</td>
<td>2.457850e+06</td>
</tr>
<tr>
<td>739653</td>
<td>1.942838e+06</td>
<td>6.070315e+06</td>
<td>0.309024</td>
<td>2.457850e+06</td>
</tr>
<tr>
<td>739654</td>
<td>1.971164e+06</td>
<td>6.841003e+06</td>
<td>0.309024</td>
<td>2.457850e+06</td>
</tr>
<tr>
<td>739655</td>
<td>1.998788e+06</td>
<td>7.605521e+06</td>
<td>0.309024</td>
<td>2.457850e+06</td>
</tr>
<tr>
<td>739656</td>
<td>2.026831e+06</td>
<td>8.370228e+06</td>
<td>0.309024</td>
<td>2.457850e+06</td>
</tr>
<tr>
<td>739657</td>
<td>2.054365e+06</td>
<td>9.120719e+06</td>
<td>0.309024</td>
<td>2.457850e+06</td>
</tr>
<tr>
<td>739658</td>
<td>2.082538e+06</td>
<td>9.890906e+06</td>
<td>0.309024</td>
<td>2.457850e+06</td>
</tr>
<tr>
<td>739659</td>
<td>2.110493e+06</td>
<td>1.066032e+07</td>
<td>0.309024</td>
<td>2.457850e+06</td>
</tr>
<tr>
<td>739660</td>
<td>2.139377e+06</td>
<td>1.145042e+07</td>
<td>0.309024</td>
<td>2.457850e+06</td>
</tr>
<tr>
<td>739661</td>
<td>2.167664e+06</td>
<td>1.222093e+07</td>
<td>0.309024</td>
<td>2.457850e+06</td>
</tr>
<tr>
<td>739662</td>
<td>2.196546e+06</td>
<td>1.301103e+07</td>
<td>0.309024</td>
<td>2.457850e+06</td>
</tr>
<tr>
<td>739663</td>
<td>2.224686e+06</td>
<td>1.378111e+07</td>
<td>0.309024</td>
<td>2.457850e+06</td>
</tr>
<tr>
<td>739664</td>
<td>2.253022e+06</td>
<td>1.456061e+07</td>
<td>0.309024</td>
<td>2.457850e+06</td>
</tr>
<tr>
<td>739665</td>
<td>2.280493e+06</td>
<td>1.531083e+07</td>
<td>0.309024</td>
<td>2.457850e+06</td>
</tr>
<tr>
<td>739666</td>
<td>2.308304e+06</td>
<td>1.607103e+07</td>
<td>0.309024</td>
<td>2.457850e+06</td>
</tr>
<tr>
<td>739667</td>
<td>2.336566e+06</td>
<td>1.682092e+07</td>
<td>0.309024</td>
<td>2.457850e+06</td>
</tr>
<tr>
<td>739668</td>
<td>2.364528e+06</td>
<td>1.759031e+07</td>
<td>0.309024</td>
<td>2.457850e+06</td>
</tr>
<tr>
<td>739669</td>
<td>2.394599e+06</td>
<td>1.853112e+07</td>
<td>0.309024</td>
<td>2.457850e+06</td>
</tr>
<tr>
<td>739670</td>
<td>2.418782e+06</td>
<td>1.910042e+07</td>
<td>0.309024</td>
<td>2.457850e+06</td>
</tr>
<tr>
<td>739671</td>
<td>2.444647e+06</td>
<td>1.986021e+07</td>
<td>0.309024</td>
<td>2.457850e+06</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IREF</th>
<th>NAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>2687</td>
<td>23</td>
</tr>
<tr>
<td>739644</td>
<td>SAO/-239 Halley</td>
</tr>
<tr>
<td>739645</td>
<td>SAO/-163 Halley</td>
</tr>
<tr>
<td>739646</td>
<td>SAO/-86 Halley</td>
</tr>
<tr>
<td>739647</td>
<td>SAO/-11 Halley</td>
</tr>
<tr>
<td>739648</td>
<td>SAO/66 Halley</td>
</tr>
<tr>
<td>739649</td>
<td>SAO/141 Halley</td>
</tr>
<tr>
<td>739650</td>
<td>SAO/218 Halley</td>
</tr>
<tr>
<td>739651</td>
<td>SAO/295 Halley</td>
</tr>
<tr>
<td>739652</td>
<td>SAO/374 Halley</td>
</tr>
<tr>
<td>739653</td>
<td>SAO/451 Halley</td>
</tr>
<tr>
<td>739654</td>
<td>SAO/530 Halley</td>
</tr>
</tbody>
</table>
Panda offers many functionalities, and can also be used in the same way as the \texttt{ast\_db} and \texttt{comet\_db} functions:

\begin{verbatim}
[20]: aphelion_condition = \{2 * db["A"] - db["QR"]\} < 0.983
    axis_condition = db["A"] < 1.3
    atiras = db[aphelion_condition & axis_condition]
[21]: len(atiras)
[21]: 349
\end{verbatim}

What? I said they can be used in the same way!

Dont worry :) If you want to know what’s happening here, the only difference is that we are now working with comets too, and some comets have a negative semi-major axis!

\begin{verbatim}
[22]: len(atiras[atiras.A < 0])
[22]: 333
\end{verbatim}

So, rewriting our condition:

\begin{verbatim}
[23]: axis_condition = (db["A"] < 1.3) & (db["A"] > 0)
             atiras = db[aphelion_condition & axis_condition]
             len(atiras)
[23]: 16
\end{verbatim}

\section*{2.4.13 Visualizing the SpaceX Tesla Roadster trip to Mars}

\begin{verbatim}
[1]: from astropy.time import Time
    from poliastro.bodies import *
    from poliastro.twobody import Orbit
\end{verbatim}
from poliastro.plotting import OrbitPlotter3D

EPOCH = Time("2018-02-18 12:00:00", scale="tdb")

roadster = Orbit.from_horizons(name="SpaceX Roadster", epoch=EPOCH, id_type="majorbody")
roadster

1 x 2 AU x 24.2 deg (HCRS) orbit around Sun () at epoch 2018-02-18 12:00:00.000 (TDB)

from poliastro.plotting.misc import plot_solar_system

frame = plot_solar_system(outer=False, epoch=EPOCH)
frame.plot(roadster, label="SpaceX Roadster", color="black")

Frame <class 'astropy.coordinates.builtin_frames.icrs.ICRS'> does not support _'obstime', time values were not returned

FigureWidget({
    'data': [{'hoverinfo': 'none',
              'line': {'color': 'rgb(31, 119, 180)', 'dash':...}

frame = OrbitPlotter3D()

frame.plot(Orbit.from_body_ephem(Earth, EPOCH), label=Earth)
frame.plot(Orbit.from_body_ephem(Mars, EPOCH), label=Mars)
frame.plot(roadster, label="SpaceX Roadster", color="black")

frame.set_view(30 * u.deg, -100 * u.deg, 2 * u.km)
FigureWidget({
    'data': [{'line': {'color': 'rgb(31, 119, 180)', 'dash': 'dash', 'width': 5},
              ...

2.4.14 Natural and artificial perturbations

import numpy as np
import matplotlib.pyplot as plt
plt.ion()

from astropy import units as u
from astropy.time import Time, TimeDelta
from astropy.coordinates import solar_system_ephemeris

from poliastro.twobody.propagation import propagate, cowell
from poliastro.ephem import build_ephem_interpolant
from poliastro.core.elements import rv2coe

from poliastro.core.util import norm
from poliastro.core.perturbations import atmospheric_drag, third_body, J2_perturbation

(continues on next page)
Atmospheric drag

The poliastro package now has several commonly used natural perturbations. One of them is atmospheric drag! See how one can monitor decay of the near-Earth orbit over time using our new module poliastro.twobody.perturbations!

```
from poliastro.bodies import Earth, Moon
from poliastro.twobody import Orbit
from poliastro.plotting import OrbitPlotter2D, OrbitPlotter3D

R = Earth.R.to(u.km).value
k = Earth.k.to(u.km ** 3 / u.s ** 2).value

orbit = Orbit.circular(Earth, 250 * u.km, epoch=Time(0.0, format="jd", scale="tdb"))

# parameters of a body
C_D = 2.2  # dimensionless (any value would do)
A = ((np.pi / 4.0) * (u.m ** 2)).to(u.km ** 2).value  # km^2
m = 100  # kg
B = C_D * A / m

# parameters of the atmosphere
rho0 = Earth.rho0.to(u.kg / u.km ** 3).value  # kg/km^3
H0 = Earth.H0.to(u.km).value

tofs = TimeDelta(np.linspace(0 * u.h, 100000 * u.s, num=2000))

rr = propagate(
    orbit,
    tofs,
    method=cowell,
    ad=atmospheric_drag,
    R=R,
    C_D=C_D,
    A=A,
    m=m,
    H0=H0,
    rho0=rho0,
)

plt.ylabel("h(t)")
plt.xlabel("t, days")
plt.plot(tofs.value, rr.data.norm() - Earth.R);
```
Evolution of RAAN due to the J2 perturbation

We can also see how the J2 perturbation changes RAAN over time!

```
[4]: r0 = np.array([-2384.46, 5729.01, 3050.46]) * u.km
   v0 = np.array([-7.36138, -2.98997, 1.64354]) * u.km / u.s
   orbit = Orbit.from_vectors(Earth, r0, v0)
   tofs = TimeDelta(np.linspace(0, 48.0 * u.h, num=2000))
   coords = propagator(orbit, tofs, method=cowell, ad=J2_perturbation, J2=Earth.J2.value, R=Earth.R.to(u.km).value)
   rr = coords.data.xyz.T.to(u.km).value
   vv = coords.data.differentials["s"].d_xyz.T.to(u.km / u.s).value

# This will be easier to compute when this is solved:
# https://github.com/poliastro/poliastro/issues/257
   raans = [rv2coe(k, r, v)[3] for r, v in zip(rr, vv)]
[5]: plt.ylabel("RAAN(t)")
   plt.xlabel("t, h")
   plt.plot(tofs.value, raans);
```
3rd body

Apart from time-independent perturbations such as atmospheric drag, J2/J3, we have time-dependent perturbations. Let’s see how Moon changes the orbit of GEO satellite over time!

```
# database keeping positions of bodies in Solar system over time
solar_system_ephemeris.set("de432s")

epoch = Time(2454283.0, format="jd", scale="tdb")  # setting the exact event date is important

# create interpolant of 3rd body coordinates (calling in on every iteration will be just too slow)
body_r = build_ephem_interpolant(
    Moon, 28 * u.day, (epoch.value * u.day, epoch.value * u.day + 60 * u.day),
    rtol=1e-2
)

initial = Orbit.from_classical(
    Earth,
    42164.0 * u.km,
    0.0001 * u.one,
    1 * u.deg,
    0.0 * u.deg,
    0.0 * u.deg,
    0.0 * u.rad,
    epoch=epoch,
)

tofs = TimeDelta(np.linspace(0, 60 * u.day, num=1000))

# multiply Moon gravity by 400 so that effect is visible :)
rr = propagate(
    initial,
    tofs,
    method=cowell,
    rtol=1e-6,
)
```
Apart from natural perturbations, there are artificial thrusts aimed at intentional change of orbit parameters. One of such changes is simultaneous change of eccentricity and inclination.

```python
from poliastro.twobody.thrust import change_inc_ecc

ecc_0, ecc_f = 0.4, 0.0
a = 42164 # km
inc_0 = 0.0 # rad, baseline
inc_f = (20.0 * u.deg).to(u.rad).value # rad
argp = 0.0 # rad, the method is efficient for 0 and 180
f = 2.4e-6 # km / s^2

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)
tofs = TimeDelta(np.linspace(0, t_f * u.s, num=1000))
rr2 = propagate(s0, tofs, method=cowell, rtol=1e-6, ad=a_d)
```

Thrusted orbits can also be plotted:

```python
frame = OrbitPlotter3D()
frame.set_attractor(Earth)
frame.plot_trajectory(rr2, label='orbit with artificial thrust')
```
2.5 High level API

The `poliastro.twobody` is one of the main modules of `poliastro` since enables the user in a very intuitive way to create, propagate and apply maneuvers to an orbit due to its OOP nature.

