poliastro Documentation

Release 0.7.dev0

Juan Luis Cano Rodríguez

Nov 14, 2017

Contents

poliastro

Astrodynamics in Python

poliastro is an open source (MIT) collection of Python functions useful in Astrodynamics and Orbital Mechanics, focusing on interplanetary applications. It provides a simple and intuitive API and handles physical quantities with units. 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: poliastro works on recent versions of Python and is released under the MIT license, hence allowing commercial use of the library.

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

plot(molniya)
```

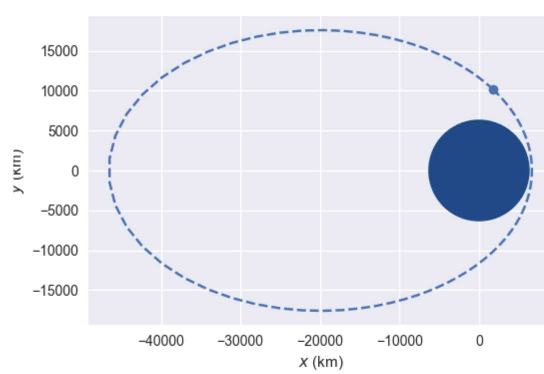


Fig. 1: Plot of a Molniya orbit around the Earth ($a = 26600\,\mathrm{km}, e = 0.75, i = 63.4 \mathrm{{}^{\circ}}$).

Contents

CHAPTER 1

Success stories

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

—Nayeli Ju (2017)

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

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

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

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

CHAPTER 2

Contents

2.1 About poliastro

2.1.1 Overview

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

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

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

2.1.2 History

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

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

2.1.3 Related software

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

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

- Skyfield: Another Astronomy Python package focused on computing observations of planetary bodies and Earth satellites written by Brandon Rhodes. It is the successor of pyephem, also written by him, but skyfield is a pure Python package and provides a much cleaner API.
- Plyades: A pioneering astrodynamics library written in Python by Helgee Eichhorn. Its clean and user friendly API inspired me to completely refactor poliastro 0.2 so it could be much easier to use. It has been stalled for a while, but at the moment of writing these lines its author is pushing new commits.
- orbital: Yet another orbital mechanics Python library written by Frazer McLean. It is very similar to poliastro (orbital plotting module was inspired in mine) but its internal structure is way smarter. It is more focused in plotting and it even provides 3D plots and animations.
- orekit-python-wrapper: According to its website, "The Orekit python wrapper enables to use Orekit within a normal python environment", using JCC. Orekit is a well-stablished, mature open source library for Astrodynamics written in Java strongly supported by several space agencies. The Python wrapper is developed by the Swedish Space Corporation.
- beyond: A young flight dynamics library written in Python with a focus on developing "a simple API for space observations". Some parts overlap with poliastro, but it also introduces many interesting features, and the examples look promising. Worth checking!
- SpiceyPy: This Python library wraps the SPICE Toolkit, a huge software collection developed by NASA which offers advanced astrodynamics functionality. Among all the wrappers available on the Internet, at the time of writing this is the most advanced and well-maintained one, although there are others.

2.1.4 Future ideas

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

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

2.1.5 Note of the original author

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

2.2 Getting started

2.2.1 Requirements

poliastro requires the following Python packages:

- NumPy, for basic numerical routines

- Astropy, for physical units and time handling
- numba (optional), for accelerating the code
- jplephem, for the planetary ephemerides using SPICE kernels
- matplotlib, for orbit plotting
- scipy, for root finding and numerical propagation
- pytest, for running the tests from the package

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

2.2.2 Installation

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

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

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

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

Alternative installation methods

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

```
$ pip install numpy  # Run this one first!
$ pip install poliastro
```

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

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

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

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

2.2.3 Testing

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

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

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

2.3 User guide

2.3.1 Defining the orbit: `Orbit` objects

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

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

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

```
# If using the Jupyter notebook, use %matplotlib inline
%matplotlib inline

import numpy as np
import matplotlib.pyplot as plt
from astropy import units as u

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

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

From position and velocity

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

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

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

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

- Defining vectorial physical quantities using Astropy units is very easy. The list is automatically converted to a `astropy.units.Quantity`, which is actually a subclass of NumPy arrays.
- If we display the orbit we just created, we get a string with the radius of pericenter, radius of apocenter, inclination and attractor:

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

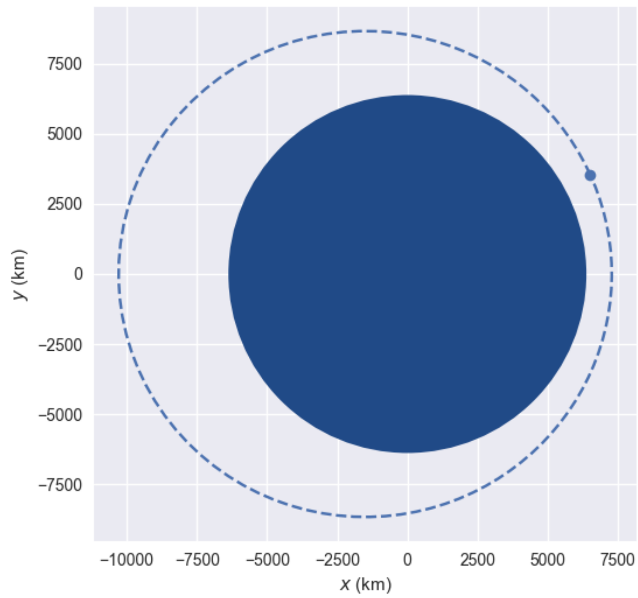
- If no time is specified, then a default value is assigned:

```
>>> ss.epoch
<Time object: scale='utc' format='jyear_str' value=J2000.000>
```

```
>>> ss.epoch.iso
'2000-01-01 12:00:00.000'
```

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

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



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

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

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

Warning: Be aware that, outside the Jupyter notebook (i.e. a normal Python interpreter or program) you might need to call `plt.show()` after the plotting commands or `plt.ion()` before them or they won't show. Check out the Matplotlib FAQ for more information.

From classical orbital elements

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

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

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

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

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

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

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

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

2.3.2 Moving forward in time: propagation

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

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

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

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

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

2.3.3 Changing the orbit: `Maneuver` objects

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

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

```
from poliastro.maneuver import Maneuver

dv = [5, 0, 0] * u.m / u.s

man = Maneuver.impulse(dv)
man = Maneuver((0 * u.s, dv))  # Equivalent
```

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

```
>>> ss_i = Orbit.circular(Earth, alt=700 * u.km)
>>> ss_i
7078 x 7078 km x 0.0 deg orbit around Earth ()
>>> hoh = Maneuver.hohmann(ss_i, 36000 * u.km)
>>> hoh.get_total_cost()
<Quantity 3.6173981270031357 km / s>
>>> hoh.get_total_time()
<Quantity 15729.741535747102 s>
```

You can also retrieve the individual vectorial impulses:

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

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

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

2.3.4 More advanced plotting: `OrbitPlotter` objects

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

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

```
from poliastro.plotting import OrbitPlotter

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

Which produces this beautiful plot:

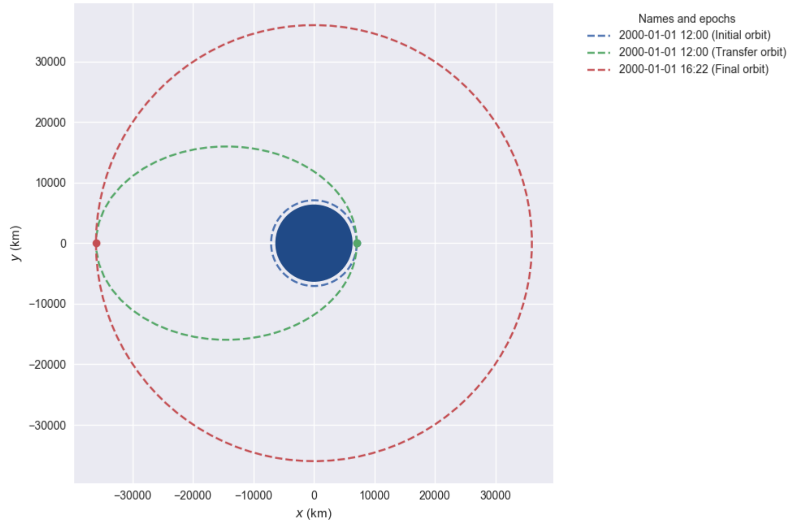


Fig. 2.1: Plot of a Hohmann transfer.

2.3.5 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 function `poliastro.ephem.get_body_ephem()` will return position and velocity vectors using low precision ephemerides available in Astropy and an `astropy.time.Time`:

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

And finally, retrieve the planet orbit:

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

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

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

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

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

2.3.6 Traveling through space: solving the Lambert problem

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

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

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

```
date_launch = time.Time('2011-11-26 15:02', scale='utc')
date_arrival = time.Time('2012-08-06 05:17', scale='utc')
tof = date_arrival - date_launch

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

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

And these are the results:

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

2.3.7 Working with NEOs

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

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

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

```
from poliastro.neos import neows

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

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

Per Python ad astra ;)

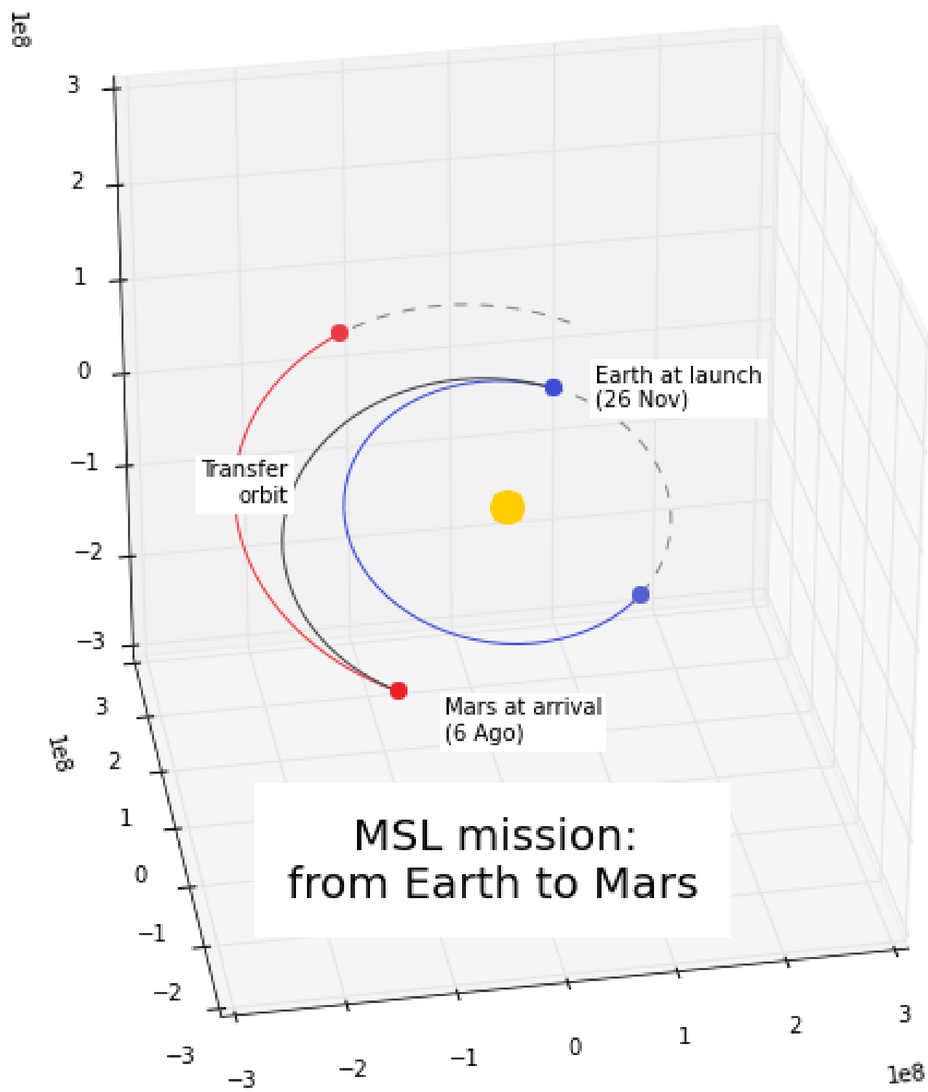


Fig. 2.2: Mars Science Laboratory orbit.

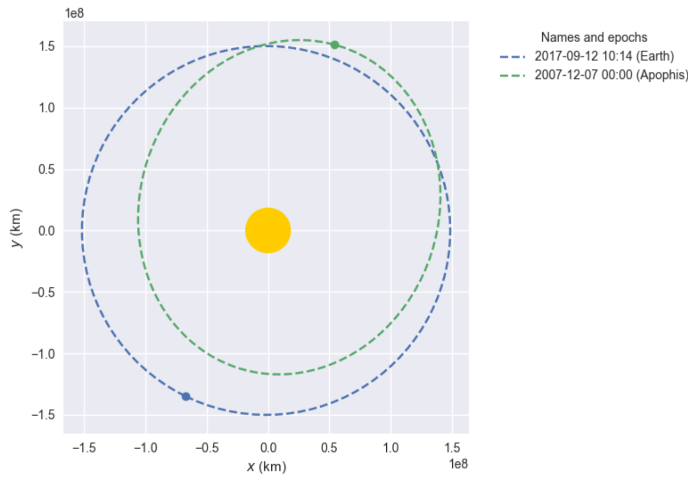
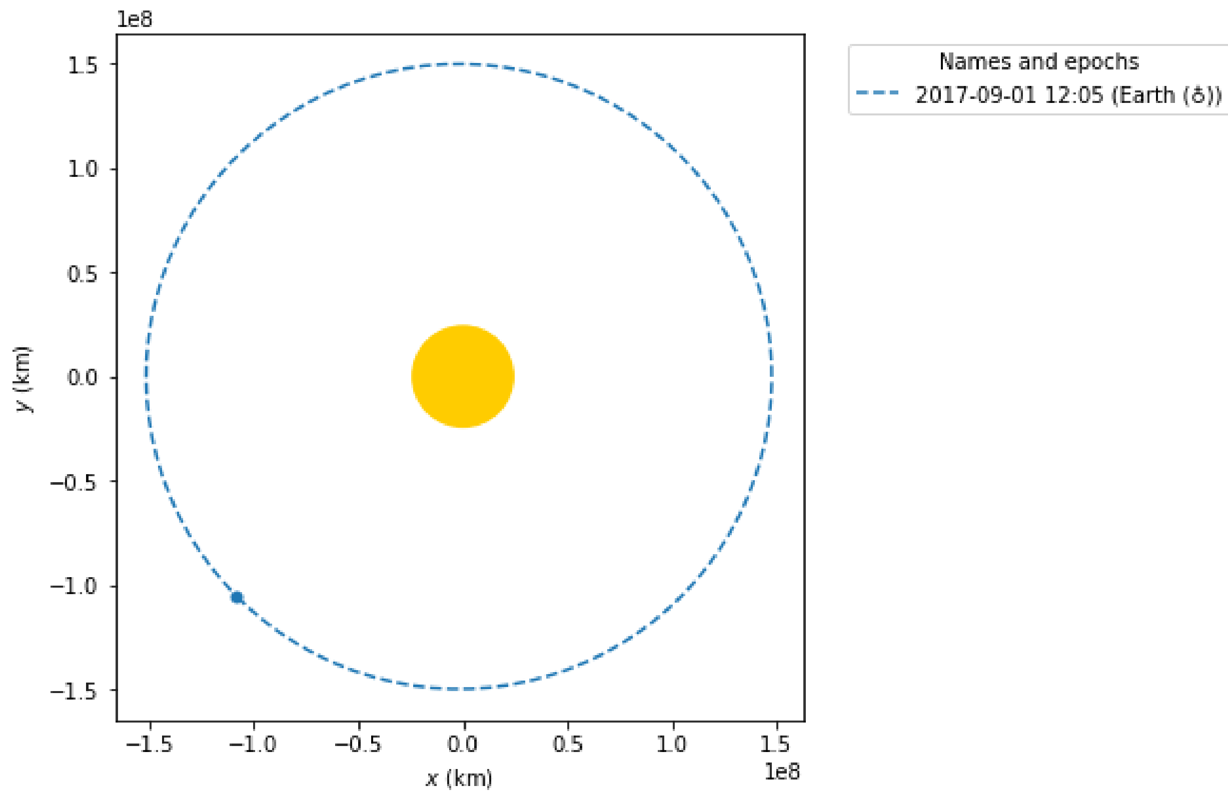


Fig. 2.3: Apophis asteroid orbit compared to Earth orbit.

2.4 Jupyter notebooks

2.4.1 Catch that asteroid!