2.5.1 Twobody module

The `poliastro.twobody` is one of the most important modules since it includes the `poliastro.twobody` module.

### Angles

Angles and anomalies.

#### `D_to_nu(D)`

True anomaly from parabolic eccentric anomaly.

- **Parameters** `D (Quantity)` – Eccentric anomaly.
- **Returns** `nu` – True anomaly.
- **Return type** `Quantity`

#### `nu_to_D(nu)`

Parabolic eccentric anomaly from true anomaly.

- **Parameters** `nu (Quantity)` – True anomaly.
- **Returns** `D` – Hyperbolic eccentric anomaly.
- **Return type** `Quantity`

#### `nu_to_E(nu, ecc)`

Eccentric anomaly from true anomaly.

- **New in version 0.4.0.**
- **Parameters**
  - `nu (Quantity)` – True anomaly.
• \texttt{ecc (Quantity)} – Eccentricity.

Returns \texttt{E} – Eccentric anomaly.

Return type \texttt{Quantity}

\texttt{poliastro.twobody.angles.nu_to_F(nu, ecc)}

Hyperbolic eccentric anomaly from true anomaly.

Parameters

• \texttt{nu (Quantity)} – True anomaly.
• \texttt{ecc (Quantity)} – Eccentricity (>1).

Returns \texttt{F} – Hyperbolic eccentric anomaly.

Return type \texttt{Quantity}

\textbf{Note:} Taken from Curtis, H. (2013). \textit{Orbital mechanics for engineering students}. 167

\texttt{poliastro.twobody.angles.E_to_nu(E, ecc)}

True anomaly from eccentric anomaly.

New in version 0.4.0.

Parameters

• \texttt{E (Quantity)} – Eccentric anomaly.
• \texttt{ecc (Quantity)} – Eccentricity.

Returns \texttt{nu} – True anomaly.

Return type \texttt{Quantity}

\texttt{poliastro.twobody.angles.F_to_nu(F, ecc)}

True anomaly from hyperbolic eccentric anomaly.

Parameters

• \texttt{F (Quantity)} – Hyperbolic eccentric anomaly.
• \texttt{ecc (Quantity)} – Eccentricity (>1).

Returns \texttt{nu} – True anomaly.

Return type \texttt{Quantity}

\texttt{poliastro.twobody.angles.M_to_E(M, ecc)}

Eccentric anomaly from mean anomaly.

New in version 0.4.0.

Parameters

• \texttt{M (Quantity)} – Mean anomaly.
• \texttt{ecc (Quantity)} – Eccentricity.

Returns \texttt{E} – Eccentric anomaly.

Return type \texttt{Quantity}

\texttt{poliastro.twobody.angles.M_to_F(M, ecc)}

Hyperbolic eccentric anomaly from mean anomaly.
Parameters

- \( M(Quantity) \) – Mean anomaly.
- \( \text{ecc}(Quantity) \) – Eccentricity (>1).

Returns \( F \) – Hyperbolic eccentric anomaly.

Return type Quantity

\text{poliastro.twobody.angles.M_to_D}(M, \text{ecc})

Parabolic eccentric anomaly from mean anomaly.

Parameters

- \( M(Quantity) \) – Mean anomaly.
- \( \text{ecc}(Quantity) \) – Eccentricity (>1).

Returns \( D \) – Parabolic eccentric anomaly.

Return type Quantity

\text{poliastro.twobody.angles.E_to_M}(E, \text{ecc})

Mean anomaly from eccentric anomaly.

New in version 0.4.0.

Parameters

- \( E(Quantity) \) – Eccentric anomaly.
- \( \text{ecc}(Quantity) \) – Eccentricity.

Returns \( M \) – Mean anomaly.

Return type Quantity

\text{poliastro.twobody.angles.F_to_M}(F, \text{ecc})

Mean anomaly from eccentric anomaly.

Parameters

- \( F(Quantity) \) – Hyperbolic eccentric anomaly.
- \( \text{ecc}(Quantity) \) – Eccentricity (>1).

Returns \( M \) – Mean anomaly.

Return type Quantity

\text{poliastro.twobody.angles.D_to_M}(D, \text{ecc})

Mean anomaly from eccentric anomaly.

Parameters

- \( D(Quantity) \) – Parabolic eccentric anomaly.
- \( \text{ecc}(Quantity) \) – Eccentricity.

Returns \( M \) – Mean anomaly.

Return type Quantity

\text{poliastro.twobody.angles.M_to_nu}(M, \text{ecc}, \text{delta}=0.01)

True anomaly from mean anomaly.

New in version 0.4.0.

Parameters
• \( M(\text{Quantity}) \) – Mean anomaly.
• \( \text{ecc}(\text{Quantity}) \) – Eccentricity.
• \( \text{delta}(\text{float (optional)}) \) – threshold of near-parabolic regime definition (from Davide Farnocchia et al)

**Returns** \( \text{nu} \) – True anomaly.

**Return type** \( \text{Quantity} \)

**Examples**

```python
>>> 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**

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

**Returns** \( \text{M} \) – Mean anomaly.

**Return type** \( \text{Quantity} \)

```

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

**New in version 0.4.0.**

**Parameters**

• \( \text{nu}(\text{Quantity}) \) – True anomaly.
• \( \text{ecc}(\text{Quantity}) \) – Eccentricity.

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

**Decorators**

Decorators.

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

**Examples**
```python
>>> 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.

**Orbit**

This is probably one of the main important modules of poliastro since it enables the user to create the `poliastro.twobody.orbit.Orbit` objects.

User can create or initialize an orbit in a very different ways but the most common ones are the following:

```
poliastro.twobody.orbit.Orbit
from_vectors from_classical from_equinoctial from_body_ephem
```

However, it is also possible to create 'special' orbits such as `poliastro.twobody.orbit.Orbit.circular` or even `poliastro.twobody.orbit.Orbit.parabolic`.

Furthermore, orbits can be sampled by `poliastro.twobody.orbit.Orbit.sample` or propagated in time by `poliastro.twobody.orbit.Orbit.propagate`. But even it is possible to apply a maneuver to an orbit with the `poliastro.twobody.orbit.Orbit.apply_maneuver`.

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

**exception** `poliastro.twobody.orbit.OrbitSamplingWarning`

**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.

**attractor**

Main attractor.

**epoch**

Epoch of the orbit.
**plane**
Fundamental plane of the frame.

**frame**
Reference frame of the orbit.
New in version 0.11.0.

**r**
Position vector.

**v**
Velocity vector.

**a**
Semilatus rectum.

**p**
Semimajor axis.

**r_p**
Radius of pericenter.

**r_a**
Radius of apocenter.

**ecc**
Eccentricity.

**inc**
Inclination.

**raan**
Right ascension of the ascending node.

**argp**
Argument of the perigee.

**nu**
True anomaly.

**f**
Second modified equinoctial element.

**g**
Third modified equinoctial element.

**h**
Fourth modified equinoctial element.

**k**
Fifth modified equinoctial element.

**L**
True longitude.

**period**
Period of the orbit.

**n**
Mean motion.

**energy**
Specific energy.
e_vec
    Eccentricity vector.

h_vec
    Specific angular momentum vector.

arglat
    Argument of latitude.

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_coords(attractor, coord, plane=<Planes.EARTH_EQUATOR: 'Earth mean Equator and Equinox of epoch (J2000.0)'>)

Creates an Orbit from an attractor and astropy SkyCoord or BaseCoordinateFrame instance.

This method accepts position and velocity in any reference frame unlike Orbit.from_vector which can accept inputs in only inertial reference frame centred at attractor. Also note that the frame information is lost after creation of the orbit and only the inertial reference frame at body centre will be used for all purposes.

Parameters

• attractor (Body) – Main attractor
• coord (astropy.coordinates.SkyCoord or BaseCoordinateFrame) – Position and velocity vectors in any reference frame. Note that coord must have a representation and its differential with respect to time.
• plane (Planes, optional) – Final orbit plane, default to Earth Equator.

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 from_horizons** (*name, epoch=None, plane=<Planes.EARTH_EQUATOR: 'Earth mean Equator and Equinox of epoch (J2000.0)'>, id_type='smallbody')

Return osculating *Orbit* of a body using JPLHorizons module of Astroquery.

**Parameters**

• **name** (*string*) – Name of the body to query for.

• **epoch** (*Time, optional*) – Epoch, default to None.

• **plane** (*Planes*) – Fundamental plane of the frame.

• **id_type** (*string, optional*) – Use “smallbody” for Asteroids and Comets, and “majorbody” for Planets and Satellites.

**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 geostationary**(*attractor*, *angular_velocity=None*, *period=None*, *hill_radius=None*)

Return the geostationary orbit for the given attractor and its rotational speed.

**Parameters**

• **attractor** (*Body*) – Main attractor.

• **angular_velocity** (*Quantity*) – Rotational angular velocity of the attractor.

• **period** (*Quantity*) – Attractor’s rotational period, ignored if angular_velocity is passed.

• **hill_radius** (*Quantity*) – Radius of Hill sphere of the attractor (optional). Hill sphere radius (in contrast with Laplace’s SOI) is used here to validate the stability of the geostationary orbit, that is to make sure that the orbital radius required for the geostationary orbit is not outside of the gravitational sphere of influence of the attractor. Hill SOI of parent (if exists) of the attractor is ignored if hill_radius is not provided.

**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**

```python
>>> from poliastro.examples import iss
>>> from astropy.coordinates import CartesianRepresentation,
    SphericalRepresentation
>>> iss.represent_as(CartesianRepresentation)
```
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.

classical()

Classical orbital elements.

pqw()

Perifocal frame (PQW) vectors.

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

Propagates an orbit a specified time.

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 (Quantity, Time, TimeDelta) – Scalar time to propagate.
- 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

Returns New orbit after propagation.

Return type Orbit

propagate_to_anomaly (value)

Propagates an orbit to a specific true anomaly.

Parameters value (Quantity) –

Returns Resulting orbit after propagation.

Return type Orbit
sample(values=100, *, min_anomaly=None, max_anomaly=None, method=<function mean_motion>)

Samples an orbit to some specified time values.

New in version 0.8.0.

Parameters

- **values (int)** – Number of interval points (default to 100).
- **max_anomaly (min_anomaly,)* – Anomaly limits to sample the orbit. For elliptic orbits the default will be \( E \in [0, 2\pi] \), and for hyperbolic orbits it will be \( \nu \in [-\nu_c, \nu_c] \), where \( \nu_c \) is either the current true anomaly or a value that corresponds to \( r = 3p \).
- **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

```python
>>> from astropy import units as u
>>> from poliastro.examples import iss
>>> iss.sample() # doctest: +ELLIPSIS
<GCRS Coordinate ...>
>>> iss.sample(10) # 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.

plot(label=None, use_3d=False)

Plots the orbit as an interactive widget.