```
In [1]: %matplotlib inline
        import matplotlib.pyplot as plt

        from astropy import units as u
        from astropy.time import Time
        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 OrbitPlotter, plot

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

In [2]: earth = Orbit.from_body_ephem(Earth, EPOCH)
        earth

Out[2]: 1 x 1 AU x 23.4 deg orbit around Sun ()

In [3]: plot(earth, label=Earth)

Out[3]: [<matplotlib.lines.Line2D at 0x7fad05cb76d8>,
         <matplotlib.lines.Line2D at 0x7fad05cb70b8>]
```




```
In [4]: from poliastro.neos import neows
```

```
In [5]: florence = neows.orbit_from_name("Florence")
        florence
```

```
Out[5]: 1 x 3 AU x 22.2 deg orbit around Sun ()
```

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

```
In [6]: florence.epoch
```

```
Out[6]: <Time object: scale='tdb' format='jd' value=2458000.5>
```

```
In [7]: florence.epoch.iso
```

```
Out[7]: '2017-09-04 00:00:00.000'
```

```
In [8]: florence.inc
```

22.15078 ° We first propagate:

```
In [9]: florence = florence.propagate(EPOCH)
        florence.epoch.tdb.iso
```

```
Out[9]: '2017-09-01 12:05:50.000'
```

And now we have to convert to another reference frame, using http://docs.astropy.org/en/stable/coordinates/.

```
In [10]: from astropy.coordinates import (
             ICRS, GCRS,
             CartesianRepresentation, CartesianDifferential
         )
         from poliastro.frames import HeliocentricEclipticJ2000
```

The NASA servers give the orbital elements of the asteroids in an Heliocentric Ecliptic frame. Fortunately, it is already defined in Astropy:

```
In [11]: florence_heclip = HeliocentricEclipticJ2000(
             x=florence.r[0], y=florence.r[1], z=florence.r[2],
             d_x=florence.v[0], d_y=florence.v[1], d_z=florence.v[2],
             representation=CartesianRepresentation,
             differential_cls=CartesianDifferential,
             obstime=EPOCH
         )
         florence_heclip
```

```
Out[11]: <HeliocentricEclipticJ2000 Coordinate (obstime=2017-09-01 12:05): (x, y, z) in km
             (  1.45904366e+08, -58569290.31320047,  2270778.95771309)
          (d_x, d_y, d_z) in km / s
             ( 7.40819577,  31.11060241,  12.80050223)>
```

Now we just have to convert to ICRS, which is the "standard" reference in which poliastro works:

```
In [12]: florence_icrs_trans = florence_heclip.transform_to(ICRS)
         florence_icrs_trans.representation = CartesianRepresentation
         florence_icrs_trans
```

```
Out[12]: <ICRS Coordinate: (x, y, z) in km
             (  1.46271269e+08, -53880006.88710973, -20906928.0521954)
          (v_x, v_y, v_z) in km / s
             ( 7.39978737,  23.46064313,  24.1234135)>
```

```
In [13]: florence_icrs = Orbit.from_vectors(
             Sun,
             r=[florence_icrs_trans.x, florence_icrs_trans.y, florence_icrs_trans.z] * u.km,
             v=[florence_icrs_trans.v_x, florence_icrs_trans.v_y, florence_icrs_trans.v_z] * (u.km /
             epoch=florence.epoch
         )
         florence_icrs
```

```
Out[13]: 1 x 3 AU x 44.6 deg orbit around Sun ()
```

```
In [14]: florence_icrs.rv()
```

```
Out[14]: (<Quantity [  1.46271269e+08, -5.38800069e+07, -2.09069281e+07] km>,
          <Quantity [  7.39978737, 23.46064313, 24.1234135 ] km / s>)
```

Let us compute the distance between Florence and the Earth:

```
In [15]: from poliastro.util import norm
```

```
In [16]: norm(florence_icrs.r - earth.r) - Earth.R
```

7060313.3 km This value is consistent with what ESA says! 7 060 160 km

```
In [17]: from IPython.display import HTML

         HTML(
         """<blockquote class="twitter-tweet" data-lang="en"><p lang="es" dir="ltr">La <a href="https
         <script async src="//platform.twitter.com/widgets.js" charset="utf-8"></script>"""
         )
```

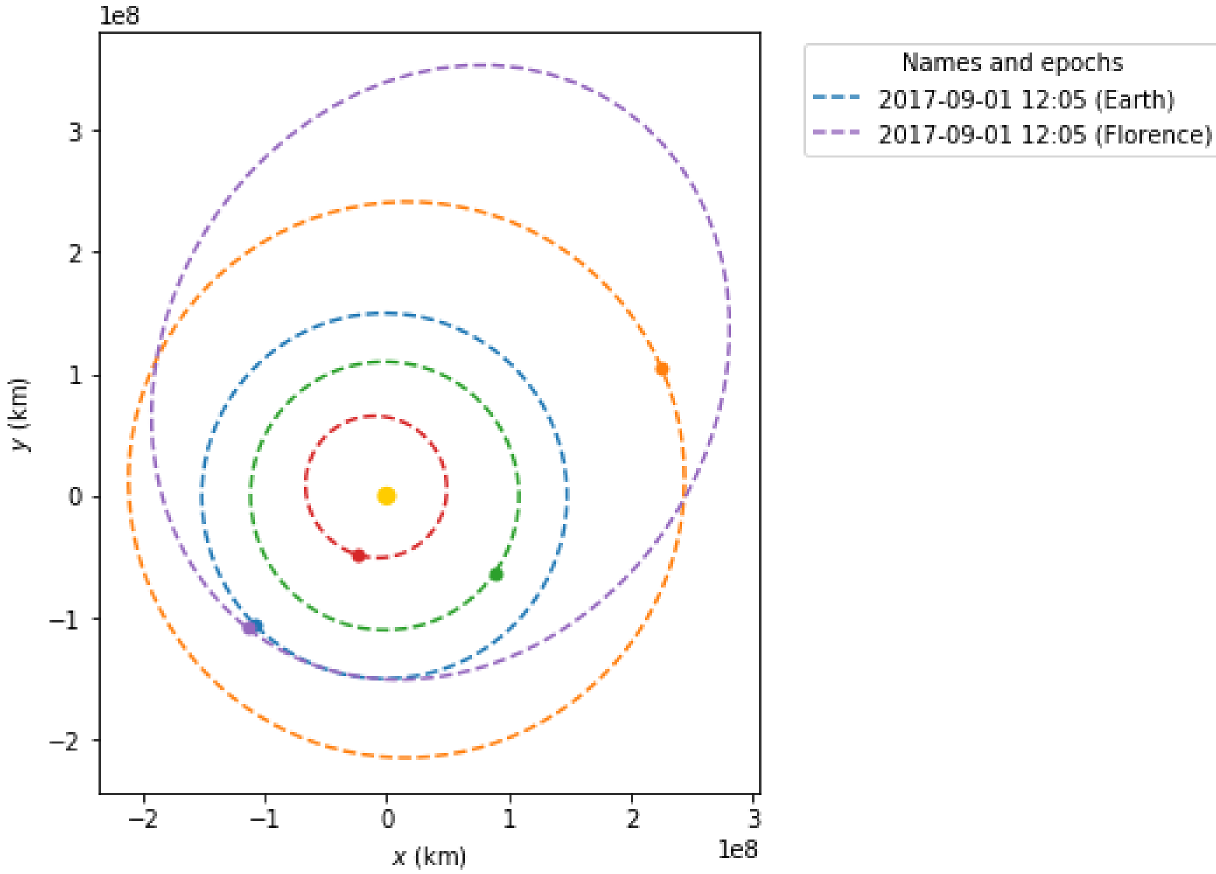
```
Out[17]: <IPython.core.display.HTML object>
```

And now we can plot!

```
In [18]: frame = OrbitPlotter()

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

         frame.plot(Orbit.from_body_ephem(Mars, EPOCH))
         frame.plot(Orbit.from_body_ephem(Venus, EPOCH))
         frame.plot(Orbit.from_body_ephem(Mercury, EPOCH))
```

```
        frame.plot(florence_icrs, label="Florence")
```

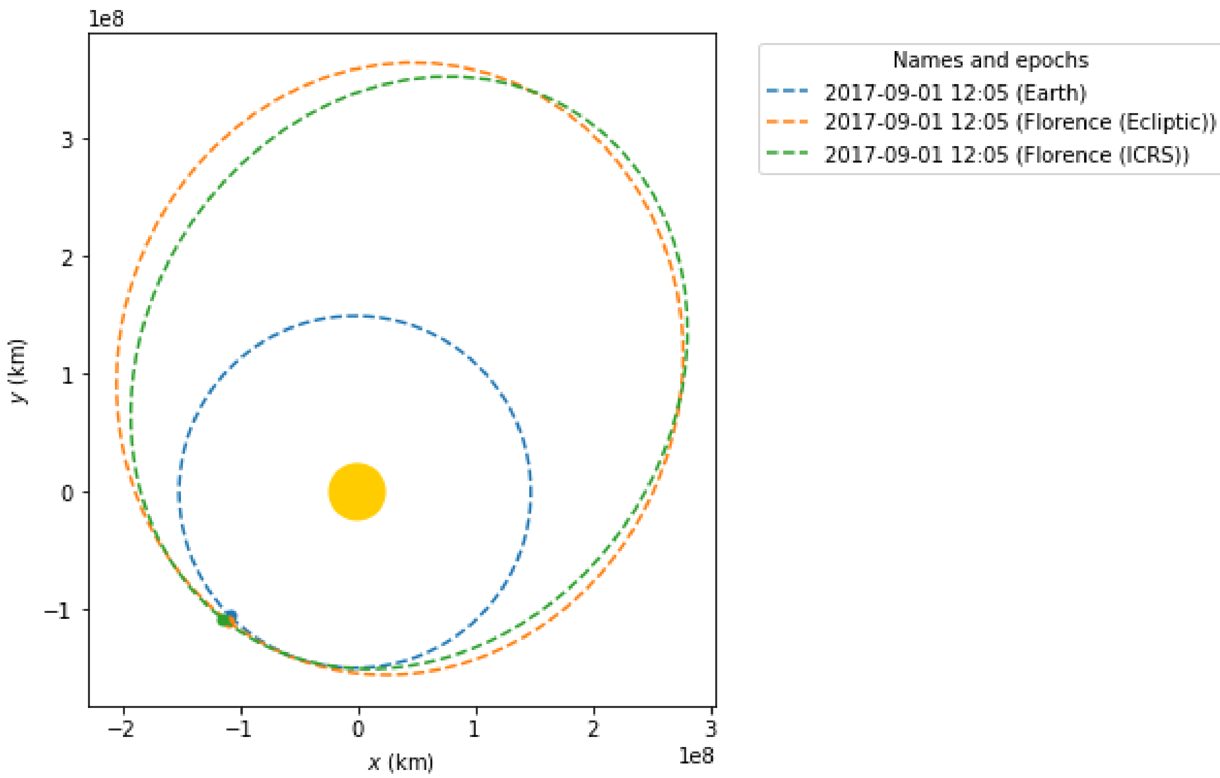
```
Out[18]: [<matplotlib.lines.Line2D at 0x7f3e2bf66320>,
          <matplotlib.lines.Line2D at 0x7f3e2bf66518>]
```



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

```
In [19]: frame = OrbitPlotter()

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

         frame.plot(florence, label="Florence (Ecliptic)")
         frame.plot(florence_icrs, label="Florence (ICRS)")
```

```
Out[19]: [<matplotlib.lines.Line2D at 0x7f3e2be71048>,
          <matplotlib.lines.Line2D at 0x7f3e2be71240>]
```



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:

```
In [20]: florence_gcrs_trans = florence_heclip.transform_to(GCRS(obstime=EPOCH))
         florence_gcrs_trans.representation = CartesianRepresentation
         florence_gcrs_trans
```

```
Out[20]: <GCRS Coordinate (obstime=2017-09-01 12:05, obsgeoloc=( 0.,  0.,  0.) m, obsgeovel=( 0.,  0
             ( 4966319.35958239, -5018473.35356456,  297867.61376881)
          (v_x, v_y, v_z) in km / s
             (-2.76873111, -1.96008601,  13.10279932)>
```

```
In [21]: florence_hyper = Orbit.from_vectors(
             Earth,
             r=[florence_gcrs_trans.x, florence_gcrs_trans.y, florence_gcrs_trans.z] * u.km,
             v=[florence_gcrs_trans.v_x, florence_gcrs_trans.v_y, florence_gcrs_trans.v_z] * (u.km /
             epoch=EPOCH
         )
         florence_hyper
```

```
Out[21]: 7066691 x -7071046 km x 104.3 deg orbit around Earth ()
```

Notice that the ephemerides of the Moon is also given in ICRS, and therefore yields a weird hyperbolic orbit!

```
In [22]: moon = Orbit.from_body_ephem(Moon, EPOCH)
         moon
```

```
Out[22]: 151218466 x -151219347 km x 23.3 deg orbit around Earth ()
```

```
In [23]: moon.a
```

−440.42131 km

```
In [24]: moon.ecc
```

343350.57 So we have to convert again.

```
In [25]: moon_icrs = ICRS(
             x=moon.r[0], y=moon.r[1], z=moon.r[2],
             v_x=moon.v[0], v_y=moon.v[1], v_z=moon.v[2],
             representation=CartesianRepresentation,
             differential_cls=CartesianDifferential
         )
         moon_icrs
```

```
Out[25]: <ICRS Coordinate: (x, y, z) in km
             (  1.41399531e+08, -49228391.42507221, -21337616.62766309)
          (v_x, v_y, v_z) in km / s
             ( 11.10890252,  25.6785744,  11.0567569)>
```

```
In [26]: moon_gcrs = moon_icrs.transform_to(GCRS(obstime=EPOCH))
         moon_gcrs.representation = CartesianRepresentation
         moon_gcrs
```

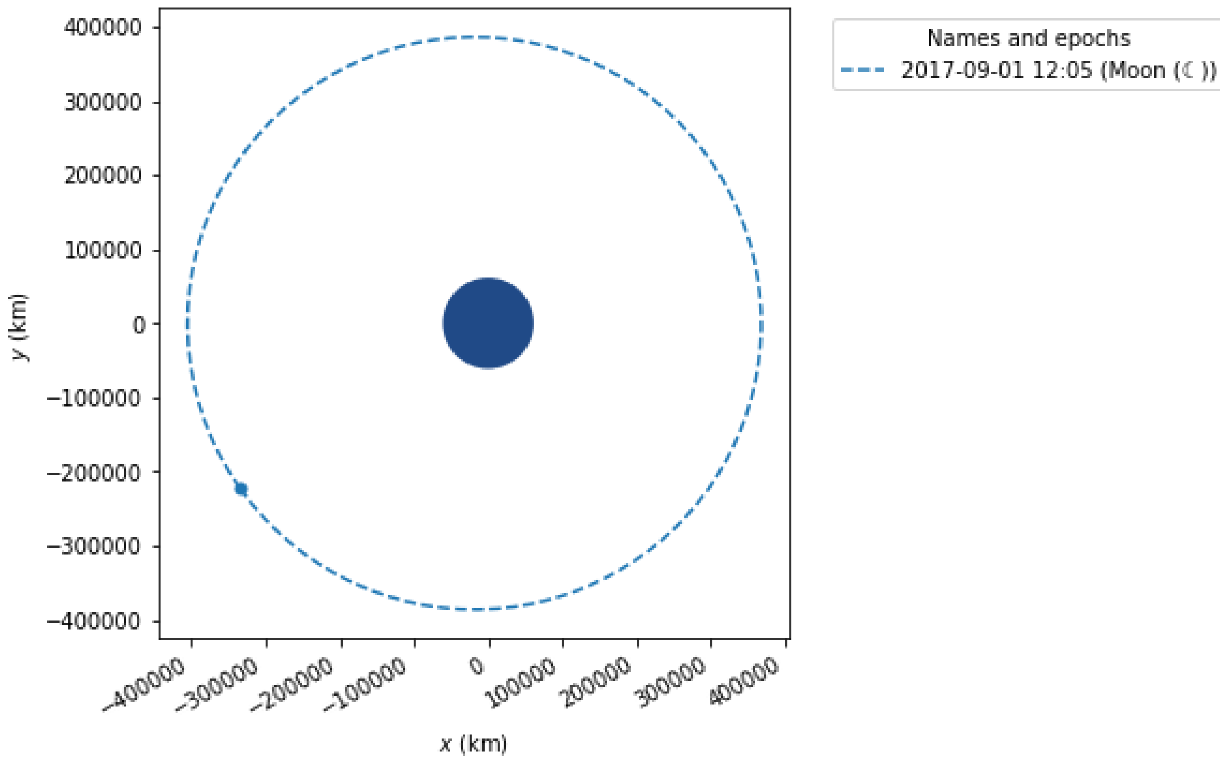
```
Out[26]: <GCRS Coordinate (obstime=2017-09-01 12:05, obsgeoloc=( 0.,  0.,  0.) m, obsgeovel=( 0.,  0
             ( 94189.90120828, -367278.24304992, -133087.21297573)
          (v_x, v_y, v_z) in km / s
             ( 0.94073662,  0.25786326,  0.03569047)>
```