Parameters

- **label (str, optional)** – Label for the orbit, defaults to empty.
- **use_3d (bool, optional)** – Produce a 3D plot, default to False.
poliastro Documentation, Release 0.12.0

poliastro.twobody.propagation.cowell \((k, r, v, \text{tofs}, rtol=1e-11, *, ad=None, **ad_kwargs)\)
Propagates orbit using Cowell’s formulation.

Parameters
- \(k\) (Quantity) – Standard gravitational parameter of the attractor.
- \(r\) (Quantity) – Position vector.
- \(v\) (Quantity) – Velocity vector.
- \(\text{tofs}\) (Quantity) – Array of times to propagate.
- \(rtol\) (float, optional) – Maximum relative error permitted, default to 1e-10.
- \(ad\) (function\((t0, u, k)\), optional) – Non Keplerian acceleration (km/s2), default to None.

Returns
- \(rr\) (~astropy.units.Quantity) – Propagated position vectors.
- \(vv\) (~astropy.units.Quantity) – Propagated velocity vectors.

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.mean_motion \((k, r, v, \text{tofs}, **kwargs)\)
Propagates orbit using Cowell’s formulation.

Parameters
- \(k\) (Quantity) – Standard gravitational parameter of the attractor.
- \(r\) (Quantity) – Position vector.
- \(v\) (Quantity) – Velocity vector.
- \(\text{tofs}\) (Quantity) – Array of times to propagate.

Returns
- \(rr\) (~astropy.units.Quantity) – Propagated position vectors.
- \(vv\) (~astropy.units.Quantity) – Propagated velocity vectors.

poliastro.twobody.propagation.kepler \((k, r, v, \text{tofs}, numiter=350, **kwargs)\)
Propagates Keplerian orbit.

Parameters
- \(k\) (Quantity) – Standard gravitational parameter of the attractor.
- \(r\) (Quantity) – Position vector.
- \(v\) (Quantity) – Velocity vector.
- \(\text{tofs}\) (Quantity) – Array of times to propagate.
- \(numiter\) (int, optional) – Maximum number of iterations, default to 35.

Returns
- \(rr\) (~astropy.units.Quantity) – Propagated position vectors.
• \texttt{vv} (\texttt{~astropy.units.Quantity}) – Propagated velocity vectors.

\textbf{Raises} \texttt{RuntimeError} – If the algorithm didn’t converge.

\textbf{Note:} This algorithm is based on Vallado implementation, and does basic Newton iteration on the Kepler equation written using universal variables. Battin claims his algorithm uses the same amount of memory but is between 40\% and 85\% faster.

\begin{verbatim}
poliastro.twobody.propagation.propagate(orbit, time_of_flight, *, method=<function mean_motion>, rtol=1e-10, **kwargs)
\end{verbatim}

Propagate an orbit some time and return the result.

\textbf{Parameters}

\begin{itemize}
  \item \texttt{orbit} (\texttt{Orbit}) – Orbit object to propagate.
  \item \texttt{time_of_flight} (\texttt{TimeDelta}) – Time of propagation.
  \item \texttt{method} (callable, optional) – Propagation method, default to \texttt{mean_motion}.
  \item \texttt{rtol} (float, optional) – Relative tolerance, default to 1e-10.
\end{itemize}

\subsection{2.5.2 Threebody module}

The \texttt{poliastro.threebody} contains a set of modules related to this physical problem:

\textbf{Flybys}

\begin{verbatim}
poliastro.threebody.flybys.compute_flyby(v_spacecraft, v_body, k, r_p, theta=<Quantity 0. deg>)
\end{verbatim}

Computes outbound velocity after a flyby.

\textbf{Parameters}

\begin{itemize}
  \item \texttt{v_spacecraft} (\texttt{Quantity}) – Velocity of the spacecraft, relative to the attractor of the body.
  \item \texttt{v_body} (\texttt{Quantity}) – Velocity of the body, relative to its attractor.
  \item \texttt{k} (\texttt{Quantity}) – Standard gravitational parameter of the body.
  \item \texttt{r_p} (\texttt{Quantity}) – Radius of periapsis, measured from the center of the body.
  \item \texttt{theta} (\texttt{Quantity}, optional) – Aim angle of the \texttt{B} vector, default to 0.
\end{itemize}

\textbf{Returns}

\begin{itemize}
  \item \texttt{v_spacecraft_out} (\texttt{~astropy.units.Quantity}) – Outbound velocity of the spacecraft.
  \item \texttt{delta} (\texttt{~astropy.units.Quantity}) – Turn angle.
\end{itemize}

\textbf{Restricted}

Circular Restricted 3-Body Problem (CR3BP)

Includes the computation of Lagrange points
poliastro.threebody.restricted.lagrange_points \((r_{12}, m_1, m_2)\)

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**
- \(r_{12}\) (Quantity) – Distance between the two bodies
- \(m_1\) (Quantity) – Mass of the main body
- \(m_2\) (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 \((m_1, r_1, m_2, r_2, n)\)

Computes the five Lagrange points in the CR3BP.

Returns the positions in the same frame of reference as \(r_1\) and \(r_2\) for the five Lagrangian points.

**Parameters**
- \(m_1\) (Quantity) – Mass of the main body. This body is the one with the biggest mass.
- \(r_1\) (Quantity) – Position of the main body.
- \(m_2\) (Quantity) – Mass of the secondary body.
- \(r_2\) (Quantity) – Position of the secondary body.
- \(n\) (Quantity) – Normal vector to the orbital plane.

**Returns** Position of the Lagrange points: \([L_1, L_2, L_3, L_4, L_5]\) The positions are of type ~astropy.units.Quantity

**Return type** list

**Sphere of Influence Module**

Sphere of Influence

Contains methods to compute radius of sphere of influence.

Laplace Radius: A laplace sphere of influence (SOI) in astrodynamics and astronomy is the oblate-spheroid-shaped region around a celestial body where the primary gravitational influence on an orbiting object is that body. This is usually used to describe the areas in the Solar System where planets dominate the orbits of surrounding objects such as moons, despite the presence of the much more massive but distant Sun. In the patched conic approximation, used in estimating the trajectories of bodies moving between the neighbourhoods of different masses using a two body approximation, ellipses and hyperbolae, the laplace radius is taken as the boundary where the trajectory switches which mass field it is influenced by. The result is:

\[
a \left( \frac{m}{M} \right)^{\frac{2}{3}}
\]

Hill Radius: In the three body problem, if that third object stays within an extremely complex boundary called the Roche lobe, the orbit of that third object about the smaller body will be stable for at least some amount of time. The Roche lobe just touches the \(L_1\) and \(L_2\) points and fans out from there. George Hill used the \(L_1\) point to define a sphere that approximated the Roche lobe. This is still intractable; the \(L_1\) point is defined by a fifth order polynomial that
cannot be solved in terms of the elementary functions. Hill further simplified things by realizing that a simple cubic equation yields a very good approximation of that intractable fifth order equation. The result is:

\[ a \left( \frac{m}{3M} \right)^{\frac{1}{3}} \]

poliastro.threebody.soi.laplace_radius(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.threebody.soi.hill_radius(body, a=None, e=None)

Approximated radius of the Hill 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).
- **e** (float, optional) – Eccentricity of the body’s orbit, default to 0 (will be computed from ephemerides).

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

Return type astropy.units.quantity.Quantity

### 2.5.3 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*

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

### 2.5.4 Neos module

NEO stands for near-Earth object. These bodies are generally asteroids or comets such that because of their orbits could be close into Earth’s surroundings. NEOs can be classified into:

- Meteroids: with a diameter lower than 50 meters
- Near Earth Comets (NEC): with orbital period lower than 200 years
- Near Earth Asteroid (NEA): most of NEO objects are in this category

*poliastro.neos* was developed as a result for the SOCIS 2017 Edition. It works by sending a request to and *NASANeo* webpage downloading all the parameters that define the orbit of the body. This means that Internet connection is required to accomplish this task.

### Dastcom5

The *poliastro.neos.dastcom5* is a module that was developed as a part of the GSOC 2017 Edition. The procedure for coding this module can be checked here *DASTCOM5*. NEOs orbit from DASTCOM5 database.

```python
class poliastro.neos.dastcom5.asteroid_db()
    Return complete DASTCOM5 asteroid database.
    Returns database – Database with custom dtype.
    Return type numpy.ndarray

class poliastro.neos.dastcom5.comet_db()
    Return complete DASTCOM5 comet database.
    Returns database – Database with custom dtype.
    Return type numpy.ndarray

class 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)

class 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.

- **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**

`avail`:
- `a` if it is available for asteroids.
- `c` if it is available for comets.
- `{number}+` since which version of DASTCOM5 is available.
<table>
<thead>
<tr>
<th>avail</th>
<th>Label</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>ac/3+</td>
<td>EPOCH</td>
<td>Time of osc. orbital elements solution, JD (CT,TDB)</td>
</tr>
<tr>
<td>ac/3+</td>
<td>CALEPO</td>
<td>Time of osc. orbital elements solution, YYYYDDMM.ffff</td>
</tr>
<tr>
<td>ac/3+</td>
<td>MA</td>
<td>Mean anomaly at EPOCH, deg (elliptical &amp; hyperbolic cases “9.999999E99” if not available)</td>
</tr>
<tr>
<td>ac/3+</td>
<td>W</td>
<td>Argument of periapsis at EPOCH, J2000 ecliptic, deg.</td>
</tr>
<tr>
<td>ac/3+</td>
<td>OM</td>
<td>Longitude of ascending node at EPOCH, J2000 ecliptic, deg.</td>
</tr>
<tr>
<td>ac/3+</td>
<td>IN</td>
<td>Inclination angle at EPOCH wrt J2000 ecliptic, deg.</td>
</tr>
<tr>
<td>ac/3+</td>
<td>EC</td>
<td>Eccentricity at EPOCH</td>
</tr>
<tr>
<td>ac/3+</td>
<td>A</td>
<td>Semi-major axis at EPOCH, au</td>
</tr>
<tr>
<td>ac/3+</td>
<td>QR</td>
<td>Perihelion distance at EPOCH, au</td>
</tr>
<tr>
<td>ac/3+</td>
<td>TP</td>
<td>Perihelion date for QR at EPOCH, JD (CT,TDB)</td>
</tr>
<tr>
<td>ac/3+</td>
<td>TPCAL</td>
<td>Perihelion date for QR at EPOCH, format YYYYMMDD.fff</td>
</tr>
<tr>
<td>ac/5+</td>
<td>TPFRAC</td>
<td>Decimal (fractional) part of TP for extended precision</td>
</tr>
<tr>
<td>ac/4+</td>
<td>SOLDAT</td>
<td>Date orbit solution was performed, JD (CT,TDB)</td>
</tr>
<tr>
<td>ac/4+</td>
<td>SRC(01)</td>
<td>Square root covariance vector. Vector-stored upper- triangular matrix with order [EC,QR,TP,OM,W,IN,</td>
</tr>
<tr>
<td>ac/3+</td>
<td>H</td>
<td>Absolute visual magnitude (IAU H-G system) (99=unknown)</td>
</tr>
<tr>
<td>ac/3+</td>
<td>G</td>
<td>Mag. slope parm. (IAU H-G) (99=unknown &amp; 0.15 not assumed)</td>
</tr>
<tr>
<td>c/3+</td>
<td>M1</td>
<td>Total absolute magnitude, mag.</td>
</tr>
<tr>
<td>c/3+</td>
<td>M2</td>
<td>Nuclear absolute magnitude, mag.</td>
</tr>
<tr>
<td>c/4+</td>
<td>K1</td>
<td>Total absolute magnitude scaling factor</td>
</tr>
<tr>
<td>c/4+</td>
<td>K2</td>
<td>Nuclear absolute magnitude scaling factor</td>
</tr>
<tr>
<td>c/4+</td>
<td>PHCOF</td>
<td>Phase coefficient for K2= 5</td>
</tr>
<tr>
<td>ac/3+</td>
<td>A1</td>
<td>Non-grav. accel., radial component, [s:10^-8 au/day^2]</td>
</tr>
<tr>
<td>ac/3+</td>
<td>A2</td>
<td>Non-grav. accel., transverse component,[s:10^-8 au/day^2</td>
</tr>
<tr>
<td>ac/4+</td>
<td>A3</td>
<td>Non-grav. accel., normal component, [s:10^-8 au/day^2]</td>
</tr>
<tr>
<td>c/4+</td>
<td>DT</td>
<td>Non-grav. lag/delay parameter, days</td>
</tr>
<tr>
<td>ac/5+</td>
<td>R0</td>
<td>Non-grav. model constant, normalizing distance, au</td>
</tr>
<tr>
<td>ac/5+</td>
<td>ALN</td>
<td>Non-grav. model constant, normalizing factor</td>
</tr>
<tr>
<td>ac/5+</td>
<td>NM</td>
<td>Non-grav. model constant, exponent m</td>
</tr>
<tr>
<td>ac/5+</td>
<td>NN</td>
<td>Non-grav. model constant, exponent n</td>
</tr>
<tr>
<td>ac/5+</td>
<td>NK</td>
<td>Non-grav. model constant, exponent k</td>
</tr>
<tr>
<td>c/4+</td>
<td>S0</td>
<td>Center-of-light estimated offset at 1 au, km</td>
</tr>
<tr>
<td>c/5+</td>
<td>TCL</td>
<td>Center-of-light start-time offset, d since “ref.time”</td>
</tr>
<tr>
<td>a/5+</td>
<td>LGK</td>
<td>Surface thermal conductivity log_10(k), (W/m/K)</td>
</tr>
<tr>
<td>ac/5+</td>
<td>RHO</td>
<td>Bulk density, kg/m^3</td>
</tr>
<tr>
<td>ac/5+</td>
<td>AMRAT</td>
<td>Solar pressure model, area/mass ratio, m^2/kg</td>
</tr>
<tr>
<td>c/5+</td>
<td>AJ1</td>
<td>Jet 1 acceleration, au/d^2</td>
</tr>
<tr>
<td>c/5+</td>
<td>AJ2</td>
<td>Jet 2 acceleration, au/d^2</td>
</tr>
<tr>
<td>c/5+</td>
<td>ET1</td>
<td>Thrust angle, colatitude of jet 1, deg.</td>
</tr>
<tr>
<td>c/5+</td>
<td>ET2</td>
<td>Thrust angle, colatitude of jet 2, deg.</td>
</tr>
<tr>
<td>c/5+</td>
<td>DTH</td>
<td>Jet model diurnal lag angle, deg. (delta_theta)</td>
</tr>
<tr>
<td>ac/5+</td>
<td>ALF</td>
<td>Spin pole orientation, RA, deg.</td>
</tr>
<tr>
<td>ac/5+</td>
<td>DEL</td>
<td>Spin pole orientation, DEC, deg.</td>
</tr>
<tr>
<td>ac/5+</td>
<td>SPHLM3</td>
<td>Earth gravity sph. harm. model limit, Earth radii</td>
</tr>
<tr>
<td>ac/5+</td>
<td>SPHLM5</td>
<td>Jupiter grav. sph. harm. model limit, Jupiter radii</td>
</tr>
<tr>
<td>ac/3+</td>
<td>RP</td>
<td>Object rotational period, hrs</td>
</tr>
<tr>
<td>ac/3+</td>
<td>GM</td>
<td>Object mass parameter, km^3/s^2</td>
</tr>
<tr>
<td>ac/3+</td>
<td>RAD</td>
<td>Object mean radius, km</td>
</tr>
<tr>
<td>ac/5+</td>
<td>EXTNT1</td>
<td>Triaxial ellipsoid, axis 1/largest equat. extent, km</td>
</tr>
<tr>
<td>ac/5+</td>
<td>EXTNT2</td>
<td>Triaxial ellipsoid, axis 2/smallest equat. extent, km</td>
</tr>
<tr>
<td>Label</td>
<td>Definition</td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td>---------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>ac/5+</td>
<td>EXTNT3 Triaxial ellipsoid, axis 3/polar extent, km</td>
<td></td>
</tr>
<tr>
<td>ac/4+</td>
<td>MOID Earth MOID at EPOCH time, au; ‘99’ if not computed</td>
<td></td>
</tr>
<tr>
<td>ac/3+</td>
<td>ALBEDO Geometric visual albedo, 99 if unknown</td>
<td></td>
</tr>
<tr>
<td>a /3+</td>
<td>BVCI B-V color index, mag., 99 if unknown</td>
<td></td>
</tr>
<tr>
<td>a /5+</td>
<td>UBCI U-B color index, mag., 99 if unknown</td>
<td></td>
</tr>
<tr>
<td>a /5+</td>
<td>IRCI 1-R color index, mag., 99 if unknown</td>
<td></td>
</tr>
<tr>
<td>ac/4+</td>
<td>RMSW RMS of weighted optical residuals, arcsec</td>
<td></td>
</tr>
<tr>
<td>ac/5+</td>
<td>RMSU RMS of unweighted optical residuals, arcsec</td>
<td></td>
</tr>
<tr>
<td>ac/5+</td>
<td>RMSN RMS of normalized optical residuals</td>
<td></td>
</tr>
<tr>
<td>ac/5+</td>
<td>RMSNT RMS of all normalized residuals</td>
<td></td>
</tr>
<tr>
<td>a /5+</td>
<td>RMSH RMS of abs. visual magnitude (H) residuals, mag.</td>
<td></td>
</tr>
<tr>
<td>c /5+</td>
<td>RMSMT RMS of MT estimate residuals, mag.</td>
<td></td>
</tr>
<tr>
<td>c /5+</td>
<td>RMSMN RMS of MN estimate residuals, mag.</td>
<td></td>
</tr>
<tr>
<td>ac/3+</td>
<td>NO Logical record-number of this object in DASTCOM</td>
<td></td>
</tr>
<tr>
<td>ac/4+</td>
<td>NOBS Number of observations of all types used in orbit soln.</td>
<td></td>
</tr>
<tr>
<td>ac/4+</td>
<td>OBSFRST Start-date of observations used in fit, YYYYMMDD</td>
<td></td>
</tr>
<tr>
<td>ac/4+</td>
<td>OBSLAST Stop-date of observations used in fit, YYYYMMDD</td>
<td></td>
</tr>
<tr>
<td>ac/5+</td>
<td>PRELTV Planet relativity ‘bit-switch’ byte: bits 0-7 are set to 1 if relativity for corresponding planet was computed, 0 if not computed</td>
<td></td>
</tr>
<tr>
<td>ac/5+</td>
<td>SPHMX3 Earth grav. model max. degree; 0=point-mass, 2= J2 only, 3= up to J3 zonal, 22= 2x2 field, 33=3x3 field, etc.</td>
<td></td>
</tr>
<tr>
<td>ac/5+</td>
<td>SPHMX5 Jupiter grav. max. deg.; 0=point-mass, 2= J2 only, 3= up to J3 zonal, 22= 2x2 field, 33=3x3 field, etc.</td>
<td></td>
</tr>
<tr>
<td>ac/5+</td>
<td>JGSEP Galilean satellites used as sep. perturbers; 0=no 1=yes</td>
<td></td>
</tr>
<tr>
<td>ac/5+</td>
<td>TWOBOD Two-body orbit model flag; 0=no 1=yes</td>
<td></td>
</tr>
<tr>
<td>ac/5+</td>
<td>NSATS Number of satellites; 99 if unknown</td>
<td></td>
</tr>
<tr>
<td>ac/4+</td>
<td>UPCRM Orbit condition code; 99 if not computed</td>
<td></td>
</tr>
<tr>
<td>ac/4+</td>
<td>LSRC Length of square-root cov. vector SRC (# elements used)</td>
<td></td>
</tr>
<tr>
<td>c /3+</td>
<td>IPYR Perihelion year (i.e., 1976, 2012, 2018, etc.)</td>
<td></td>
</tr>
<tr>
<td>ac/3+</td>
<td>NDEL Number of radar delay measurements used in orbit soln.</td>
<td></td>
</tr>
<tr>
<td>ac/3+</td>
<td>NDOP Number of radar Doppler measurements used in orbit soln.</td>
<td></td>
</tr>
<tr>
<td>c /5+</td>
<td>NOBSMT Number of magnitude measurements used in total mag. soln.</td>
<td></td>
</tr>
<tr>
<td>c /5+</td>
<td>NOBSMN Number of magnitude measurements used in nuc. mag. soln.</td>
<td></td>
</tr>
<tr>
<td>c /3+</td>
<td>COMNUM IAU/ comet number (parsed from DESIG)</td>
<td></td>
</tr>
<tr>
<td>ac/3+</td>
<td>EQUONX Equinox of orbital elements (‘1950’ or ‘2000’)</td>
<td></td>
</tr>
<tr>
<td>ac/4+</td>
<td>PENAM Planetary ephemeris ID/name</td>
<td></td>
</tr>
<tr>
<td>ac/3+</td>
<td>SBNAM Small-body perturber ephemeris ID/name</td>
<td></td>
</tr>
<tr>
<td>a /3</td>
<td>SPTYPT Tholen spectral type</td>
<td></td>
</tr>
<tr>
<td>a /4+</td>
<td>SPTYPS SASS-II spectral type</td>
<td></td>
</tr>
<tr>
<td>ac/3+</td>
<td>DAKC Data arc span (year-year, OR integer # of days)</td>
<td></td>
</tr>
<tr>
<td>a /3+</td>
<td>COMNT1 Asteroid comment line #1</td>
<td></td>
</tr>
<tr>
<td>a /3+</td>
<td>COMNT2 Asteroid comment line #2</td>
<td></td>
</tr>
<tr>
<td>c /3+</td>
<td>COMNT3 Comet comment line #1</td>
<td></td>
</tr>
<tr>
<td>c /3+</td>
<td>COMNT4 Comet comment line #2</td>
<td></td>
</tr>
<tr>
<td>ac/3+</td>
<td>DESIG Object designation</td>
<td></td>
</tr>
<tr>
<td>ac/4+</td>
<td>ESTL Dynamic parameter estimation list. Last symbol set to ‘+’ if list is too long for field; check object record comments for full list</td>
<td></td>
</tr>
<tr>
<td>ac/3+</td>
<td>IREF Solution reference/ID/name</td>
<td></td>
</tr>
<tr>
<td>ac/3+</td>
<td>NAME Object name</td>
<td></td>
</tr>
</tbody>
</table>

2.5. High level API
NEOWS module

NEOs orbit from NEOWS and JPL SBDB

```python
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`

```python
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`

```python
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`
2.5.5 Plotting module

The \texttt{poliastro.plotting} contains a set of submodules in which the basic classes and functions for plotting orbit objects are described. This module contains the following submodules:

![Submodule Diagram]

- **Core module**
  This module contains the basic classes for plots in 2-dimensions and 3-dimensions:

```python
class poliastro.plotting.core.OrbitPlotter3D(figure=None, dark=False)
 OrbitPlotter3D class.
```

```python
class poliastro.plotting.core.OrbitPlotter2D(figure=None)
 OrbitPlotter2D class.
```

New in version 0.9.0.