```
In [27]: moon = Orbit.from_vectors(
             Earth,
             [moon_gcrs.x, moon_gcrs.y, moon_gcrs.z] * u.km,
             [moon_gcrs.v_x, moon_gcrs.v_y, moon_gcrs.v_z] * (u.km / u.s),
             epoch=EPOCH
         )
         moon
```

```
Out[27]: 367937 x 405209 km x 19.4 deg orbit around Earth ()
```

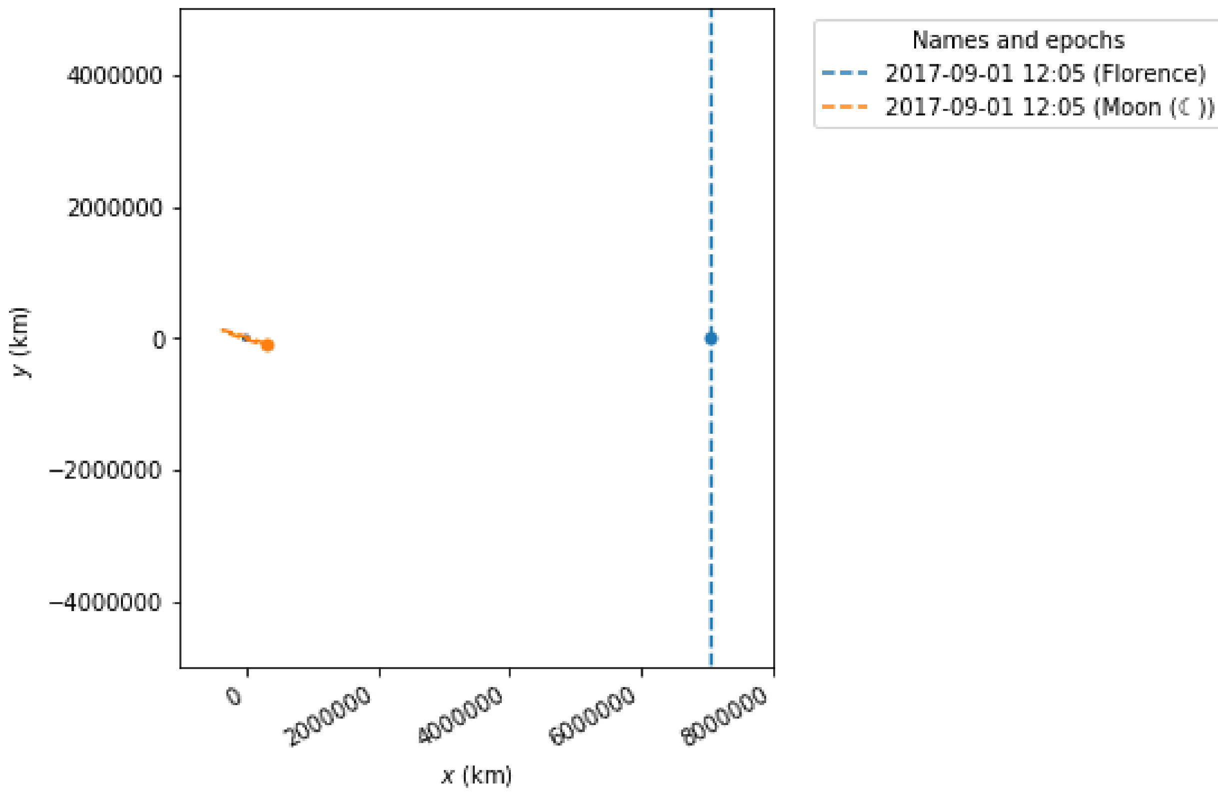
And finally, we plot the Moon:

```
In [28]: plot(moon, label=Moon)
         plt.gcf().autofmt_xdate()
```



And now for the final plot:

```
In [29]: frame = OrbitPlotter()

         # 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()
```



Per Python ad astra!

2.4.2 Comparing Hohmann and bielliptic transfers

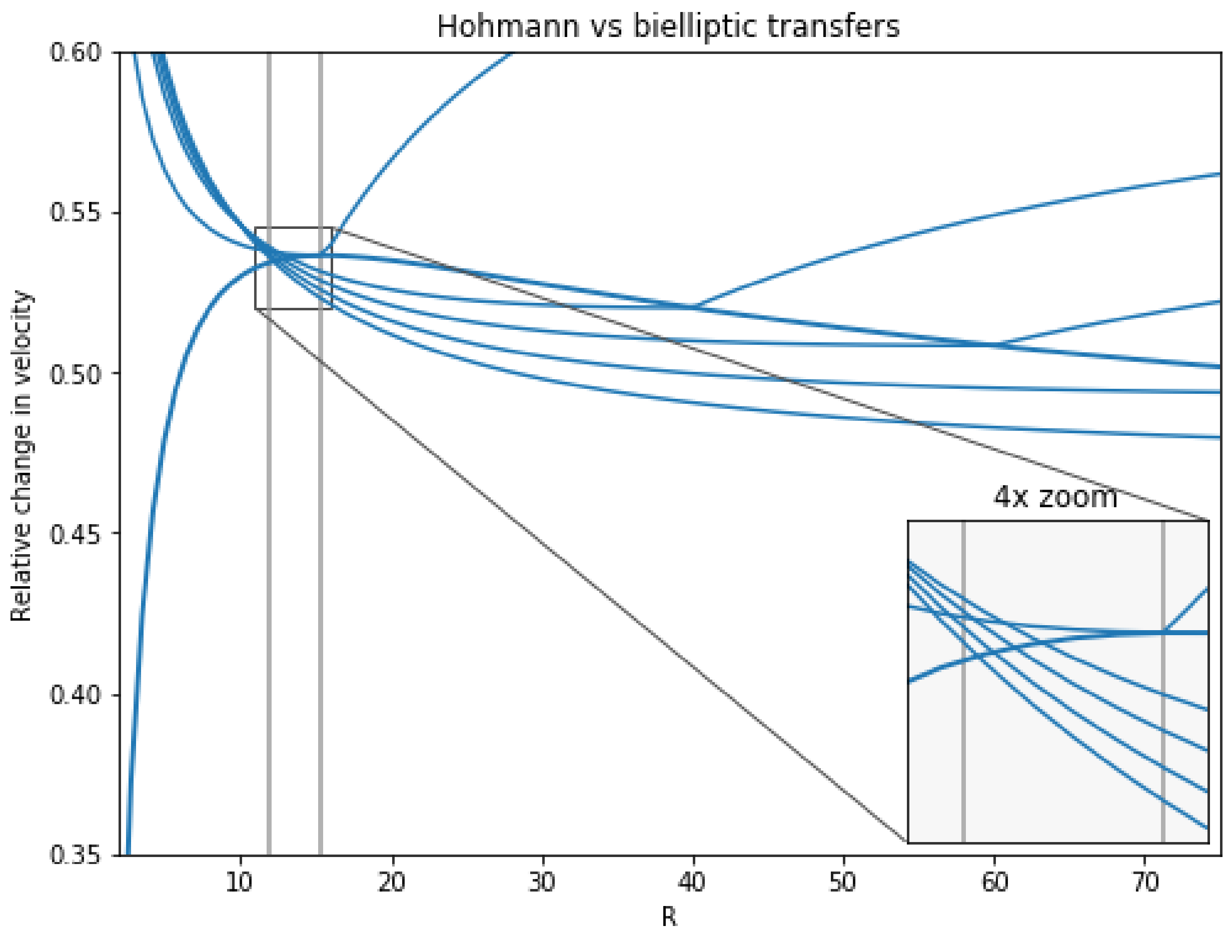
In [1]:
```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt

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.maneuver import Maneuver
```

In [2]:
```
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):
```

```
            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/Development/astropy/astropy/units/quantity.py:641: RuntimeWarning: invalid value encount
  *arrays, **kwargs)
```

```
In [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'})
            ax_zoom.plot(R, hohmann_data, lw=2)
            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")
```



2.4.3 New Horizons launch and trajectory

Main data source: Guo & Farquhar "New Horizons Mission Design" http://www.boulder.swri.edu/pkb/ssr/ssr-mission-design.pdf

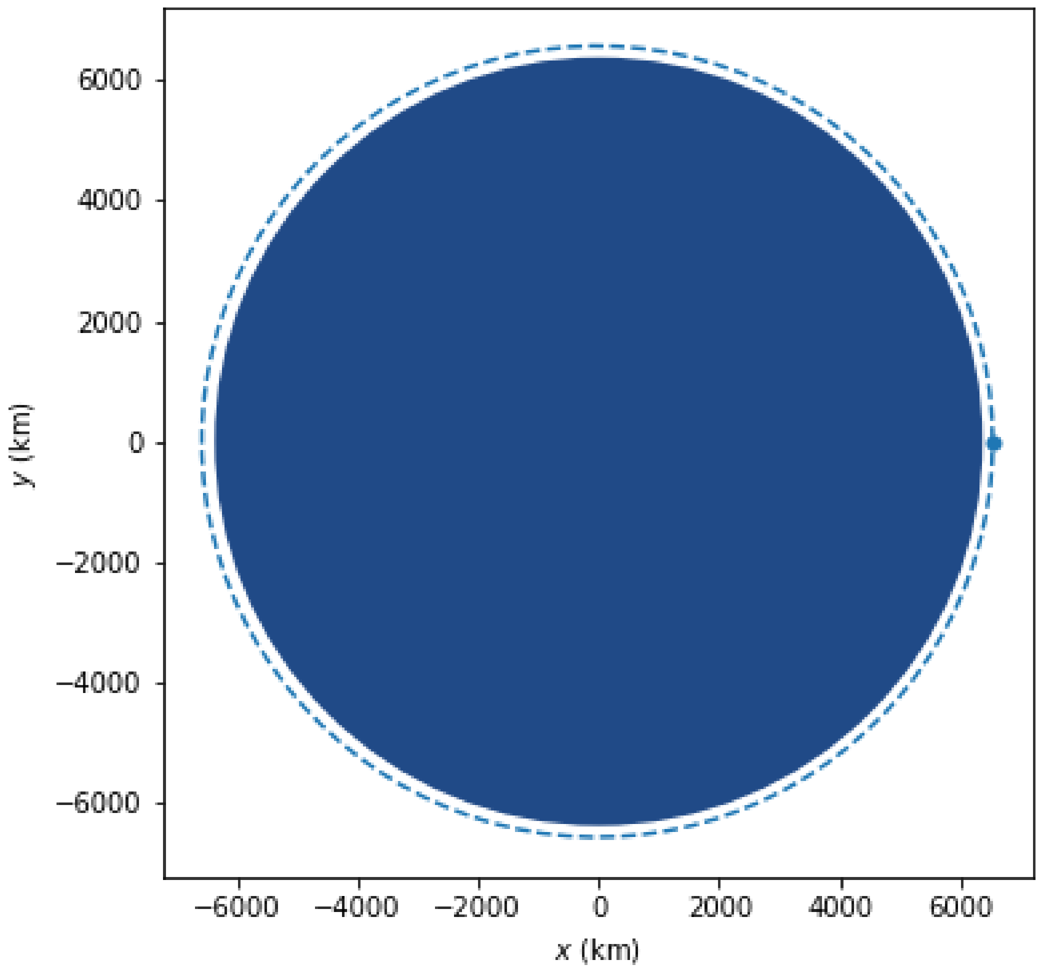
```
In [1]: %matplotlib inline
        import matplotlib
        import matplotlib.pyplot as plt

        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 plot, OrbitPlotter
        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]

In [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, 0 * u.deg,  # We don't mind
                               time.Time("2006-01-19", scale='utc'))

plot(parking)
parking.v
```



$[0,\ 7.8198936,\ 0]\ \frac{\mathrm{km}}{\mathrm{s}}$

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_{∞}:

$$v^2 = v_e^2 + v_{\infty}^2$$

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

```
In [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)
```

16.718069 $\frac{\mathrm{km}}{\mathrm{s}}$ Quoting "New Horizons Mission Design":

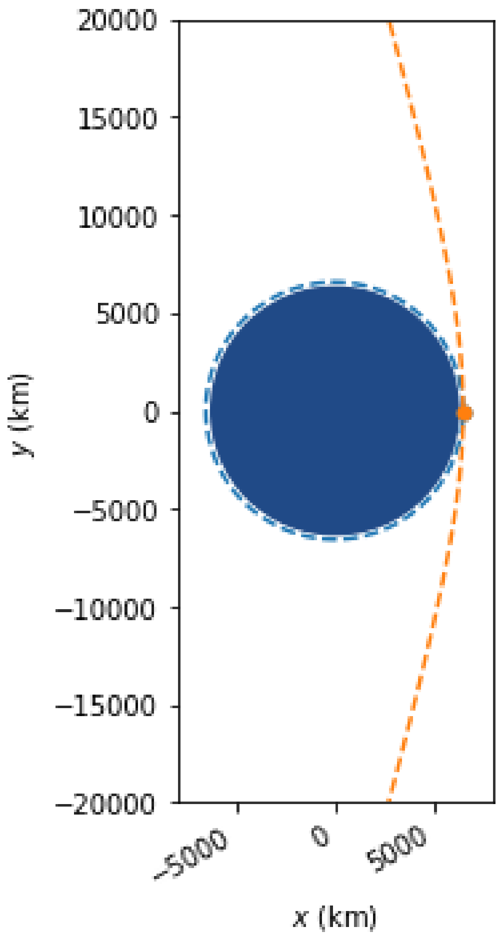
> After a short coast in the parking orbit, the spacecraft was then injected into the desired heliocentric orbit by the Centaur second stage and Star 48B third stage. At the Star 48B burnout, the New Horizons spacecraft reached the highest Earth departure speed, **estimated at 16.2 km/s**, becoming the fastest spacecraft ever launched from Earth. [Emphasis mine]

```
In [4]: v_estimated = 16.2 * u.km / u.s

        print("Relative error of {:.2f} %".format((norm(exit.v) - v_estimated) / v_estimated * 100))
```

```
Relative error of 3.20 %
```

So it stays within the same order of magnitude. Which is reasonable, because real life burns are not instantaneous.

```
In [5]: op = OrbitPlotter()

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

        plt.xlim(-8000, 8000)
        plt.ylim(-20000, 20000)
        plt.gcf().autofmt_xdate()
```



Option b): Compute v_∞ 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.