- **set_frame** \((p\_vec, q\_vec, w\_vec)\)
  Sets perifocal frame.

  \textbf{Raises} \texttt{ValueError} – If the vectors are not a set of mutually orthogonal unit vectors.

---

2.5. High level API
plot (orbit, *, label=None, color=None)
Plots state and osculating orbit in their plane.

Parameters
- orbit (Orbit) – Orbit to plot.
- label (string, optional) – Label of the orbit.
- color (string, optional) – Color of the line and the position.

Misc module

The poliastro.plotting.misc module contains different miscellaneous related to plotting, such as plotting the solar system:

poliastro.plotting.misc.plot_solar_system (outer=True, epoch=None, use_3d=False)
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.
- use_3d (bool, optional) – Produce 3D plot, default to False.

Porkchop module

This module contains the function that enables the user to make porkchop plots between two poliastro.bodies.Body instances. Other parameters to customize the plots such us setting the maximum C3 or showing the time lines can be passed to this function.

This is the script for porkchop plotting

poliastro.plotting.porkchop.porkchop (departure_body, target_body, launch_span, arrival_span, ax=None, time_lines=True, max_c3=<Quantity 45. km2 / s2>)
Plots porkchop between two bodies.

Parameters
- departure_body (poliastro.bodies.Body) – Body from which departure is done
- target_body (poliastro.bodies.Body) – Body for targeting
- launch_span (astropy.time.Time) – Time span for launch
- arrival_span (astropy.time.Time) – Time span for arrival
- ax (matplotlib.axes.Axes) – For custom figures

Returns
- c3_launch (np.ndarray) – Characteristic launch energy
- c3_arrival (np.ndarray) – Characteristic arrival energy
- tof (np.ndarray) – Time of flight for each transfer
Example

```python
>>> from poliastro.plotting.porkchop import porkchop
>>> from poliastro.bodies import Earth, Mars
>>> from poliastro.util import time_range

>>> launch_span = time_range("2005-04-30", end="2005-10-07")
>>> arrival_span = time_range("2005-11-16", end="2006-12-21")
>>> c3dpt, c3arr, tof = porkchop(Earth, Mars, launch_span, arrival_span)
```

Static module

The `poliastro.plotting.static` module contains the basic class `StaticOrbitPlotter` that is used in by other modules of the `poliastro.plotting`.

```python
class poliastro.plotting.static.StaticOrbitPlotter(ax=None, num_points=150, dark=False)
```

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.

```python
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.

```python
plot_trajectory(trajectory, *, label=None, color=None)
```

Plots a precomputed trajectory.

Parameters

- `trajectory (BaseRepresentation, BaseCoordinateFrame)` – Trajectory to plot.
- `label (str, optional)` – Label.
- `color (str, optional)` – Color string.

```python
set_attractor(attractor)
```

Sets plotting attractor.

Parameters `attractor (Body)` – Central body.

```python
plot(orbit, label=None, color=None, method=<function mean_motion>)
```

Plots state and osculating orbit in their plane.

Util module

The `poliastro.plotting.util` module contains some utilities such as drawing the circles or spheres to represent the different celestial bodies.

2.5.6 Initial Orbit Determination (IOD) module

The initial orbit determination problem consists on finding the orbit that passes through two different position vectors during a given finite period of time.

`py:mod:poliastro` includes two algorithms under the name `lambert` for solving this problem, also known as Lambert’s problem. `poliastro.iod` is composed by the following two sub-modules:
Izzo

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.

Vallado

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.
2.5.7 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:


radii were obtained from:


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)

2.5.8 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.
- `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.
- 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

2.5.9 Examples module

Example data.

poliastro.examples.iss = 6772 x 6790 km x 51.6 deg (GCRS) orbit around Earth () at epoch 2013-03-18 12:00:00.000 (UTC)

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 epoch J2000.000 (TT)

Molniya orbit example
Soyuz geostationary transfer orbit (GTO) example

Comet 67P/Churyumov–Gerasimenko orbit example

2.5.10 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)

class 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.
2.5.11 Maneuver module

Orbital maneuvers.

```python
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>)
```

```python
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.

def get_total_time()
    Returns total time of the maneuver.

def get_total_cost()
    Returns total cost of the maneuver.
```

2.5.12 Util module

Function helpers.

```python
costiastro.util.circular_velocity(k, a)
    Compute circular velocity for a given body ($k$) and semimajor axis ($a$).

costiastro.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

poliastro.util.hyp_nu_limit(ecc, r_max_ratio=inf)
Limit true anomaly for hyperbolic orbits.

Parameters
- ecc(Quantity) – Eccentricity, should be larger than 1.
- r_max_ratio(float, optional) – Value of $r_{max}/p$ for this angle, default to infinity.

2.6 Core level API

The poliastro.core includes a set of modules that form the kernel of poliastro. All defined functions in these modules work with Numba’s JIT compiler (Numba web site), since many of the computations behind poliastro involve hard numerical methods. This is the Low-Level API of poliastro, being the basis of the High-Level API, the one expected to be used.
The `poliastro.core.angles` module contains functions related to conversion between different angles used to define different orbital elements.

**poliastro.core.angles.D_to_nu**

True anomaly from parabolic eccentric anomaly.

\[ \nu = 2 \cdot \arctan (D) \]

**Parameters**

- `D (Quantity)` – Eccentric anomaly.

**Returns**

- `nu` – True anomaly.

**Return type** `Quantity`

**Note:** 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.core.angles.nu_to_D**

Parabolic eccentric anomaly from true anomaly.

\[ D = \tan \frac{\nu}{2} \]

**Parameters**

- `nu (Quantity)` – True anomaly.

**Returns**

- `D` – Hyperbolic eccentric anomaly.

**Return type** `Quantity`

**Note:** 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.core.angles.nu_to_E**

Eccentric anomaly from true anomaly.

New in version 0.4.0.

\[ E = 2 \arctan \sqrt{\frac{1 - e}{1 + e}} \tan \frac{\nu}{2} \]

**Parameters**

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

**Returns**

- `E` – Eccentric anomaly.
Return type Quantity

poliastro.core.angles.nu_to_F
Hyperbolic eccentric anomaly from true anomaly.
\[ F = \ln \left( \frac{\sin(\nu)\sqrt{e^2 - 1 + \cos(\nu) + e}}{1 + e \cos(\nu)} \right) \]

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.core.angles.E_to_nu
True anomaly from eccentric anomaly.
New in version 0.4.0.
\[ \nu = 2 \arctan \left( \sqrt{\frac{1 + e}{1 - e}} \tan \frac{E}{2} \right) \]

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

Returns nu – True anomaly.

Return type Quantity

poliastro.core.angles.F_to_nu
True anomaly from hyperbolic eccentric anomaly.

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

Returns nu – True anomaly.

Return type Quantity

poliastro.core.angles.M_to_E
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.core.angles.M_to_F
Hyperbolic eccentric anomaly from mean anomaly.

Parameters

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

Returns F – Hyperbolic eccentric anomaly.

Return type Quantity

poliastro.core.angles.M_to_D
Parabolic eccentric anomaly from mean anomaly.

Parameters

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

Returns D – Parabolic eccentric anomaly.

Return type Quantity

poliastro.core.angles.E_to_M
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.core.angles.F_to_M
Mean anomaly from eccentric anomaly.

Parameters

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

Returns M – Mean anomaly.

Return type Quantity

poliastro.core.angles.D_to_M
Mean anomaly from eccentric anomaly.

Parameters

• D (Quantity) – Parabolic eccentric anomaly.
• ecc (Quantity) – Eccentricity.

Returns M – Mean anomaly.

Return type Quantity
poliastro.core.angles.M_to_nu

True anomaly from mean anomaly.

New in version 0.4.0.

Parameters

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

Returns \( \nu \) – True anomaly.

Return type \( \text{Quantity} \)

Examples

```python
>>> from numpy import radians, degrees
>>> degrees(M_to_nu(radians(30.0), 0.06))
33.67328493021166
```

poliastro.core.angles.nu_to_M

Mean anomaly from true anomaly.

New in version 0.4.0.

Parameters

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

Returns \( M \) – Mean anomaly.

Return type \( \text{Quantity} \)

poliastro.core.angles.fp_angle

Returns the flight path angle.

\[
\gamma = \arctan \left( \frac{e \sin \theta}{1 + e \cos \theta} \right)
\]

Parameters

- \( \nu (\text{Quantity}) \) – True anomaly
- \( \text{ecc} (\text{Quantity}) \) – Eccentricity
- \( \text{fp_angle (Quantity)} \) – Flight path angle

Note: Algorithm taken from Vallado 2007, pp. 113.
The `poliastro.core.elements` module contains a set of functions related to orbital elements conversion. This module contains a set of functions that can be used to convert between different elements that define the orbit of a body.

`poliastro.core.elements.rv_pqw`

Returns \( \vec{r} \) and \( \vec{v} \) vectors in perifocal frame.

\[
\vec{r} = \frac{h^2}{\mu} \frac{1}{1 + e \cos(\theta)} \begin{bmatrix} \cos(\theta) \\ \sin(\theta) \\ 0 \end{bmatrix}
\]

\[
\vec{v} = \frac{h^2}{\mu} \begin{bmatrix} -\sin(\theta) \\ e + \cos(\theta) \\ 0 \end{bmatrix}
\]

**Parameters**

- \( k \) (float) – Standard gravitational parameter (km\(^3\) / s\(^2\)).
- \( p \) (float) – Semi-latus rectum or parameter (km).
- \( ecc \) (float) – Eccentricity.
- \( nu \) (float) – True anomaly (rad).

**Returns**

- \( r \) (ndarray) – Position. Dimension 3 vector
- \( v \) (ndarray) – Velocity. Dimension 3 vector
Examples

```python
>>> from poliastro.core.elements import rv_pqw
>>> from poliastro.constants import GM_earth

k = GM_earth # Earth gravitational parameter

ecc = 0.3 # Eccentricity
h = 60000e6 # Angular momentum of the orbit [m^2]/[s]
nu = np.deg2rad(120) # True Anomaly [rad]
p = h**2 / k # Parameter of the orbit
r, v = rv_pqw(k, p, ecc, nu)

# Printing the results
r = [-5312706.25105345 9201877.15251336 0] [m]
v = [-5753.30180931 -1328.66813933 0] [m]/[s]
```

Note: These formulas can be checked at Curtis 3rd Edition, page 110. Also the example proposed is 2.11 of Curtis 3rd Edition book.