```
In [6]: 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)
```

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

```
In [7]: C_3_lambert = (norm(nh_v_0 - v_earth)).to(u.km / u.s)**2
        C_3_lambert
```

$158.45628 \frac{\mathrm{km}^2}{\mathrm{s}^2}$

```
In [8]: print("Relative error of {:.2f} %".format((C_3_lambert - C_3_A) / C_3_A * 100))
```

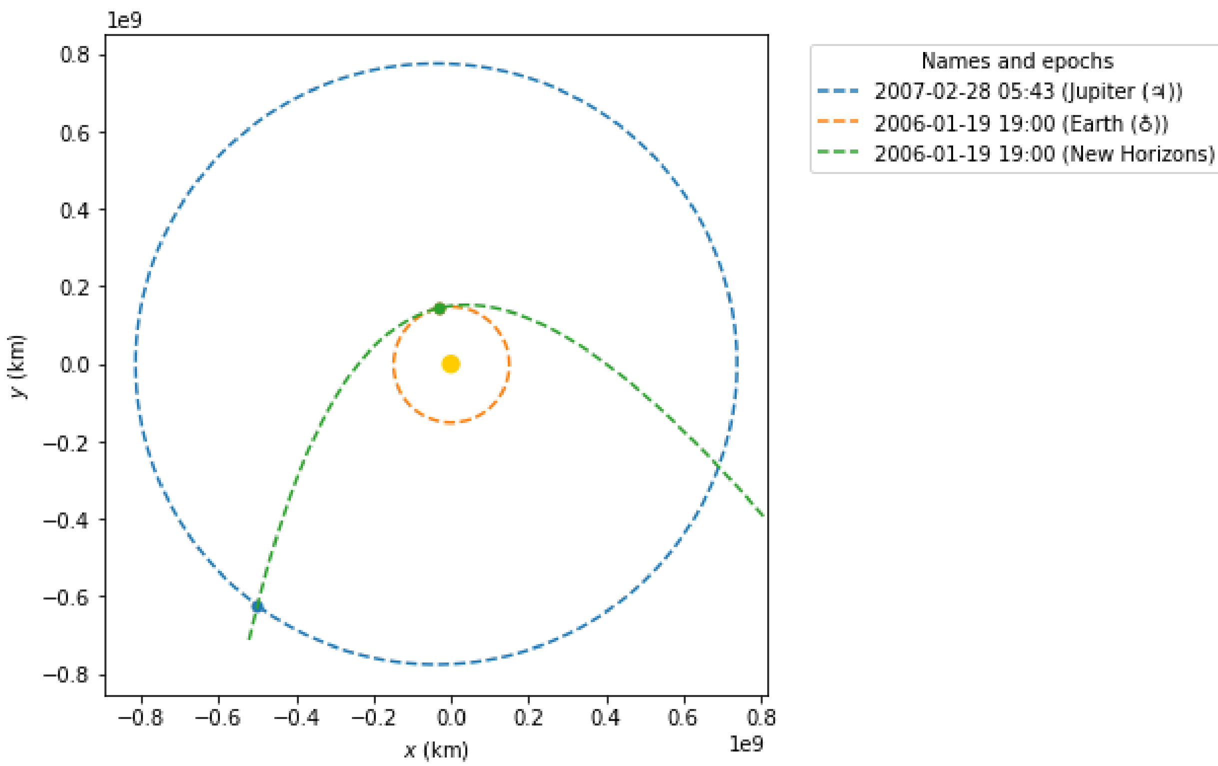
```
Relative error of 0.51 %
```

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

From Earth to Jupiter

```
In [9]: from poliastro.plotting import BODY_COLORS

        nh = Orbit.from_vectors(Sun, nh_r_0.to(u.km), nh_v_0.to(u.km / u.s), nh_date)

        op = OrbitPlotter(num_points=1000)

        op.plot(nh_jup, label=Jupiter)

        plt.gca().set_autoscale_on(False)

        op.plot(nh_earth, label=Earth)
        op.plot(nh, label="New Horizons")
```

```
Out[9]: [<matplotlib.lines.Line2D at 0x16562f13198>,
         <matplotlib.lines.Line2D at 0x16562f07e80>]
```



2.4.4 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.

```
In [1]: %matplotlib notebook
        import numpy as np
        import matplotlib.pyplot as plt
        from mpl_toolkits.mplot3d import Axes3D

        import astropy.units as u
        from astropy import time

        from poliastro import iod
        from poliastro.bodies import Sun
        from poliastro.twobody import Orbit
```

We need a binary file from NASA called *SPICE kernel* to compute the position and velocities of the planets. Astropy downloads it for us:

```
In [2]: from astropy.coordinates import solar_system_ephemeris
        from poliastro.ephem import get_body_ephem

        solar_system_ephemeris.set("jpl")

Out[2]: <ScienceState solar_system_ephemeris: '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.

```
In [3]: # 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)
```

$6086.2503\ \mathrm{h}$ With the date of launch and the date of landing we can compute the Julian days. The Julian day is an integer assigned to a date, and it's useful for not having to deal with leap years, changes of calendar and other messy stuff. It is measured from around 4713 BC so it is a pretty big number, as we'll see:

```
In [4]: # Calculate vector of times from launch and arrival Julian days
        dt = (date_arrival - date_launch) / N

        # Idea from http://docs.astropy.org/en/stable/time/#getting-started
        times_vector = date_launch + dt * np.arange(N + 1)
```

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

```
In [5]: rr_earth, vv_earth = get_body_ephem("earth", times_vector)

In [6]: rr_earth[:, 0]
```

$[64600643,\ 1.2142487 \times 10^{8},\ 52640047]\ \mathrm{km}$

```
In [7]: vv_earth[:, 0]
```

$[-2352414.3,\ 1032013.3,\ 447276.92]\ \mathrm{\frac{km}{d}}$

```
In [8]: rr_mars, vv_mars = get_body_ephem("mars", times_vector)
```

```
In [9]: rr_mars[:, 0]
```

$[-1.2314963 \times 10^{8},\ 1.9075251 \times 10^{8},\ 90809654]$ km

```
In [10]: vv_mars[:, 0]
```

$[-1730626.7,\ -811069.96,\ -325255.38]\ \frac{\mathrm{km}}{\mathrm{d}}$ 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!

```
In [11]: # Compute the transfer orbit!
         r0 = rr_earth[:, 0]
         rf = rr_mars[:, -1]

         (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)
```

The rest of the code is boilerplate we need for a beautiful plot: we retrieve all the intermediate positions of the transfer orbit, and compute some more vectors outside of the mission time frame to decorate the plot.

This code sucks. Pull requests welcome!

```
In [12]: # Extract whole orbit of Earth, Mars and transfer (for plotting)
         rr_trans = np.zeros_like(rr_earth)
         rr_trans[:, 0] = r0
         for ii in range(1, len(times_vector)):
             tof = (times_vector[ii] - times_vector[0]).to(u.day)
             rr_trans[:, ii] = ss0_trans.propagate(tof).r

In [13]: # Now compute the trail, better backwards
         date_final = date_arrival - 1 * u.year
         dt2 = (date_final - date_launch) / N

         times_rest_vector = date_launch + dt2 * np.arange(N + 1)
         rr_earth_rest, _ = get_body_ephem("earth", times_rest_vector)
         rr_mars_rest, _ = get_body_ephem("mars", times_rest_vector)
```

The positions are in the International Standard Reference Frame, which has the Equator as the fundamental plane

---

And finally, we can plot the figure! There is no more magic here, just passing the position vectors to matplotlib `plot` function and adding some style to the plot.

```
In [14]: # Plot figure
         # To add arrows:
         # https://github.com/matplotlib/matplotlib/blob/v2.0.0/lib/matplotlib/streamplot.py#L172-L1

         fig = plt.figure(figsize=(5, 5))
         ax = fig.add_subplot(111, projection='3d')

         def plot_body(ax, r, color, size, border=False, **kwargs):
             """Plots body in axes object.

             """
             return ax.plot(*r[:, None], marker='o', color=color, ms=size, mew=int(border), **kwargs)
```

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

        # Plotting orbits is easy!
        ax.plot(*rr_earth.to(u.km).value, c=color_earth0)
        ax.plot(*rr_mars.to(u.km).value, c=color_mars0)

        ax.plot(*rr_trans.to(u.km).value, c=color_trans)

        ax.plot(*rr_earth_rest.to(u.km).value, ls='--', c=color_orbit)
        ax.plot(*rr_mars_rest.to(u.km).value, ls='--', c=color_orbit)

        # But plotting planets feels even magical!
        plot_body(ax, np.zeros(3), color_sun, 16)

        plot_body(ax, r0.to(u.km).value, color_earth0, 8)
        plot_body(ax, rr_earth[:, -1].to(u.km).value, color_earthf, 8)

        plot_body(ax, rr_mars[:, 0].to(u.km).value, color_mars0, 8)
        plot_body(ax, rf.to(u.km).value, color_marsf, 8)

        # Add some text
        #ax.text(-0.75e8, -3.5e8, -1.5e8, "MSL mission:\nfrom Earth to Mars", size=20, ha='center',
        ax.text(r0[0].to(u.km).value * 1.4, r0[1].to(u.km).value * 0.4, r0[2].to(u.km).value * 1.25,
                "Earth at launch\n(26 Nov)", ha="left", va="bottom")#, backgroundcolor='#ffffff')
        ax.text(rf[0].to(u.km).value * 0.7, rf[1].to(u.km).value * 1.1, rf[2].to(u.km).value,
                "Mars at arrival\n(6 Ago)", ha="left", va="top")#, backgroundcolor='#ffffff')
        ax.text(-1.9e8, 8e7, 0, "Transfer\norbit", ha="right", va="center")#, backgroundcolor='#fff

        # Tune axes
        ax.set_xlim(-3e8, 3e8)
        ax.set_ylim(-3e8, 3e8)
        ax.set_zlim(-3e8, 3e8)

        # And finally!
        ax.view_init(30, 260)
        ax.set_title("MSL mission:\nfrom Earth to Mars")
        fig.savefig("trans_30_260.png", bbox_inches='tight')
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

And now, let's do it interactively!

```
In [15]: def go_to_mars(offset=0., tof_=6000.):
             # Initial data
             N = 50

             date_launch = time.Time('2011-11-26 15:02', scale='utc') + offset * u.day
             #date_arrival = time.Time('2012-08-06 05:17', scale='utc')
             tof = tof_ * u.h

             # Calculate vector of times from launch and arrival
```

```
date_arrival = date_launch + tof
dt = (date_arrival - date_launch) / N

# Idea from http://docs.astropy.org/en/stable/time/#getting-started
times_vector = date_launch + dt * np.arange(N + 1)

rr_earth, vv_earth = get_body_ephem("earth", times_vector)
rr_mars, vv_mars = get_body_ephem("mars", times_vector)

# Compute the transfer orbit!
r0 = rr_earth[:, 0]
rf = rr_mars[:, -1]

(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)

# Extract whole orbit of Earth, Mars and transfer (for plotting)
rr_trans = np.zeros_like(rr_earth)
rr_trans[:, 0] = r0
for ii in range(1, len(times_vector)):
    tof = (times_vector[ii] - times_vector[0]).to(u.day)
    rr_trans[:, ii] = ss0_trans.propagate(tof).r

# Better compute backwards
date_final = date_arrival - 1 * u.year
dt2 = (date_final - date_launch) / N

times_rest_vector = date_launch + dt2 * np.arange(N + 1)
rr_earth_rest, _ = get_body_ephem("earth", times_rest_vector)
rr_mars_rest, _ = get_body_ephem("mars", times_rest_vector)

# Plot figure
fig = plt.gcf()
ax = plt.gca()
ax.cla()

def plot_body(ax, r, color, size, border=False, **kwargs):
    """Plots body in axes object.

    """
    return ax.plot(*r[:, None], marker='o', color=color, ms=size, mew=int(border), **kwa

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

# Plotting orbits is easy!
ax.plot(*rr_earth.to(u.km).value, color=color_earth0)
ax.plot(*rr_mars.to(u.km).value, color=color_mars0)

ax.plot(*rr_trans.to(u.km).value, color=color_trans)
ax.plot(*rr_earth_rest.to(u.km).value, ls='--', color=color_orbit)
```

```
        ax.plot(*rr_mars_rest.to(u.km).value, ls='--', color=color_orbit)

        # But plotting planets feels even magical!
        plot_body(ax, np.zeros(3), color_sun, 16)

        plot_body(ax, r0.to(u.km).value, color_earth0, 8)
        plot_body(ax, rr_earth[:, -1].to(u.km).value, color_earthf, 8)

        plot_body(ax, rr_mars[:, 0].to(u.km).value, color_mars0, 8)
        plot_body(ax, rf.to(u.km).value, color_marsf, 8)

        # Add some text
        #ax.text(-0.75e8, -3.5e8, -1.5e8, "MSL mission:\nfrom Earth to Mars", size=20, ha='cente
        ax.text(r0[0].to(u.km).value * 1.4, r0[1].to(u.km).value * 0.4, r0[2].to(u.km).value * 1
                "Earth at launch\n({0:%b %d})".format(date_launch.datetime),
                ha="left", va="bottom", backgroundcolor='#ffffff')
        ax.text(rf[0].to(u.km).value * 0.7, rf[1].to(u.km).value * 1.1, rf[2].to(u.km).value,
                "Mars at arrival\n({0:%b %d})".format(date_arrival.datetime),
                ha="left", va="top", backgroundcolor='#ffffff')
        ax.text(-1.9e8, 8e7, 0, "Transfer\norbit", ha="right", va="center", backgroundcolor='#ff

        # Tune axes
        ax.set_xlim(-3e8, 3e8)
        ax.set_ylim(-3e8, 3e8)
        ax.set_zlim(-3e8, 3e8)
        ax.view_init(30, 260)
```

```
In [16]: fig = plt.figure(figsize=(5, 5))
         ax = fig.add_subplot(111, projection='3d')

         go_to_mars();
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
In [17]: from ipywidgets import interact

         interact(go_to_mars, offset=(-100., 300.), tof_=(100., 12000.));
```

```
Widget Javascript not detected.  It may not be installed or enabled properly.
```

Not bad! Hope you found it interesting. In case you didn't but are still reading, here is some music that you may enjoy:

```
In [18]: from IPython.display import YouTubeVideo
         YouTubeVideo('zSgiXGELjbc')
```

2.4.5 Going to Jupiter with Python using Jupyter and poliastro

```
In [1]: %matplotlib inline
        import numpy as np
        import matplotlib.pyplot as plt

        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 plot, OrbitPlotter
        from poliastro.util import norm

        solar_system_ephemeris.set("jpl")

Out[1]: <ScienceState solar_system_ephemeris: 'jpl'>

In [2]: ## Initial data
        # Links and sources: https://github.com/poliastro/poliastro/wiki/EuroPython:-Per-Python-ad-A
        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')
```

```
In [3]: # Initial state of the Earth
        ss_e0 = Orbit.from_body_ephem(Earth, date_launch)
        r_e0, v_e0 = ss_e0.rv()
```

```
In [4]: r_e0
```

$[1.0246553 \times 10^{8}, \ -1.023135 \times 10^{8}, \ -44353346]$ km

```
In [5]: v_e0
```

$[1847708.5, \ 1594323.4, \ 691089.12] \ \frac{\mathrm{km}}{\mathrm{d}}$

```
In [6]: # 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()
```

```
In [7]: # Assume that the insertion velocity is tangential to that of the Earth
        dv = C_3**.5 * v_e0 / norm(v_e0)
        man = Maneuver.impulse(dv)
```

```
In [8]: # Inner Cruise 1
        ic1 = ss_e0.apply_maneuver(man)
        ic1.rv()
```

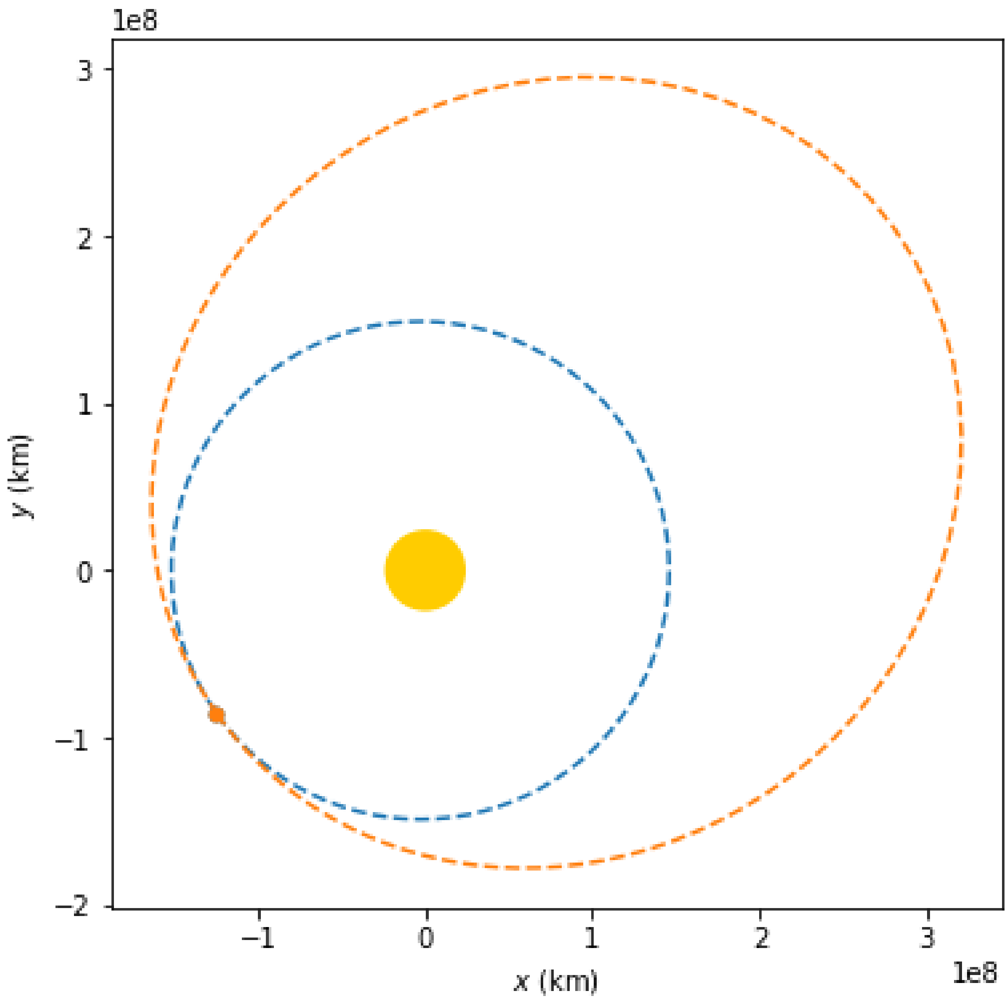
```
Out[8]: (<Quantity [  1.02465527e+08, -1.02313505e+08, -4.43533465e+07] km>,
         <Quantity [ 2198705.82621214, 1897186.74383867,  822370.88977492] km / d>)
```

```
In [9]: ic1.period.to(u.year)
```

2.1515474 yr

```
In [10]: op = OrbitPlotter()

         op.plot(ss_e0)
         op.plot(ic1)
```

```
Out[10]: [<matplotlib.lines.Line2D at 0x22d5f6f7390>,
          <matplotlib.lines.Line2D at 0x22d5f6f7fd0>]
```




```
In [11]: # We propagate until the aphelion
         ss_aph = ic1.propagate(ic1.period / 2)
         ss_aph.epoch
```

```
Out[11]: <Time object: scale='utc' format='iso' value=2012-09-01 14:38:54.507>
```

```
In [12]: # Let's compute the Lambert solution to do the flyby of the Earth
         time_of_flight = date_flyby - ss_aph.epoch
         time_of_flight
```

```
Out[12]: <TimeDelta object: scale='tai' format='jd' value=403.19589691198047>
```

```
In [13]: (v_aph, v_fly), = izzo.lambert(Sun.k, ss_aph.r, ss_efly.r, time_of_flight)
```

```
In [14]: # Check the delta-V
         norm(v_aph - ss_aph.v)  # Too high!
```

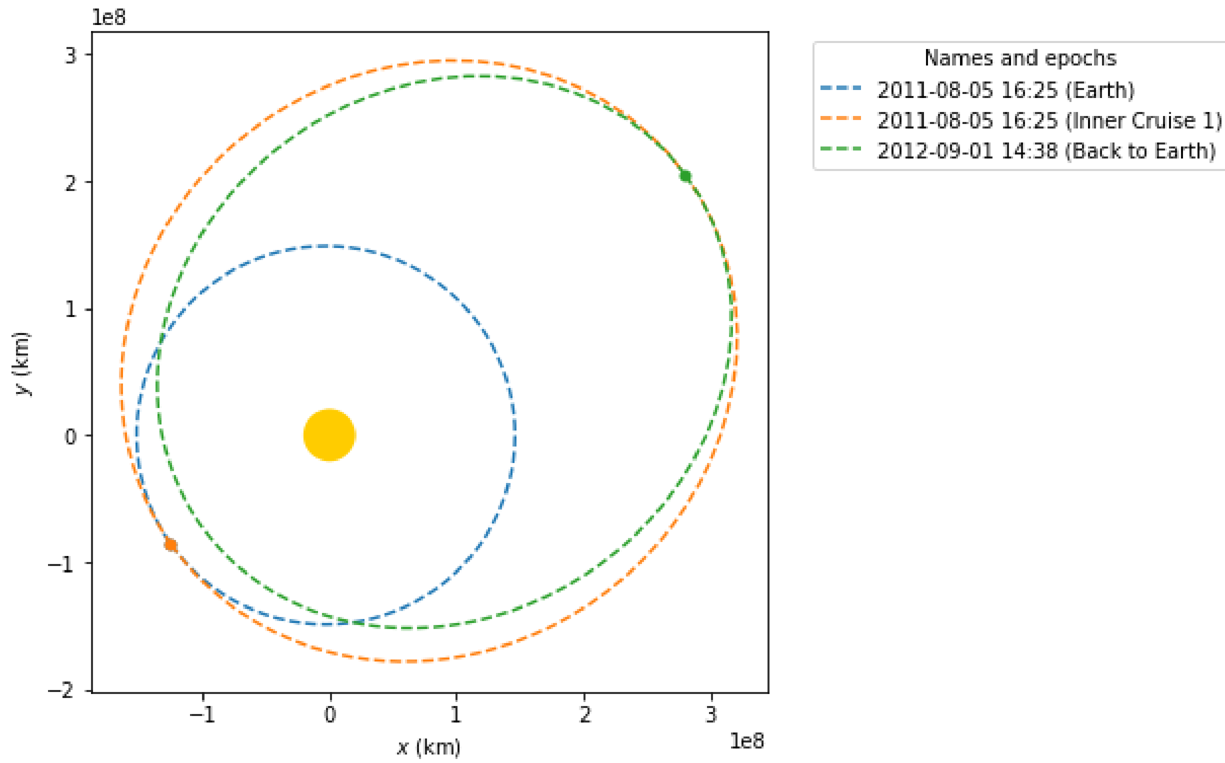
$1.079866 \frac{\mathrm{km}}{\mathrm{s}}$

```
In [15]: 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)
```

```
In [16]: op = OrbitPlotter()

         op.plot(ss_e0, label="Earth")
         op.plot(ic1, label="Inner Cruise 1")
```

```
#op.plot(ss_efly)
op.plot(ss_aph_post, label="Back to Earth")
```

Out[16]:
```
[<matplotlib.lines.Line2D at 0x22d612d8630>,
 <matplotlib.lines.Line2D at 0x22d612d8c18>]
```

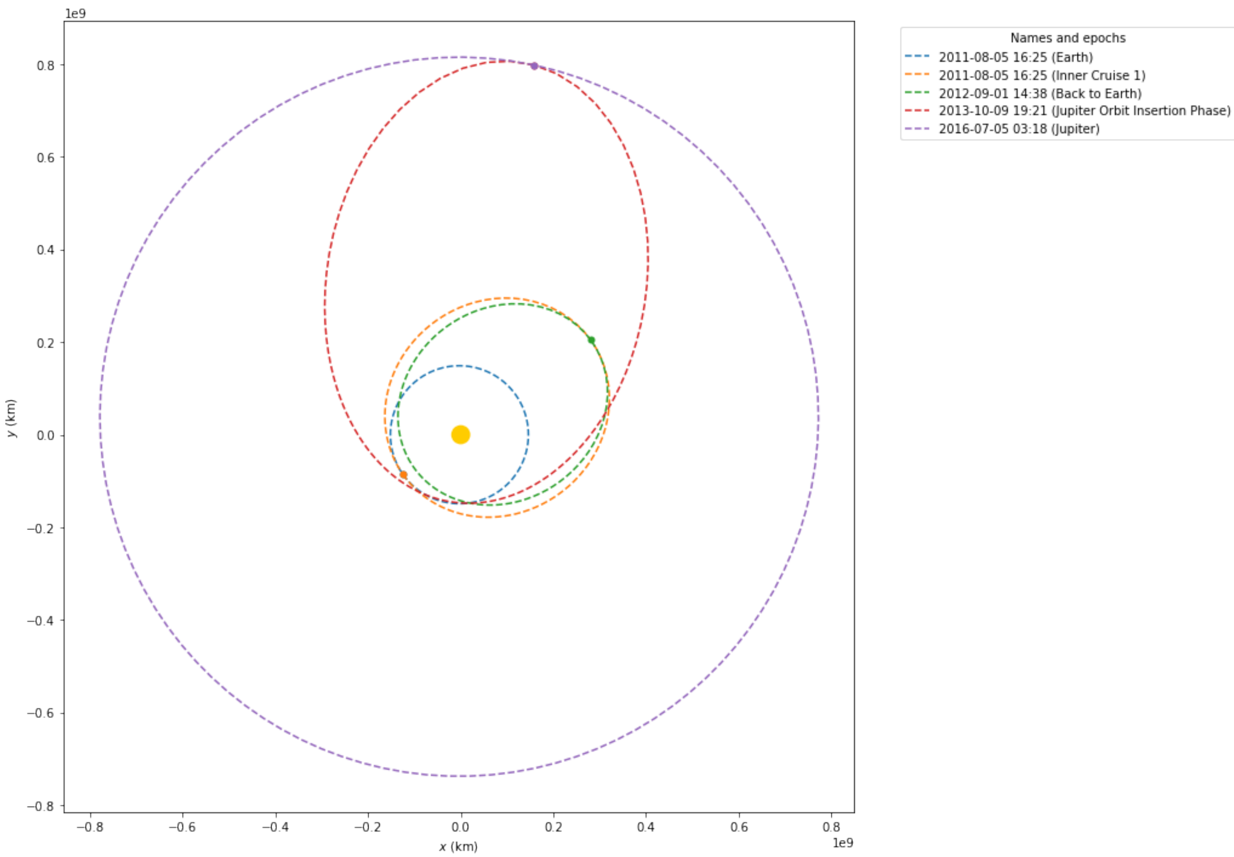


In [17]:
```
# And now, go to Jupiter!
ss_j = Orbit.from_body_ephem(Jupiter, date_arrival)
r_j, v_j = ss_j.rv()
```

In [18]:
```
(v_flypre, v_oip), = izzo.lambert(Sun.k, r_efly, r_j, date_arrival - date_flyby)
```

In [19]:
```
ss_oip = Orbit.from_vectors(Sun, r_j, v_oip, epoch=date_flyby)
```

In [20]:
```
fig, ax = plt.subplots(figsize=(9, 12))

op = OrbitPlotter(ax)

op.plot(ss_e0, label="Earth")
op.plot(ic1, label="Inner Cruise 1")
#op.plot(ss_efly)
op.plot(ss_aph_post, label="Back to Earth")
op.plot(ss_oip, label="Jupiter Orbit Insertion Phase")
op.plot(ss_j, label="Jupiter")

fig.savefig("jupiter.png")
```



2.4.6 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:

$$\ddot{\vec{r}} = -\frac{\mu}{|\vec{r}|^3}\vec{r} + \vec{a}_d$$

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

---

First example

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

```
In [1]: %matplotlib inline
        import numpy as np
        from astropy import units as u

        from matplotlib import ticker
```

```
from matplotlib import pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

from scipy.integrate import ode

from poliastro.bodies import Earth
from poliastro.twobody import Orbit
from poliastro.examples import iss

from poliastro.twobody.propagation import func_twobody

from poliastro.util import norm

from ipywidgets.widgets import interact, fixed
```

```
In [2]: 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])
```

```
In [3]: u0 = state_to_vector(iss)
        u0
```

```
Out[3]: array([  8.59072560e+02,  -4.13720368e+03,   5.29556871e+03,
                 7.37289205e+00,   2.08223573e+00,   4.39999794e-01])
```

```
In [4]: t = np.linspace(0, 10 * iss.period, 500).to(u.s).value
        t[:10]
```

```
Out[4]: array([    0.        ,   111.36211826,   222.72423652,   334.08635478,
                 445.44847304,   556.8105913 ,   668.17270956,   779.53482782,
                 890.89694608,  1002.25906434])
```

```
In [5]: dt = t[1] - t[0]
        dt
```

```
Out[5]: 111.36211825977986
```

```
In [6]: k = Earth.k.to(u.km**3 / u.s**2).value
```

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

```
In [7]: 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

        constant_accel_factory(accel=1e-5)(t[0], u0, k)
```

```
Out[7]: array([  9.60774274e-06,   2.71339728e-06,   5.73371317e-07])
```

```
In [8]: help(func_twobody)
```

```
Help on function func_twobody in module poliastro.twobody.propagation:

func_twobody(t0, u_, k, ad)
    Differential equation for the initial value two body problem.

    This function follows Cowell's formulation.

    Parameters
```

```
----------
t0 : float
    Time.
u_ : ~numpy.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).
```

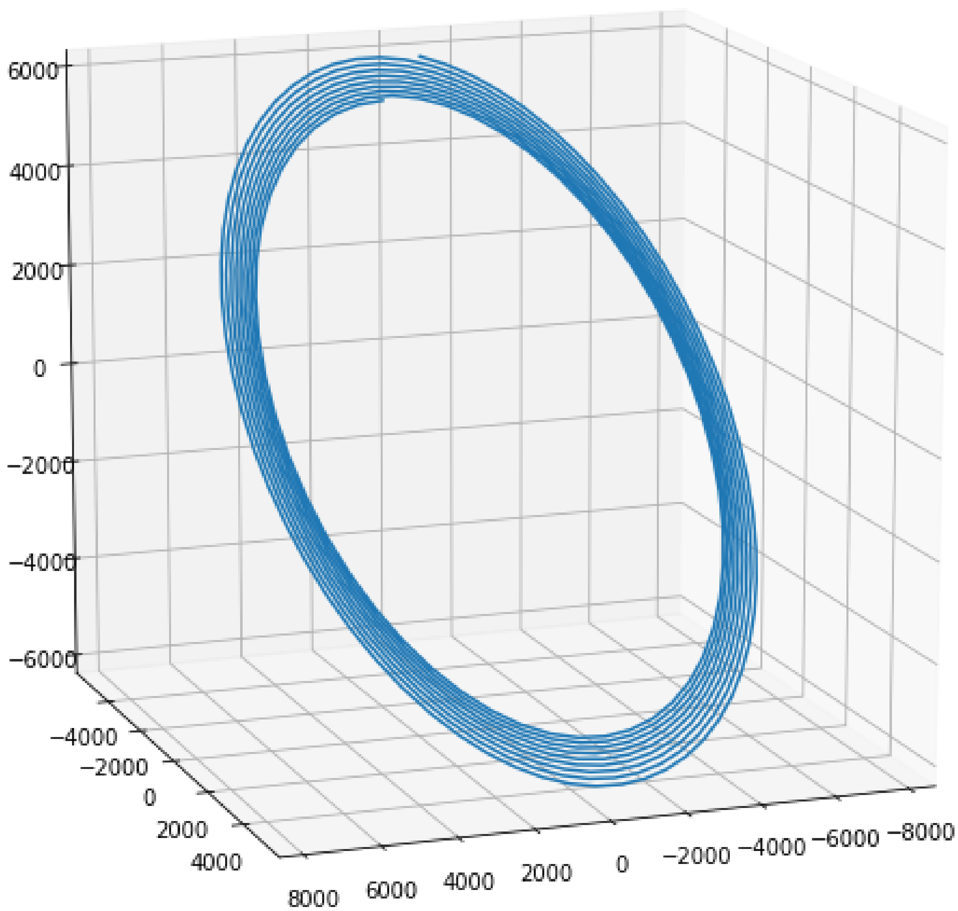
Now we setup the integrator manually using `scipy.integrate.ode`. We cannot provide the Jacobian since we don't know the form of the acceleration in advance.

```
In [9]: res = np.zeros((t.size, 6))
        res[0] = u0
        ii = 1

        accel = 1e-5

        rr = ode(func_twobody).set_integrator('dop853')  # All parameters by default
        rr.set_initial_value(u0, t[0])
        rr.set_f_params(k, constant_accel_factory(accel))

        while rr.successful() and rr.t + dt < t[-1]:
            rr.integrate(rr.t + dt)
            res[ii] = rr.y
            ii += 1

        res[:5]
```

```
Out[9]: array([[  8.59072560e+02,  -4.13720368e+03,   5.29556871e+03,
                  7.37289205e+00,   2.08223573e+00,   4.39999794e-01],
               [  1.67120051e+03,  -3.87307888e+03,   5.30240756e+03,
                  7.19314492e+00,   2.65498748e+00,  -3.17310887e-01],
               [  2.45692273e+03,  -3.54744387e+03,   5.22509021e+03,
                  6.89930296e+00,   3.18546088e+00,  -1.06938976e+00],
               [  3.20378169e+03,  -3.16548222e+03,   5.06486727e+03,
                  6.49612475e+00,   3.66524400e+00,  -1.80427142e+00],
               [  3.89994802e+03,  -2.73326986e+03,   4.82430776e+03,
                  5.99011730e+00,   4.08674433e+00,  -2.51027603e+00]])
```

And we plot the results:

```
In [10]: fig = plt.figure(figsize=(10, 10))
         ax = fig.add_subplot(111, projection='3d')

         ax.plot(*res[:, :3].T)

         ax.view_init(14, 70)
```