**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].

The modified equinoctial orbital elements are a set of orbital elements that are useful for trajectory analysis and optimization. They are valid for circular, elliptic, and hyperbolic orbits. These direct modified equinoctial equations exhibit no singularity for zero eccentricity and orbital inclinations equal to 0 and 90 degrees. However, two of the components are singular for an orbital inclination of 180 degrees.

\[
p = a(1 - e^2) \quad (2.1)
\]

\[
f = e \cos(\omega + 2\Omega) \quad (2.2)
\]

\[
g = e \sin(\omega + 2\Omega) \quad (2.3)
\]

\[
h = \tan\left(\frac{i}{2}\right) \cos(2\Omega) \quad (2.4)
\]

\[
k = \tan\left(\frac{i}{2}\right) \sin(2\Omega) \quad (2.5)
\]

\[
L = \Omega + \omega + \theta \quad (2.6)
\]

2.6. Core level API
Parameters

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

1. First the angular momentum is computed:

   \[ \vec{h} = \vec{r} \times \vec{v} \]

2. With it the eccentricity can be solved:

   \[
   \begin{align*}
   \vec{e} &= \frac{1}{\mu} \left[ (v^2 - \frac{\mu}{r}) \vec{r} - (\vec{r} \cdot \vec{v}) \vec{v} \right] \\
   e &= \sqrt{\vec{e} \cdot \vec{e}}
   \end{align*}
   \]

3. The node vector line is solved:

   \[ \vec{N} = \vec{k} \times \vec{h} \]

   \[ N = \sqrt{\vec{N} \cdot \vec{N}} \]

4. The right ascension node is computed:

   \[
   \Omega = \begin{cases} 
   \cos^{-1} \left( \frac{N_x}{N} \right) & \text{if } N_y \geq 0 \\
   360^\circ - \cos^{-1} \left( \frac{N_x}{N} \right) & \text{if } N_y < 0
   \end{cases}
   \]

5. The argument of perigee:

   \[
   \omega = \begin{cases} 
   \cos^{-1} \left( \frac{N_x}{N_e} \right) & \text{if } e_z \geq 0 \\
   360^\circ - \cos^{-1} \left( \frac{N_x}{N_e} \right) & \text{if } e_z < 0
   \end{cases}
   \]

6. And finally the true anomaly:

   \[
   \nu = \begin{cases} 
   \cos^{-1} \left( \frac{v_r}{v} \right) & \text{if } v_r \geq 0 \\
   360^\circ - \cos^{-1} \left( \frac{v_r}{v} \right) & \text{if } v_r < 0
   \end{cases}
   \]
Parameters

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

Returns

- \( p \) (float) – Semi-latus rectum of parameter (km)
- \( \text{ecc} \) (float) – Eccentricity
- \( \text{inc} \) (float) – Inclination (rad)
- \( \text{raan} \) (float) – Right ascension of the ascending nod (rad)
- \( \text{argp} \) (float) – Argument of Perigee (rad)
- \( \text{nu} \) (float) – True Anomaly (rad)

Examples

```python
>>> from poliastro.core.elements import rv2coe
>>> from poliastro.constants import GM_earth
>>> from astropy import units as u
>>> import numpy as np

>>> k = GM_earth.to(u.km**3 / u.s**2) #Earth gravitational parameter
>>> r = [-6045, -3490, 2500] * u.km
>>> v = [-3.457, 6.618, 2.533] * u.km / u.s

>>> p, ecc, inc, raan, argp, nu = rv2coe(k, r, v)

>>> print("p: ", p, ", [km]"
    p: 8530.47436396927 [km]
>>> print("ecc: ", ecc)
    ecc: 0.17121118195416898
>>> print("inc: ", np.rad2deg(inc), ", [deg]"
    inc: 153.2492285182475 [deg]
>>> print("raan: ", np.rad2deg(raan), ", [deg]"
    raan: 255.27928533439618 [deg]
>>> print("argp: ", np.rad2deg(argp), ", [deg]"
    argp: 20.068139973005362 [deg]
>>> print("nu: ", np.rad2deg(nu), ", [deg]"
    nu: 28.445804984192108 [deg]
```

Note: This example is a real exercise from Orbital Mechanics for Engineering students by Howard D.Curtis. This exercise is 4.3 of 3rd. Edition, page 200.

```
poliastro.core.elements.mee2coe

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.
2.6.3 Hyper module

Utility hypergeometric functions.

```
poliastro.core.hyper.hyp2f1b
```

Hypergeometric function \( 2F1(3, 1, 5/2, x) \), see [Battin].

**Note:** More information about hypergeometric function can be checked at https://en.wikipedia.org/wiki/Hypergeometric_function

2.6.4 Initial Orbit Determination (IOD) module

The `poliastro.core.iod` deals with the problem of determining an orbit being given two position vectors along it and the that the body takes to travel from one to another.

This problem is known as Lambert’s problem and many algorithms have developed to solved since the main difficult of it is focused on numerical methods. This module contains two different functions that enable to solve the Lambert problem.

```
poliastro.core.iod.vallado
```

Solves the Lambert’s problem.

The algorithm returns the initial velocity vector and the final one, these are computed by the following expressions:

\[
\vec{v}_o = \frac{1}{g} (\vec{r} - f\vec{r}_0) \\
\vec{v} = \frac{1}{g} (\dot{\vec{r}} - \vec{r}_0)
\]

Therefore, the lagrange coefficients need to be computed. For the case of Lamber’s problem, they can be
expressed by terms of the initial and final vector:

\[ f = 1 - \frac{y}{r_o} \]  
\[ g = \sqrt{\frac{\mu}{y}} \]  
\[ \dot{g} = 1 - \frac{y}{r} \]  
\[ (2.13) \]

\[ (2.14) \]

\[ (2.15) \]

Where \( y(z) \) is a function that depends on the `poliastro.core.stumpff` coefficients:

\[ y = r_o + r + A \frac{zS(z) - 1}{\sqrt{C(z)}} \]

\[ A = \sin(\Delta \nu) \sqrt{\frac{rr_o}{1 - \cos(\Delta \nu)}} \]

The value of \( z \) to evaluate the stump functions is solved by applying a Numerical method to the following equation:

\[ z_{i+1} = z_i - \frac{F(z_i)}{F'(z_i)} \]

Function \( F(z) \) to the expression:

\[ F(z) = \left[ \frac{y(z)}{C(z)} \right]^{\frac{3}{2}} S(z) + A \sqrt{y(z)} - \sqrt{\mu} \Delta t \]

Parameters

- \( k \) (float) – Gravitational Parameter
- \( r_0 \) (array) – Initial position vector
- \( r \) (array) – Final position vector
- \( tof \) (~float) – Time of flight
- \( numiter \) (int) – Number of iterations to
- \( rtol \) (int) – Number of revolutions

Returns

- \( v_0 \) (~np.array) – Initial velocity vector
- \( v \) (~np.array) – Final velocity vector

Examples

```python
>>> from poliastro.core.iod import vallado
>>> from astropy import units as u
>>> import numpy as np
>>> from poliastro.bodies import Earth
>>> k = Earth.k.to(u.km**3 / u.s**2)
>>> r1 = np.array([5000, 10000, 2100])*u.km #Initial position vector
>>> r2 = np.array([-14600, 2500, 7000])*u.km #Final position vector
>>> tof = 3600*u.s #Time of flight
>>> v1, v2 = vallado(k.value, r1.value, r2.value, tof.value, short=True, numiter=35, rtol=1e-8)
```

(continues on next page)
>>> v1 = v1*u.km / u.s
>>> v2 = v2*u.km / u.s
>>> print(v1, v2)
[-5.99249499 1.92536673 3.24563805] km / s [-3.31245847 -4.196619 -0.38528907] km / s

Note: This procedure can be found in section 5.3 of Curtis, with all the theoretical description of the problem. Analytical example can be found in the same book under name Example 5.2.

poliastro.core.iod.izzo
Applies izzo algorithm to solve Lambert’s problem.

Parameters

- **k** *(float)* – Gravitational Constant
- **r1** *(array)* – Initial position vector
- **r2** *(array)* – Final position vector
- **tof** *(float)* – Time of flight between both positions
- **M** *(int)* – Number of revolutions
- **numiter** *(int)* – Number of iterations
- **rotl** *(float)* – Error tolerance

Returns

- **v1** *(~numpy.array)* – Initial velocity vector
- **v2** *(~numpy.array)* – Final velocity vector

2.6.5 Perturbations module

The *poliastro.core.perturbations* enables the user of *poliastro* to reproduce some perturbations, since certain conditions can not always be assumed ideal. This module enables to recreate those non-ideal conditions.

*poliastro.core.perturbations.J2_perturbation*
Calculates $J_2$ perturbation acceleration (km/s²)

$$
\vec{\rho} = \frac{3}{2} \frac{J_2 \mu R^2}{r^4} \left[ \frac{x}{r} \left( \frac{5 z^2 - 1}{r^2} \right) \hat{i} + \frac{y}{r} \left( \frac{5 y^2 - 1}{r^2} \right) \hat{j} + \frac{z}{r} \left( \frac{5 z^2 - 3}{r^2} \right) \hat{k} \right]
$$
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, \((\text{km}^3/\text{s}^2)\)
- **J2** (*float*) – oblateness factor
- **R** (*float*) – attractor radius

**Note:** 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, \((\text{km}^3/\text{s}^2)\)
- **J3** (*float*) – oblateness factor
- **R** (*float*) – attractor radius

**Note:** 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)

\[
\vec{p} = -\frac{1}{2} \rho v_{rel} \left( \frac{C_D A}{m} \right) \vec{v}_{rel}
\]

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, \((\text{km}^3/\text{s}^2)\)
- **C_D** (*float*) – dimensionless drag coefficient ()
- **A** (*float*) – frontal area of the spacecraft \((\text{km}^2)\)
- **m** (*float*) – mass of the spacecraft (kg)
- **H0** (*float*) – atmospheric scale height, (km)
- **rho0** (*float*) – the exponent density pre-factor, \((\text{kg} / \text{m}^3)\)

**Note:** This function provides the acceleration due to atmospheric drag. We follow Howard Curtis, section 12.4

the atmospheric density model is \(\rho(H) = \rho_0 \times \exp(-H / H_0)\)
**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)

\[ \vec{\ddot{p}} = \mu_m \left( \frac{\vec{r}_m/s}{r_m^3} - \frac{\vec{r}_m}{r_m^3} \right) \]

**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

**Note:** This formula is taken from Howard Curtis, section 12.10. As an example, a third body could be the gravity from the Moon acting on a small satellite.

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

\[ \vec{\ddot{p}} = -\nu \frac{S}{c} \left( \frac{C_R A}{m} \right) \frac{\vec{r}}{r} \]

**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 (km²)
- `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

**Note:** This function provides the acceleration due to star light pressure. We follow Howard Curtis, section 12.9
The main purpose of `poliastro.core.propagation` is to propagate a body along its orbit.

`poliastro.core.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.core.propagation.mean_motion`

Propagates orbit using mean motion. This algorithm depends on the geometric shape of the orbit.

For the case of the strong elliptic or strong hyperbolic orbits:

\[
M = M_0 + \frac{\mu^2}{r^3} \left(1 - e^2\right)^{\frac{3}{2}} t
\]

New in version 0.9.0.

**Parameters**

- `k (float)` – Standard gravitational parameter
- `r0 (Quantity)` – Initial position vector wrt attractor center.
- `v0 (Quantity)` – Initial velocity vector.
- `tof (float)` – Time of flight (s).
poliastro Documentation, Release 0.12.0

Note: 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.propagation.kepler
Solves Kepler’s Equation by applying a Newton-Raphson method.

If the position of a body along its orbit wants to be computed for a specific time, it can be solved by terms of the Kepler’s Equation:

\[
E = M + e \sin E
\]

In this case, the equation is written in terms of the Universal Anomaly:

\[
\sqrt{\mu} \Delta t = \frac{r_o v_o}{\sqrt{\mu}} \chi^2 C(\alpha \chi^2) + (1 - \alpha r_o) \chi^3 S(\alpha \chi^2) + r_o \chi
\]

This equation is solved for the universal anomaly by applying a Newton-Raphson numerical method. Once it is solved, the Lagrange coefficients are returned:

\[
f = 1 \frac{\chi^2}{r_o} C(\alpha \chi^2) \quad (2.17)
\]
\[
g = \Delta t - \frac{1}{\sqrt{\mu}} \chi^3 S(\alpha \chi^2) \quad (2.18)
\]
\[
f' = \frac{\sqrt{\mu}}{r v_o} (\alpha \chi^3 S(\alpha \chi^2) \quad (2.19)
\]
\[
g' = 1 - \frac{\chi^2}{r} C(\alpha \chi^2) \quad (2.20)
\]

(2.21)

Lagrange coefficients can be related then with the position and velocity vectors:

\[
\vec{r}' = f \vec{r}_o + g \vec{v}_o \quad (2.22)
\]
\[
\vec{v}' = f' \vec{r}_o + g' \vec{v}_o \quad (2.23)
\]

Parameters
- \( k (\text{float}) \) – Standard gravitational parameter
- \( r_0 (\text{array}) \) – Initial position vector
- \( v_0 (\text{array}) \) – Initial velocity vector
- \( \text{numiter} (\text{int}) \) – Number of iterations

Returns
- \( f (\text{float}) \) – First Lagrange coefficient
- \( g (\text{float}) \) – Second Lagrange coefficient
- \( fdot (\text{float}) \) – Derivative of the first coefficient
- \( gdot (\text{float}) \) – Derivative of the second coefficient

Note: The theoretical procedure is explained in section 3.7 of Curtis in really deep detail. For analytical example, check in the same book for example 3.6.
2.6.7 Stumpff module

The Stumpff functions, developed by Karl Stumpff, are used for analyzing orbits using the universal variable formulation.

**poliastro.core.stumpff.c2**
Second Stumpff function.
For positive arguments:

\[
e_2(\psi) = \frac{1 - \cos \sqrt{\psi}}{\psi}
\]

**poliastro.core.stumpff.c3**
Third Stumpff function.
For positive arguments:

\[
e_3(\psi) = \frac{\sqrt{\psi} - \sin \sqrt{\psi}}{\sqrt{\psi^3}}
\]
2.6.8 Util module

The `poliastro.core.util` module contains, mainly, a set of functions related to vectorial computations.

`poliastro.core.util.circular_velocity`

Compute circular velocity for a given body given the gravitational parameter and the semimajor axis.

\[ v = \sqrt{\frac{\mu}{a}} \]

**Parameters**
- `k` *(float)* – Gravitational Parameter
- `a` *(float)* – Semimajor Axis

`poliastro.core.util.rotate`

Rotates a vector around axis x, y or z a Counter ClockWise angle.

\[ R(\theta) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \]

**Parameters**
- `vec` *(ndarray)* – Dimension 3 vector.
- `angle` *(float)* – Angle of rotation (rad).
- `axis` *(int)* – Axis to be rotated (X-axis: 0, Y-axis: 1, Z-axis: 2)

**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)` or use the convenience function `transform`.

**References**

http://en.wikipedia.org/wiki/Rotation_matrix#Ambiguities
poliastro.core.util.transform
Rotates a coordinate system around axis a positive right-handed angle.

**Note:** 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
Returns the norm of a 3 dimension vector.

\[ ||\vec{v}|| = \sqrt{\sum_{i=1}^{n} v_i^2} \]

**Parameters**
vec (ndarray) – Dimension 3 vector.

**Examples**

```python
>>> from poliastro.core.util import norm
>>> from astropy import units as u

>>> vec = [1, 1, 1] * u.m
>>> norm(vec)
1.7320508075688772
```

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

\[ \vec{w} = \vec{u} \times \vec{v} = \begin{vmatrix}
  \hat{i} & \hat{j} & \hat{k} \\
u_y & v_y & u_z \\
u_x & v_x & u_z
\end{vmatrix} \]

**Parameters**

- a (ndarray) – 3 Dimension vector.
- b (ndarray) – 3 Dimension vector.

**Examples**

```python
>>> from poliastro.core.util import cross
>>> from astropy import units as u

>>> i = [1, 0, 0] * u.m
>>> j = [0, 1, 0] * u.m
>>> cross(i, j)
array([0., 0., 1.])
```

**Note:** np.cross is not supported in numba nopython mode, see https://github.com/numba/numba/issues/2978

## 2.7 What’s new

### 2.7.1 poliastro 0.12.0 - 2019-02-21

This major release brings lots of new features, several breaking changes that improve the overall consistency of the library, and a stronger bet on Plotly as the default plotting backend, as well as the usual bug fixes. This has been the
biggest release in terms of contributors so far and we feel we are reaching a tipping point, which makes us extremely proud and also busier!

### Highlights

- **New defaults for plotting:** We are now switching to Plotly for the default plotting backend as it has better interactive capabilities in the notebook, while keeping the matplotlib backend for publication-quality, 2D static plots. There might be some rough edges in the installation or in trying to keep the aspect ratio still, so we ask for user feedback.

- **Reorganization of propagation capabilities:** We made some changes to the propagation APIs to be more coherent and flexible and simpler to understand for new contributors. We removed some features from `sample()` to keep it simpler while moving some of them to `poliastro.twobody.propagation.propagate()`, and we splitted `propagate()` by adding `propagate_to_anomaly()`. At the cost of some breakage, we think this is a positive change that will make the library more maintainable in the future and reduce the number of bugs.

- **Better integration with reference frames:** We took one step further in our endeavor to integrate better with Astropy reference frames by adding a `from_coords()` method that accepts any frame, be it inertial or not.

- **Refactor of Orbit objects:** The `Orbit` was designed a long time ago and some design choices prevented all its orbital properties to appear in the documentation, while also making people think that they had to use an internal property. After a simple refactor this is no longer the case, and the code is still fast while being much simpler to understand. Did you know that you can compute the semilatus rectum, the modified equinoctial elements, the eccentricity vector or the mean motion of an `Orbit`? Now there are no excuses!

### New features

- **New orbit creation methods:** We can create an `Orbit` directly from JPL HORIZONS data using `from_horizons()`, from Astropy SkyCoord and BaseCoordinateFrame objects using `from_coords()`, and Geostationary orbits around an attractor using `geostationary()`. We plan to keep adding more in the coming releases.

- **New propagation methods:** We now have more specific methods for certain tasks, like `propagate_to_anomaly()` to propagate an `Orbit` to a certain anomaly, and we can specify the anomaly limits when using `sample()`.

- **New simple plotting method:** We added a `plot()` to quickly plot an `Orbit` without additional imports, in 2D or 3D.

- **Dark theme for Plotly plots:** It is now possible to create Plotly plots with a dark background, perfect for recreating our Solar System!

- **Computation of the Hill radius:** To complement the existing Laplace sphere of influence (or just Sphere of Influence) available with `poliastro.threebody.soi.laplace_radius()`, we added the Hill radius as well with the function `poliastro.threebody.soi.hill_radius()`.

- **Porkchop plots:** By popular demand, we can now produce gorgeous Porkchop plots to analyze launch opportunities between origin and destination bodies by using `poliastro.plotting.porkchop.porkchop()`. We plan to expand its capabilities by being able to target any body of the Solar System. Stay tuned!
Bugs fixed

- **Issue #435**: Orbit properties were not discoverable
- **Issue #469**: Better error for collinear points in Lambert problem
- **Issue #476**: Representation of orbits with no frame
- **Issue #477**: Propagator crashed when propagating a hyperbolic orbit 0 seconds
- **Issue #480**: OrbitPlotter2D did not have a `set_frame()` method
- **Issue #483**: OrbitPlotter2D `OrbitPlotter2D` results were not correct
- **Issue #518**: Trajectories were not redrawn when the frame was changed
• Issue #548: Improve installation instructions to include interactive and test dependencies
• Issue #573: Fix outdated matplotlib version limits

Backwards incompatible changes

• The old `OrbitPlotter` has been renamed to `poliastro.plotting.static.StaticOrbitPlotter`, please adjust your imports accordingly.
• `propagate()`, `sample()`, `poliastro.twobody.propagation.propagate()` and all propagators in `poliastro.twobody.propagation` now have different signatures, and the first two lost some functionality. Check out the notebooks and their respective documentation.
• The `poliastro.threebody` has been reorganized and some functions moved there.

Other updates

• We now follow the Black style guide
• The API docs are now more organized and should be easier to browse and understand.
• We are working towards documenting how to use poliastro in JupyterLab, please tell us about anything we may have missed.
• poliastro will be presented at the fifth PyCon Namibia

Contributors

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

• Juan Luis Cano
• Shreyas Bapat
• Jorge Martínez+
• Hrishikesh Goyal+
• Sahil Orionis+
• Helge Eichhorn+
• Antonina Geryak
• Aditya Vikram+

2.7.2 poliastro 0.11.1 - 2018-12-27

This release fixes some bugs found in 0.11.0 and prepares the ground for bigger API and code changes.

Bugs fixed

• Issue #281: Plotly graphs not showing in documentation
• Issue #469: OrbitPlotter.set_frame error
• Issue #476: Error when representing orbits with no reference frame
• Issue #482: Non deterministic legend layout
• Issue #492: Better error for collinear orbits in Lambert and corner case arithmetic

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

Contributors

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

• Juan Luis Cano
• Shreyas Bapat
• Ole Streicher+
• Antoniya Karpova+

2.7.3 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 dependencies 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 added 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+

2.7.4 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+

2.7.5 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.
2.7.6 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.

2.7.7 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.

2.7.8 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 access 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 Rodriguez 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.

2.7.9 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.

2.7.10 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 \(p\) instead of the semimajor axis \(a\) made in 0.4.0, so \(a\) must be used again. This change is definitive.

2.7.11 poliastro 0.4.2 - 2015-12-24

Fixed packaging problems.

2.7.12 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.

2.7.13 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.

2.7.14 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.

2.7.15 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.

2.7.16 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 References

Nanos gigantum humeris insidentes.

2.8.1 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:
2.8.2 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.


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.
p

poliastro.bodies, 83
poliastro.constants, 93
poliastro.coordinates, 93
poliastro.core.angles, 98
poliastro.core.elements, 102
poliastro.core.hyper, 106
poliastro.core.iod, 106
poliastro.core.perturbations, 108
poliastro.core.propagation, 111
poliastro.core.stumpff, 113
poliastro.core.util, 114
poliastro.examples, 94
poliastro.frames, 95
poliastro.iod.izzo, 92
poliastro.iod.vallado, 92
poliastro.maneuver, 96
poliastro.neos.dastcom5, 84
poliastro.neos.neows, 88
poliastro.plotting.core, 89
poliastro.plotting.misc, 90
poliastro.plotting.porkchop, 90
poliastro.plotting.static, 91
poliastro.plotting.util, 91
poliastro.threebody.flybys, 81
poliastro.threebody.restricted, 81
poliastro.threebody.soi, 82
poliastro.twobody.angles, 69
poliastro.twobody.decorators, 72
poliastro.twobody.orbit, 73
poliastro.twobody.propagation, 79
poliastro.util, 96
Index

A
a (poliastro.twobody.orbit.Orbit attribute), 74
apply_maneuver() (poliastro.twobody.orbit.Orbit method), 79
arglat (poliastro.twobody.orbit.Orbit attribute), 75
argp (poliastro.twobody.orbit.Orbit attribute), 74
asteroid_db() (in module poliastro.neos.dastcom5), 84
atmospheric_drag (in module poliastro.core.perturbations), 109
attractor (poliastro.twobody.orbit.Orbit attribute), 73

B
bielliptic() (poliastro.maneuver.Maneuver class method), 96