Interactivity

This is the last time we used `scipy.integrate.ode` directly. Instead, we can now import a convenient function from poliastro:

```
In [11]: from poliastro.twobody.propagation import cowell
```

```
In [12]: def plot_iss(thrust=0.1, mass=2000.):
             r0, v0 = iss.rv()
             k = iss.attractor.k
             t = np.linspace(0, 10 * iss.period, 500).to(u.s).value
             u0 = state_to_vector(iss)

             res = np.zeros((t.size, 6))
             res[0] = u0
```

```
        accel = thrust / mass

        # Perform the whole integration
        r0 = r0.to(u.km).value
        v0 = v0.to(u.km / u.s).value
        k = k.to(u.km**3 / u.s**2).value
        ad = constant_accel_factory(accel)
        r, v = r0, v0
        for ii in range(1, len(t)):
            r, v = cowell(k, r, v, t[ii] - t[ii - 1], ad=ad)
            x, y, z = r
            vx, vy, vz = v
            res[ii] = [x, y, z, vx, vy, vz]

        fig = plt.figure(figsize=(8, 6))
        ax = fig.add_subplot(111, projection='3d')

        ax.set_xlim(-20e3, 20e3)
        ax.set_ylim(-20e3, 20e3)
        ax.set_zlim(-20e3, 20e3)

        ax.view_init(14, 70)

        return ax.plot(*res[:, :3].T)
```

```
In [13]: interact(plot_iss, thrust=(0.0, 0.2, 0.001), mass=fixed(2000.))
```

```
A Jupyter Widget
```

```
Out[13]: <function __main__.plot_iss>
```

Error checking

```
In [14]: rtol = 1e-13
         full_periods = 2
```

```
In [15]: u0 = state_to_vector(iss)
         tf = ((2 * full_periods + 1) * iss.period / 2).to(u.s).value

         u0, tf
```

```
Out[15]: (array([  8.59072560e+02,  -4.13720368e+03,   5.29556871e+03,
                   7.37289205e+00,   2.08223573e+00,   4.39999794e-01]),
          13892.424252907538)
```

```
In [16]: iss_f_kep = iss.propagate(tf * u.s, rtol=1e-18)
```

```
In [17]: r0, v0 = iss.rv()
         r, v = cowell(k, r0.to(u.km).value, v0.to(u.km / u.s).value, tf, rtol=rtol)

         iss_f_num = Orbit.from_vectors(Earth, r * u.km, v * u.km / u.s, iss.epoch + tf * u.s)
```

```
In [18]: iss_f_num.r, iss_f_kep.r
```

```
Out[18]: (<Quantity [ -835.92108005, 4151.60692532,-5303.60427969] km>,
          <Quantity [ -835.92108005, 4151.60692532,-5303.60427969] km>)
```

```
In [19]: 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)
```

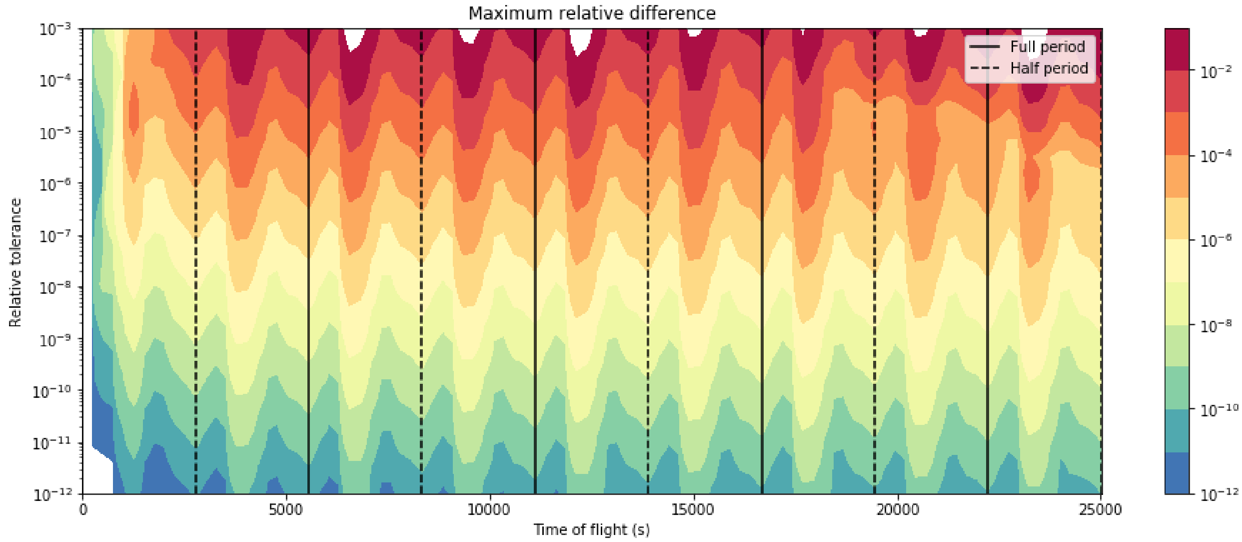
```
In [20]: #assert np.allclose(iss_f_num.a, iss_f_kep.a, rtol=rtol, atol=1e-08 * u.km)
         #assert np.allclose(iss_f_num.ecc, iss_f_kep.ecc, rtol=rtol)
         #assert np.allclose(iss_f_num.inc, iss_f_kep.inc, rtol=rtol, atol=1e-08 * u.rad)
```

```
        #assert np.allclose(iss_f_num.raan, iss_f_kep.raan, rtol=rtol, atol=1e-08 * u.rad)
        #assert np.allclose(iss_f_num.argp, iss_f_kep.argp, rtol=rtol, atol=1e-08 * u.rad)
        #assert np.allclose(iss_f_num.nu, iss_f_kep.nu, rtol=rtol, atol=1e-08 * u.rad)
```

Too bad I cannot access the internal state of the solver. I will have to do it in a blackbox way.

```
In [21]: u0 = state_to_vector(iss)
         full_periods = 4

         tof_vector = np.linspace(0, ((2 * full_periods + 1) * iss.period / 2).to(u.s).value, num=10(
         rtol_vector = np.logspace(-3, -12, num=30)

         res_array = np.zeros((rtol_vector.size, tof_vector.size))
         for jj, tof in enumerate(tof_vector):
             rf, vf = iss.propagate(tof * u.s, rtol=1e-12).rv()
             for ii, rtol in enumerate(rtol_vector):
                 rr = ode(func_twobody).set_integrator('dop853', rtol=rtol, nsteps=1000)
                 rr.set_initial_value(u0, 0.0)
                 rr.set_f_params(k, constant_accel_factory(0.0))  # Zero acceleration

                 rr.integrate(rr.t + tof)

                 if rr.successful():
                     uf = rr.y

                     r, v = uf[:3] * u.km, uf[3:] * u.km / u.s

                     res = max(norm((r - rf) / rf), norm((v - vf) / vf))
                 else:
                     res = np.nan

                 res_array[ii, jj] = res

/home/juanlu/.miniconda36/envs/poliastro36/lib/python3.6/site-packages/scipy/integrate/_ode.py:1035:
  self.messages.get(idid, 'Unexpected idid=%s' % idid))
```

```
In [22]: fig, ax = plt.subplots(figsize=(16, 6))

         xx, yy = np.meshgrid(tof_vector, rtol_vector)

         cs = ax.contourf(xx, yy, res_array, levels=np.logspace(-12, -1, num=12),
                          locator=ticker.LogLocator(), cmap=plt.cm.Spectral_r)
         fig.colorbar(cs)

         for nn in range(full_periods + 1):
             lf = ax.axvline(nn * iss.period.to(u.s).value, color='k', ls='-')
             lh = ax.axvline((2 * nn + 1) * iss.period.to(u.s).value / 2, color='k', ls='--')

         ax.set_yscale('log')

         ax.set_xlabel("Time of flight (s)")
         ax.set_ylabel("Relative tolerance")

         ax.set_title("Maximum relative difference")

         ax.legend((lf, lh), ("Full period", "Half period"))

Out[22]: <matplotlib.legend.Legend at 0x7f9ad3788e80>
```



Numerical validation

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

$$\frac{\mathrm{d}a}{a_0} = 2\frac{F}{mV_0}\,\mathrm{d}t, \qquad \frac{\Delta V}{V_0} = \frac{1}{2}\frac{\Delta a}{a_0}$$

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

```
In [24]: ss = Orbit.circular(Earth, 500 * u.km)
         tof = 20 * ss.period

         ad = constant_accel_factory(1e-7)

         r0, v0 = ss.rv()
         r, v = cowell(k, r0.to(u.km).value, v0.to(u.km / u.s).value,
                       tof.to(u.s).value, ad=ad)

         ss_final = Orbit.from_vectors(Earth, r * u.km, v * u.km / u.s, ss.epoch + rr.t * u.s)
```

```
In [25]: da_a0 = (ss_final.a - ss.a) / ss.a
         da_a0
```

$2.9896209 \times 10^{-6}\ \frac{\mathrm{km}}{\mathrm{m}}$

```
In [26]: dv_v0 = abs(norm(ss_final.v) - norm(ss.v)) / norm(ss.v)
         2 * dv_v0
```

0.0029960537

```
In [27]: np.allclose(da_a0, 2 * dv_v0, rtol=1e-2)
```

```
Out[27]: True
```

```
In [28]: dv = abs(norm(ss_final.v) - norm(ss.v))
         dv
```

$0.011403892\ \frac{\mathrm{km}}{\mathrm{s}}$

```
In [29]: accel_dt = accel * u.km / u.s**2 * (t[-1] - t[0]) * u.s
         accel_dt
```

$0.55569697 \, \frac{\mathrm{km}}{\mathrm{s}}$

```
In [30]: np.allclose(dv, accel_dt, rtol=1e-2, atol=1e-8 * u.km / u.s)
```

```
Out[30]: False
```

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:

```
In [31]: ss_final.ecc
```

6.6621428×10^{-6}

References

- [Edelbaum, 1961] "Propulsion requirements for controllable satellites"

2.4.7 Revisiting Lambert's problem in Python

```
In [1]: %matplotlib notebook
        import numpy as np
        import matplotlib.pyplot as plt
        from cycler import cycler

        from poliastro.iod import izzo

        plt.rc('text', usetex=True)
```

Part 1: Reproducing the original figure

```
In [2]: 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
```

```
In [3]: fig, ax = plt.subplots(figsize=(8, 6))
        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 = izzo._compute_y(x[ii], ll)
                    T_x0[ii] = izzo._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='
        ax.annotate("$\lambda = -1$", xy=(0.3, 2.5), xytext=(0.65, 2.75), arrowprops=dict(arrowstyle=

        ax.grid()
        ax.set_xlabel("$x$")
        ax.set_ylabel("$T$")
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

Part 2: Locating T_{min}

```
In [4]: for M in M_list:
            for ll in ll_list:
                x_T_min, T_min = izzo._compute_T_min(ll, M, 10, 1e-8)
                ax.plot(x_T_min, T_min, 'kx', mew=2)
```

Part 3: Try out solution

```
In [5]: T_ref = 1
        ll_ref = 0

        (x_ref, _), = izzo._find_xy(ll_ref, T_ref, 0, 10, 1e-8)
        x_ref
```

```
Out[5]: 0.43344673453504257
```

```
In [6]: ax.plot(x_ref, T_ref, 'o', mew=2, mec='red', mfc='none')
```

```
Out[6]: [<matplotlib.lines.Line2D at 0x7fb45a5d2da0>]
```

Part 4: Run some examples

```
In [7]: from astropy import units as u

        from poliastro.bodies import Earth
```

Single revolution

```
In [8]: 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)
        v
```

$[-3.4515665, 0.91031354, 0] \frac{\mathrm{km}}{\mathrm{s}}$

```
In [9]: k = Earth.k
        r0 = [5000.0, 10000.0, 2100.0] * u.km
        r = [-14600.0, 2500.0, 7000.0] * u.km
        tof = 1.0 * u.h

        expected_va = [-5.9925, 1.9254, 3.2456] * u.km / u.s
        expected_vb = [-3.3125, -4.1966, -0.38529] * u.km / u.s

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

$[-3.3124585, -4.196619, -0.38528906] \frac{\mathrm{km}}{\mathrm{s}}$

Multiple revolutions

```
In [10]: k = Earth.k
         r0 = [22592.145603, -1599.915239, -19783.950506] * u.km
         r = [1922.067697, 4054.157051, -8925.727465] * u.km
         tof = 10 * u.h

         expected_va = [2.000652697, 0.387688615, -2.666947760] * u.km / u.s
         expected_vb = [-3.79246619, -1.77707641, 6.856814395] * u.km / u.s

         expected_va_l = [0.50335770, 0.61869408, -1.57176904] * u.km / u.s
         expected_vb_l = [-4.18334626, -1.13262727, 6.13307091] * u.km / u.s

         expected_va_r = [-2.45759553, 1.16945801, 0.43161258] * u.km / u.s
         expected_vb_r = [-5.53841370, 0.01822220, 5.49641054] * u.km / u.s
```

```
In [11]: (v0, v), = izzo.lambert(k, r0, r, tof, M=0)
         v
```

$[-3.7924662, -1.7770764, 6.8568144] \frac{\mathrm{km}}{\mathrm{s}}$

```
In [12]: (_, v_l), (_, v_r) = izzo.lambert(k, r0, r, tof, M=1)
```

```
In [13]: v_l
```

$[-4.1833463, -1.1326273, 6.1330709] \frac{\mathrm{km}}{\mathrm{s}}$

```
In [14]: v_r
```

$[-5.5384132, 0.018222134, 5.4964102] \frac{\mathrm{km}}{\mathrm{s}}$

2.4.8 Studying Hohmann transfers

```
In [1]: %matplotlib inline
        import numpy as np

        import matplotlib.pyplot as plt

        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
```

```
In [2]: Earth.k
```

$3.9860044 \times 10^{14} \frac{\mathrm{m}^3}{\mathrm{s}^2}$

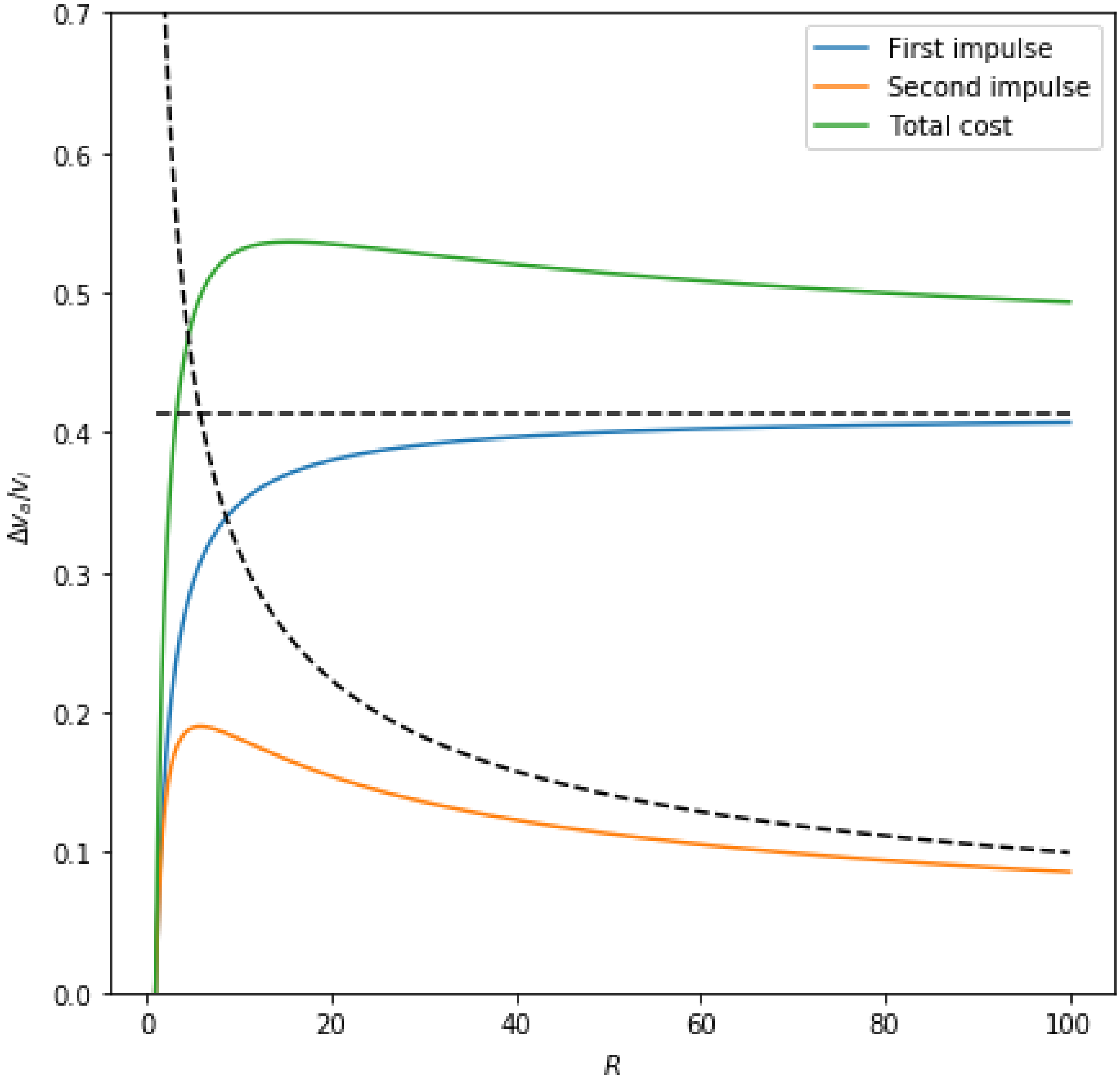
```
In [3]: ss_i = Orbit.circular(Earth, alt=800 * u.km)
        ss_i

Out[3]: 7178 x 7178 km x 0.0 deg orbit around Earth ()

In [4]: r_i = ss_i.a.to(u.km)
        r_i
```

7178.1366 km

```
In [5]: v_i_vec = ss_i.v.to(u.km / u.s)
        v_i = norm(v_i_vec)
        v_i
```

$7.4518315 \frac{\mathrm{km}}{\mathrm{s}}$

```
In [6]: N = 1000
        dv_a_vector = np.zeros(N) * u.km / u.s
        dv_b_vector = dv_a_vector.copy()
        r_f_vector = r_i * np.linspace(1, 100, num=N)
        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)

In [8]: 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 cos

        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")
```



2.4.9 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.

```
In [1]: %matplotlib inline

        from astropy import time
        from poliastro.twobody.orbit import Orbit
        from poliastro.bodies import Earth
        from poliastro.plotting import OrbitPlotter
```

NeoWS module

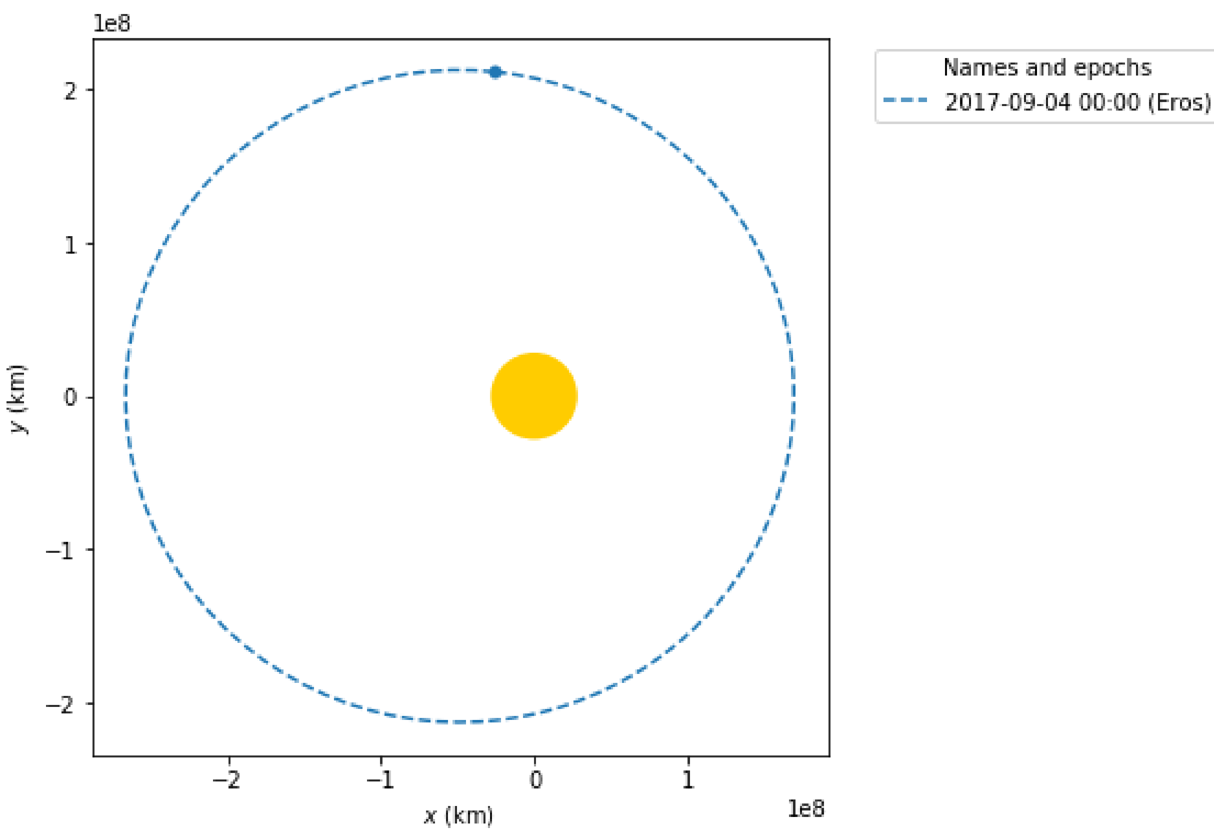
This module make requests to NASA NEO Webservice, so you'll need an internet connection to run the next examples.

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

```
In [2]: from poliastro.neos import neows
```

```
In [3]: eros = neows.orbit_from_name('Eros')

        frame = OrbitPlotter()
        frame.plot(eros, label='Eros')
```

```
Out[3]: [<matplotlib.lines.Line2D at 0x7fe2970f80f0>,
         <matplotlib.lines.Line2D at 0x7fe2970f89e8>]
```

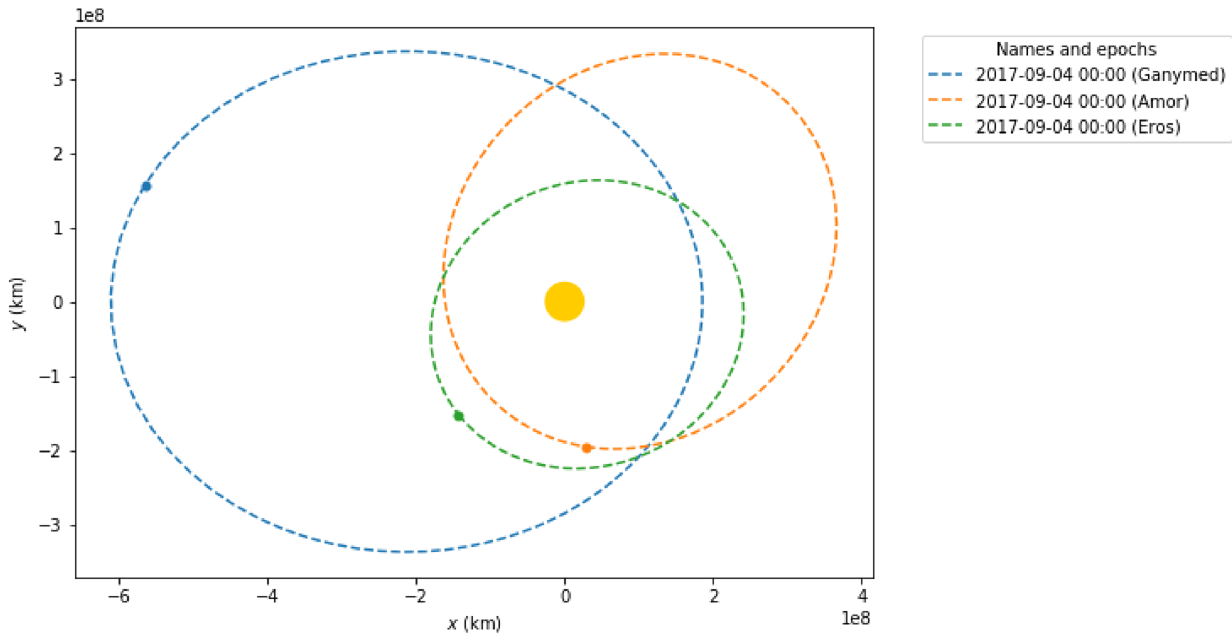


You can also search by IAU number or SPK-ID (there is a faster `neows.orbit_from_spk_id()` function in that case, although):

```
In [4]: 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 = OrbitPlotter()
frame.plot(ganymed, label='Ganymed')
frame.plot(amor, label='Amor')
frame.plot(eros, label='Eros')
```

```
Out[4]: [<matplotlib.lines.Line2D at 0x7fe296e03be0>,
         <matplotlib.lines.Line2D at 0x7fe296e03dd8>]
```



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.

```
In [5]: neows.orbit_from_name('*alley')
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-5-8e1358d8245f> in <module>()
----> 1 neows.orbit_from_name('*alley')

~/Development/poliastro/poliastro-library/src/poliastro/neos/neows.py in orbit_from_name(name, api_ke
    126
    127     """
--> 128     spk_id = spk_id_from_name(name)
    129     if spk_id is not None:
    130         return orbit_from_spk_id(spk_id, api_key)

~/Development/poliastro/poliastro-library/src/poliastro/neos/neows.py in spk_id_from_name(name)
    101         for body in object_list[:obj_num]:
    102             bodies += body.string + '\n'
--> 103         raise ValueError(str(len(object_list)) + ' different bodies found:\n' + bodies)
    104     else:
```

```
    105         raise ValueError('Object could not be found. You can visit: ' +

ValueError: 6 different bodies found:
903 Nealley (1918 EM)
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:

```
In [6]: eros.epoch.iso

Out[6]: '2017-09-04 00:00'

In [7]: epoch = time.Time(2458000.0, scale='tdb', format='jd')
        eros_november = eros.propagate(epoch)
        eros_november.epoch.iso

Out[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:

```
In [8]: neows.orbit_from_name('Toutatis', api_key='DEMO_KEY')

Out[8]: 1 x 4 AU x 0.4 deg orbit around Sun ()
```

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).

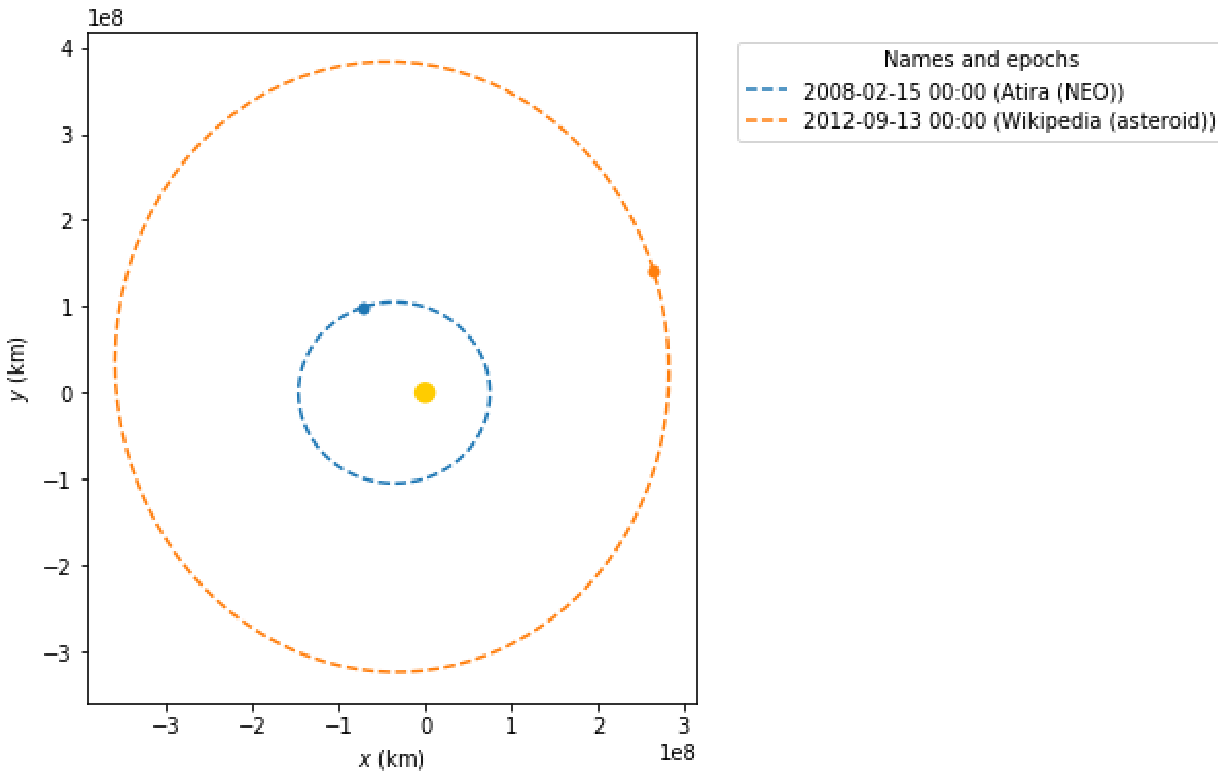
It relies on DASTCOM5 database, a NASA/JPL maintained asteroid and comet database. This database has to be downloaded at least once in order to use this module. According to its README, it is updated typically a couple times per day, but potentially as frequently as once per hour, so you can download 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()`:

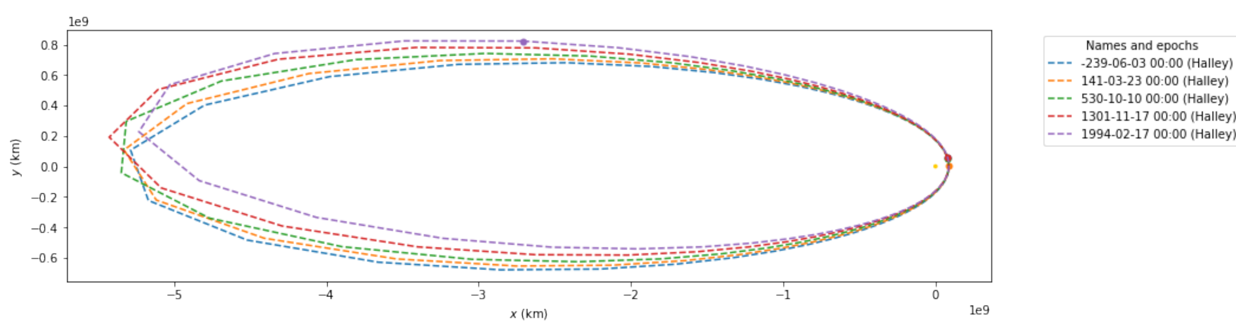
```
In [9]: from poliastro.neos import dastcom5

In [10]: atira = dastcom5.orbit_from_name('atira')[0] # NEO
         wikipedia = dastcom5.orbit_from_name('wikipedia')[0] # Asteroid, but not NEO.
         frame = OrbitPlotter()
         frame.plot(atira, label='Atira (NEO)')
         frame.plot(wikipedia, label='Wikipedia (asteroid)')

Out[10]: [<matplotlib.lines.Line2D at 0x22c8cdf2828>,
          <matplotlib.lines.Line2D at 0x22c8cdf2a58>]
```



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:

```
In [11]: halleys = dastcom5.orbit_from_name('1P')

         frame = OrbitPlotter()
         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')
```

```
Out[11]: [<matplotlib.lines.Line2D at 0x22c8cb79668>,
          <matplotlib.lines.Line2D at 0x22c8cb34e80>]
```



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:

```
In [12]: 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 r
```

```
Out[12]: ('NO',
          'NOBS',
          'OBSFRST',
          'OBSLAST',
          'EPOCH',
          'CALEPO',
          'MA',
          'W',
          'OM',
          'IN',
          'EC',
          'A',
          'QR',
          'TP',
          'TPCAL',
          'TPFRAC',
          'SOLDAT',
          'SRC1',
          'SRC2',
          'SRC3')
```

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 = 2*a - q`, but there are probably many other ways.

```
In [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:

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