Body (class in poliastro.bodies), 84
body_centered_to_icrs() (in module poliastro.coordinates), 93

circular() (poliastro.twobody.orbit.Orbit class method), 76
circular_velocity (in module poliastro.core.util), 114
circular_velocity() (in module poliastro.util), 96
classical() (poliastro.twobody.orbit.Orbit method), 78
coe2mee (in module poliastro.core.elements), 103
coe2rv (in module poliastro.core.elements), 103
comet_db() (in module poliastro.neos.dastcom5), 84
compute_flyby() (in module poliastro.threebody.flybys), 81
cowell() (in module poliastro.twobody.propagation), 79
cross (in module poliastro.core.util), 115

download_dastcom5() (in module poliastro.neos.dastcom5), 85

e_to_M (in module poliastro.core.angles), 100
e_to_M() (in module poliastro.twobody.orbit), 71
e_to_nu (in module poliastro.core.angles), 99
e_to_nu() (in module poliastro.twobody.orbit), 70
e_vec (poliastro.twobody.orbit.Orbit attribute), 74
ecc (poliastro.twobody.orbit.Orbit attribute), 74
energy (poliastro.twobody.orbit.Orbit attribute), 74
etire_db() (in module poliastro.neos.dastcom5), 85
ePOCH (poliastro.twobody.orbit.Orbit attribute), 73

f (poliastro.twobody.orbit.Orbit attribute), 74
F_to_M (in module poliastro.core.angles), 100
F_to_M() (in module poliastro.twobody.orbit), 71
F_to_nu (in module poliastro.core.angles), 99
F_to_nu() (in module poliastro.twobody.orbit), 70
fp_angle (in module poliastro.core.angles), 101
fp_angle() (in module poliastro.twobody.orbit), 72
frame (poliastro.twobody.orbit.Orbit attribute), 74
from_body_ephem() (poliastro.twobody.orbit.Orbit class method), 76
from_classical() (poliastro.twobody.orbit.Orbit class method), 75
from_coords() (poliastro.twobody.orbit.Orbit class method), 75
from_equinoctial() (poliastro.twobody.orbit.Orbit class method), 76
from_horizons() (poliastro.twobody.orbit.Orbit class method), 76
from_vectors() (poliastro.twobody.orbit.Orbit class method), 75
func_twobody() (in module poliastro.core.propagation), 111

G

g (poliastro.twobody.orbit.Orbit attribute), 74
geostationary() (poliastro.twobody.orbit.Orbit class method), 77
get_frame() (in module poliastro.frames), 95
get_total_cost() (poliastro.maneuver.Maneuver method), 96
get_total_time() (poliastro.maneuver.Maneuver method), 96

H

h (poliastro.twobody.orbit.Orbit attribute), 74
h_vec (poliastro.twobody.orbit.Orbit attribute), 75
HCRS (class in poliastro.frames), 95
HeliocentricEclipticJ2000 (class in poliastro.frames), 95
hill_radius() (in module poliastro.threebody.soi), 83
hohmann() (poliastro.maneuver.Maneuver class method), 96
hyp2f1b (in module poliastro.core.hyper), 106
hyp_nu_limit() (in module poliastro.util), 97

I

icrs_to_body_centered() (in module poliastro.coordinates), 93
impulse() (poliastro.maneuver.Maneuver class method), 96
inc (poliastro.twobody.orbit.Orbit attribute), 74
inertial_body_centered_to_pqw() (in module poliastro.coordinates), 94
iss (in module poliastro.examples), 94
izzo (in module poliastro.core.iod), 108

J

J2_perturbation (in module poliastro.core.perturbations), 108
J3_perturbation (in module poliastro.core.perturbations), 109
JupiterICRS (class in poliastro.frames), 95

K

k (poliastro.twobody.orbit.Orbit attribute), 74
kepler() (in module poliastro.core.propagation), 112
kepler() (in module poliastro.twobody.propagation), 80

L

L (poliastro.twobody.orbit.Orbit attribute), 74
lagrange_points() (in module poliastro.threebody.restricted), 81
lagrange_points_vec() (in module poliastro.threebody.restricted), 82
lambert() (in module poliastro.iod.izzo), 92
lambert() (in module poliastro.iod.vallado), 92
laplace_radius() (in module poliastro.threebody.soi), 83

M

M_to_D (in module poliastro.core.angles), 100
M_to_D() (in module poliastro.twobody.angles), 71
M_to_E (in module poliastro.core.angles), 99
M_to_E() (in module poliastro.twobody.angles), 70
M_to_F (in module poliastro.core.angles), 100
M_to_F() (in module poliastro.twobody.angles), 70
M_to_nu (in module poliastro.core.angles), 100
M_to_nu() (in module poliastro.twobody.angles), 70
Maneuver (class in poliastro.maneuver), 96
MarsICRS (class in poliastro.frames), 95
mean_motion (in module poliastro.core.propagation), 111
mean_motion() (in module poliastro.twobody.propagation), 80
mee2coe (in module poliastro.core.elements), 105
MercuryICRS (class in poliastro.frames), 95
molniya (in module poliastro.examples), 94

N

n (poliastro.twobody.orbit.Orbit attribute), 74
NeptuneICRS (class in poliastro.frames), 95
norm (in module poliastro.core.util), 115
norm() (in module poliastro.util), 97
nu (poliastro.twobody.orbit.Orbit attribute), 74
nu_to_D (in module poliastro.core.angles), 98
nu_to_D() (in module poliastro.twörper.angles), 69
nu_to_E (in module poliastro.core.angles), 98
nu_to_E() (in module poliastro.twörper.angles), 69
nu_to_F (in module poliastro.core.angles), 99
nu_to_F() (in module poliastro.twörper.angles), 70
nu_to_M (in module poliastro.core.angles), 101
nu_to_M() (in module poliastro.twörper.angles), 72

O

Orbit (class in poliastro.twobody.orbit), 73
orbit_from_name() (in module poliastro.neos.dastcom5), 84
orbit_from_name() (in module poliastro.neos.neows), 88
orbit_from_record() (in module poliastro.neos.dastcom5), 84
orbit_from_spk_id() (in module poliastro.neos.neows), 88
OrbitPlotter2D (class in poliastro.plotting.core), 89
Index

Index 135

OrbitPlotter3D (class in poliastro.plotting.core), 89
OrbitSamplingWarning, 73

P
p (poliastro.twobody.orbit.Orbit attribute), 74
parabolic() (poliastro.twobody.orbit.Orbit class method), 77
period (poliastro.twobody.orbit.Orbit attribute), 74
plane (poliastro.twobody.orbit.Orbit attribute), 73
Planes (class in poliastro.frames), 95
plot() (poliastro.plotting.core.OrbitPlotter2D method), 89
plot() (poliastro.plotting.static.StaticOrbitPlotter method), 91
plot() (poliastro.twobody.orbit.Orbit method), 79
plot_solar_system() (in module poliastro.plotting.misc), 90
plot_trajectory() (tro.plotting.static.StaticOrbitPlotter method), 91
PlutoICRS (class in poliastro.frames), 95
poliastro.bodies (module), 83
poliastro.constants (module), 93
poliastro.coordinates (module), 93
poliastro.core.angles (module), 98
poliastro.core.elements (module), 102
poliastro.core.hyper (module), 106
poliastro.core.iod (module), 106
poliastro.core.perturbations (module), 108
poliastro.core.propagation (module), 111
poliastro.core.stumpff (module), 113
poliastro.core.util (module), 114
poliastro.examples (module), 94
poliastro.frames (module), 95
poliastro.iod.izzo (module), 92
poliastro.iod.vallado (module), 92
poliastro.maneuver (module), 96
poliastro.neos.dastcom5 (module), 84
poliastro.neos.neows (module), 88
poliastro.plotting.core (module), 89
poliastro.plotting.misc (module), 90
poliastro.plotting.porkchop (module), 90
poliastro.plotting.static (module), 91
poliastro.plotting.util (module), 91
poliastro.threebody.flybys (module), 81
poliastro.threebodyrestricted (module), 81
poliastro.threebody.soi (module), 82
poliastro.twobody.angles (module), 69
poliastro.twobody.decorators (module), 72
poliastro.twobody.orbit (module), 73
poliastro.twobody.propagation (module), 79
poliastro.util (module), 96
porkchop() (in module poliastro.plotting.porkchop), 90
pqw() (poliastro.twobody.orbit.Orbit method), 78
propagate() (in module poliastro.twobody.propagation), 81
propagate() (poliastro.twobody.orbit.Orbit method), 78
propagate_to_anomaly() (poliastro.twobody.orbit.Orbit method), 78

R
r (poliastro.twobody.orbit.Orbit attribute), 74
r_a (poliastro.twobody.orbit.Orbit attribute), 74
r_p (poliastro.twobody.orbit.Orbit attribute), 74
raan (poliastro.twobody.orbit.Orbit attribute), 74
radiation_pressure() (in module poliastro.core.perturbations), 110
read_headers() (in module poliastro.neos.dastcom5), 85
read_record() (in module poliastro.neos.dastcom5), 85
record_from_name() (in module poliastro.neos.dastcom5), 84
represent_as() (poliastro.twobody.orbit.Orbit method), 77
rotate (in module poliastro.core.util), 114
rotate() (in module poliastro.util), 96
rv() (poliastro.twobody.orbit.Orbit method), 78
rv2coe (in module poliastro.core.elements), 104
rv_pqw (in module poliastro.core.elements), 102

S
sample() (poliastro.twobody.orbit.Orbit method), 78
SaturnICRS (class in poliastro.frames), 95
set_attractor() (poliastro.plotting.static.StaticOrbitPlotter method), 91
set_frame() (poliastro.plotting.core.OrbitPlotter2D method), 89
set_frame() (poliastro.plotting.static.StaticOrbitPlotter method), 91
shadow_function (in module poliastro.core.perturbations), 109
soyuz_gto (in module poliastro.examples), 94
spk_id_from_name() (in module poliastro.neos.neows), 88
state_from_vector() (in module poliastro.twobody.decorators), 72
StaticOrbitPlotter (class in poliastro.plotting.static), 91
string_record_from_name() (in module poliastro.neos.dastcom5), 85
T
third_body() (in module poliastro.core.perturbations), 110
time_range() (in module poliastro.util), 97
TimeScaleWarning, 73
to_icrs() (poliastro.twobody.orbit.Orbit method), 78
transform (in module poliastro.core.util), 114
transform() (in module poliastro.coordinates), 94
transform() (in module poliastro.util), 97

U
UranusICRS (class in poliastro.frames), 95

V
v (poliastro.twobody.orbit.Orbit attribute), 74
vallado (in module poliastro.core.iod), 106
VenusICRS (class in poliastro.frames), 95