```
Out[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 :)

```
In [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`.

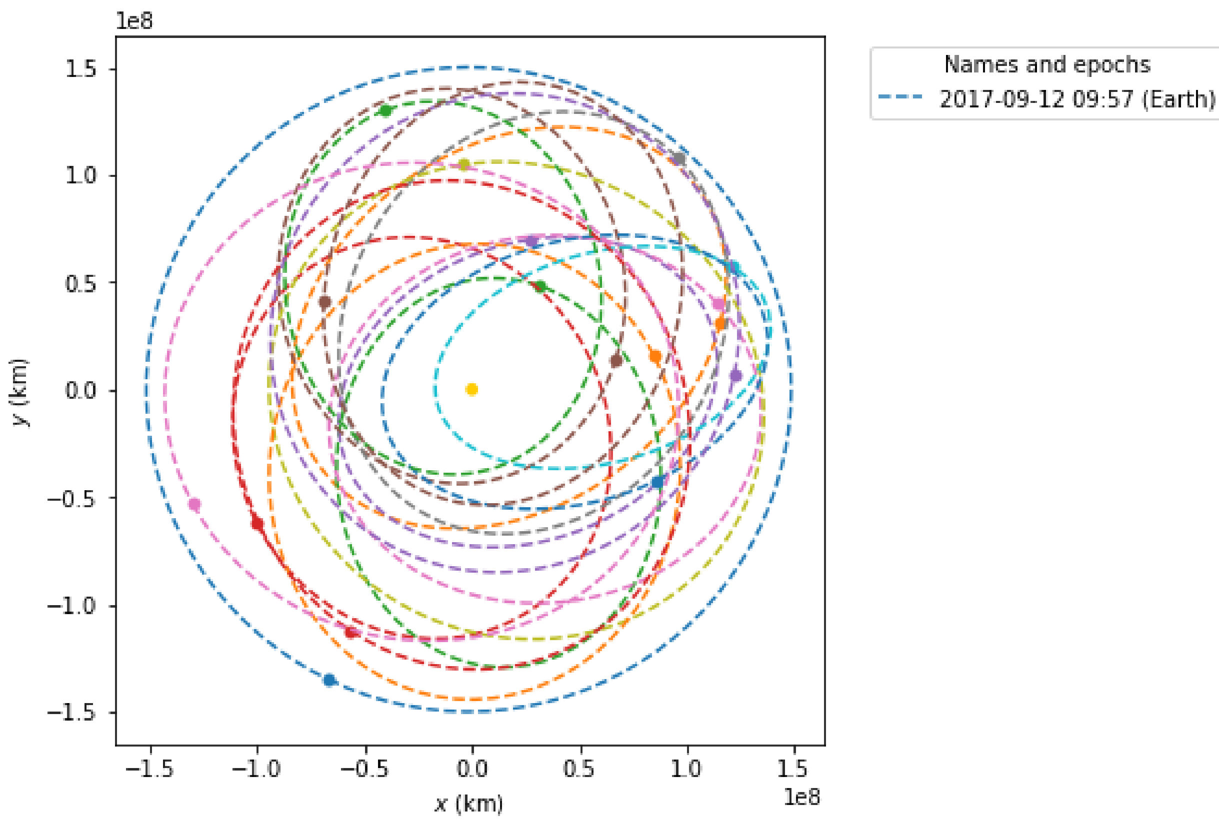
There are two ways:

- 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:

```
In [16]: frame = OrbitPlotter()

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

         for record in atiras['NO']:
             ss = dastcom5.orbit_from_record(record)
             frame.plot(ss)
```

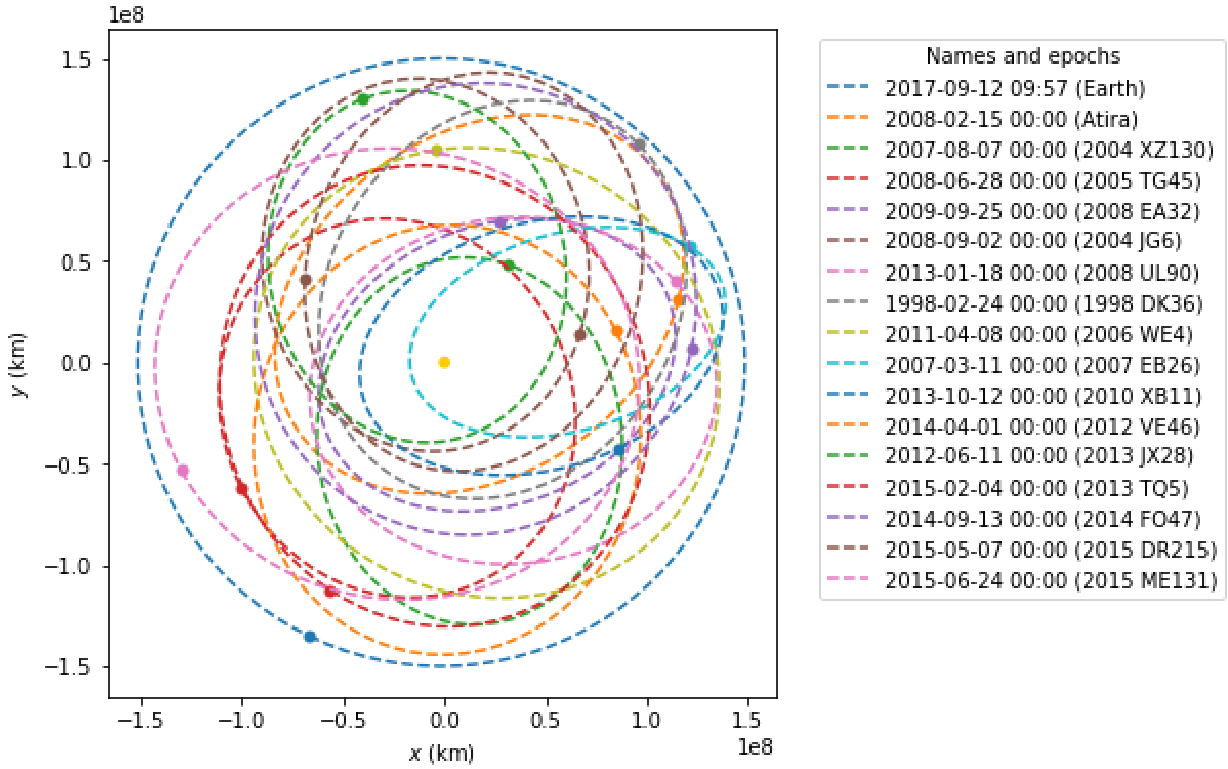


This slightly incorrect, given that Earth coordinates are in a different frame from asteroids. However, for the purpose of this notebook, the effect is barely noticeable.

If we needed also the names of each asteroid, we could do:

```
In [17]: frame = OrbitPlotter()

         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)
    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:

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

```
Out[18]: 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):

```
In [19]: db[db.NAME == 'Halley'] # As you can see, Halley is the name of an asteroid too, did you kno
```

```
Out[19]: NUMBER   NOBS    OBSFRST    OBSLAST       EPOCH     CALEPOCH           MA   \
         2687       2688   1732  19500814  20160827  2455701.5  20110520.0   22.786977
         737341   900001    161         0         0  1633920.5  -2390607.0    0.165294
         737342   900002    161         0         0  1661840.5  -1631115.0    0.031113
         737343   900003    161         0         0  1689880.5   -860823.0    0.211345
         737344   900004    161         0         0  1717320.5   -111008.0  359.963217
         737345   900005    161         0         0  1745200.5    660206.0    0.142118
         737346   900006    161         0         0  1772640.5   1410324.0    0.019990
```

```
737347  900007   161         0         0  1800800.5   2180429.0  359.761537
737348  900008   161         0         0  1828920.5   2950425.0    0.057332
737349  900009   161         0         0  1857720.5   3740301.0    0.158377
737350  900010   161         0         0  1885960.5   4510625.0  359.959606
737351  900011   161         0         0  1914920.5   5301008.0    0.135791
737352  900012   161         0         0  1942840.5   6070318.0    0.032109
737353  900013   161         0         0  1971160.5   6840929.0  359.952171
737354  900014   161         0         0  1998800.5   7600602.0    0.157833
737355  900015   161         0         0  2026840.5   8370310.0    0.124640
737356  900016   161         0         0  2054360.5   9120714.0  359.940520
737357  900017   161         0         0  2082520.5   9890819.0  359.774025
737358  900018   161         0         0  2110480.5  10660308.0  359.839206
737359  900019   161         0         0  2139360.5  11450402.0  359.793477
737360  900020   161         0         0  2167680.5  12221015.0    0.201554
737361  900021   161         0         0  2196560.5  13011109.0    0.179564
737362  900022   161         0         0  2224680.5  13781105.0  359.927910
737363  900023   161         0         0  2253040.5  14560628.0    0.234781
737364  900024   161         0         0  2280480.5  15310814.0  359.842327
737365  900025   161         0         0  2308300.5  16071024.0  359.954121
737366  900026   161         0         0  2308300.5  16071024.0  359.954121
737367  900027   278         0         0  2335640.5  16820831.0  359.805545
737368  900028   278         0         0  2335640.5  16820831.0  359.805545
737369  900029   718         0         0  2363600.5  17590321.0    0.101706
737370  900030   718         0         0  2363600.5  17590321.0    0.101706
737371  900031   653         0         0  2391600.5  18351118.0    0.020156
737372  900032   653         0         0  2418800.5  19100509.0    0.243754
737373  900033  7428  18350821  19940111  2449400.5  19940217.0   38.384264

                 W         OM          IN        EC          A        QR  \
2687    183.484408  95.422520    3.454589  0.143215   3.165183  2.711882
737341   88.110000  30.810000  163.470000  0.967600  18.067901  0.585400
737342   89.110000  32.060000  163.700000  0.967700  18.095975  0.584500
737343   90.778000  34.018000  163.340000  0.967680  18.118812  0.585600
737344   92.559000  35.904000  163.589000  0.967370  17.995709  0.587200
737345   92.652000  36.129000  163.577000  0.967550  18.030817  0.585100
737346   93.694000  37.219000  163.437000  0.967840  18.132463  0.583140
737347   94.147000  37.908000  163.574000  0.967980  18.159588  0.581470
737348   95.241000  39.111000  163.367000  0.968750  18.429120  0.575910
737349   96.510000  40.579000  163.542000  0.968590  18.375995  0.577190
737350   97.028000  41.210000  163.479000  0.968910  18.454165  0.573740
737351   97.582000  41.974000  163.394000  0.968710  18.395334  0.575590
737352   98.799000  43.261000  163.476000  0.968040  18.173655  0.580830
737353   99.149000  43.800000  163.418000  0.968150  18.197174  0.579580
737354   99.997000  44.687000  163.443000  0.967850  18.097667  0.581840
737355  100.101000  44.930000  163.447000  0.967810  18.090090  0.582320
737356  100.777000  45.646000  163.311000  0.968070  18.169746  0.580160
737357  101.484000  46.561000  163.399000  0.967890  18.122392  0.581910
737358  102.473000  47.624000  163.112000  0.968870  18.454867  0.574500
737359  103.704000  49.054000  163.224000  0.968790  18.416854  0.574790
737360  103.849000  49.304000  163.192000  0.968840  18.427792  0.574210
737361  104.500000  50.152000  163.076000  0.968930  18.432893  0.572710
737362  105.295000  51.020000  163.113000  0.968370  18.216883  0.576200
737363  105.835000  51.866000  162.890000  0.968000  18.115625  0.579700
737364  106.976000  53.057000  162.917000  0.967750  18.021705  0.581200
737365  107.550300  53.770100  162.905500  0.967490  17.951861  0.583615
737366  107.550300  53.770100  162.905500  0.967490  17.951861  0.583615
737367  109.221400  55.566900  162.264900  0.967933  18.168865  0.582621
737368  109.221400  55.566900  162.264900  0.967933  18.168865  0.582621
737369  110.709300  57.245800  162.372500  0.967686  18.087083  0.584466
```

```
737370  110.709300  57.245800  162.372500  0.967686  18.087083  0.584466
737371  110.704300  57.518500  162.258800  0.967394  17.989419  0.586563
737372  111.737100  58.562900  162.218600  0.967302  17.958530  0.587208
737373  111.332485  58.420081  162.262691  0.967143  17.834144  0.585978

                  TP         TPCAL    TPFRAC        SOLDAT     DESIG  \
2687    2.455571e+06  2.011011e+07  0.309024  2.457850e+06  1982 HG1
737341  1.633908e+06 -2.390525e+06  0.620000  0.000000e+00        1P
737342  1.661838e+06 -1.631113e+06  0.070000  0.000000e+00        1P
737343  1.689864e+06 -8.608065e+05  0.962000  0.000000e+00        1P
737344  1.717323e+06 -1.110108e+05  0.349000  0.000000e+00        1P
737345  1.745189e+06  6.601260e+05  0.460000  0.000000e+00        1P
737346  1.772639e+06  1.410322e+06  0.934000  0.000000e+00        1P
737347  1.800819e+06  2.180518e+06  0.223000  0.000000e+00        1P
737348  1.828916e+06  2.950420e+06  0.898000  0.000000e+00        1P
737349  1.857708e+06  3.740216e+06  0.842000  0.000000e+00        1P
737350  1.885964e+06  4.510628e+06  0.749000  0.000000e+00        1P
737351  1.914910e+06  5.300927e+06  0.630000  0.000000e+00        1P
737352  1.942838e+06  6.070315e+06  0.976000  0.000000e+00        1P
737353  1.971164e+06  6.841003e+06  0.267000  0.000000e+00        1P
737354  1.998788e+06  7.600521e+06  0.171000  0.000000e+00        1P
737355  2.026831e+06  8.370228e+06  0.770000  0.000000e+00        1P
737356  2.054365e+06  9.120719e+06  0.174000  0.000000e+00        1P
737357  2.082538e+06  9.890906e+06  0.188000  0.000000e+00        1P
737358  2.110493e+06  1.066032e+07  0.434000  0.000000e+00        1P
737359  2.139377e+06  1.145042e+07  0.061000  0.000000e+00        1P
737360  2.167664e+06  1.222093e+07  0.323000  0.000000e+00        1P
737361  2.196546e+06  1.301103e+07  0.082000  0.000000e+00        1P
737362  2.224686e+06  1.378111e+07  0.187000  0.000000e+00        1P
737363  2.253022e+06  1.456061e+07  0.133000  0.000000e+00        1P
737364  2.280493e+06  1.531083e+07  0.739000  0.000000e+00        1P
737365  2.308304e+06  1.607103e+07  0.040600  0.000000e+00        1P
737366  2.308304e+06  1.607103e+07  0.040600  0.000000e+00        1P
737367  2.335656e+06  1.682092e+07  0.779400  0.000000e+00        1P
737368  2.335656e+06  1.682092e+07  0.779400  0.000000e+00        1P
737369  2.363593e+06  1.759031e+07  0.562300  0.000000e+00        1P
737370  2.363593e+06  1.759031e+07  0.562300  0.000000e+00        1P
737371  2.391599e+06  1.835112e+07  0.939600  0.000000e+00        1P
737372  2.418782e+06  1.910042e+07  0.678500  0.000000e+00        1P
737373  2.446467e+06  1.986021e+07  0.395317  2.452124e+06        1P

              IREF    NAME
2687            23  Halley
737341  SAO/-239  Halley
737342  SAO/-163  Halley
737343   SAO/-86  Halley
737344   SAO/-11  Halley
737345    SAO/66  Halley
737346   SAO/141  Halley
737347   SAO/218  Halley
737348   SAO/295  Halley
737349   SAO/374  Halley
737350   SAO/451  Halley
737351   SAO/530  Halley
737352   SAO/607  Halley
737353   SAO/684  Halley
737354   SAO/760  Halley
737355   SAO/837  Halley
737356   SAO/912  Halley
```

```
737357   SAO/989  Halley
737358  SAO/1066  Halley
737359  SAO/1145  Halley
737360  SAO/1222  Halley
737361  SAO/1301  Halley
737362  SAO/1378  Halley
737363  SAO/1456  Halley
737364  SAO/1531  Halley
737365    H593/0  Halley
737366  SAO/1607  Halley
737367    I353/0  Halley
737368  SAO/1682  Halley
737369    J103/0  Halley
737370  SAO/1759  Halley
737371  SAO/1835  Halley
737372  SAO/1910  Halley
737373   J863/77  Halley
```

Panda offers many functionalities, and can also be used in the same way as the `ast_db` and `comet_db` functions:

```
In [20]: aphelion_condition = (2 * db['A'] - db['QR']) < 0.983
         axis_condition = db['A'] < 1.3
         atiras = db[aphelion_condition & axis_condition]
```

```
In [21]: len(atiras)
```

```
Out[21]: 347
```

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!

```
In [22]: len(atiras[atiras.A < 0])
```

```
Out[22]: 331
```

So, rewriting our condition:

```
In [23]: axis_condition = (db['A'] < 1.3) & (db['A'] > 0)
         atiras = db[aphelion_condition & axis_condition]
         len(atiras)
```

```
Out[23]: 16
```

2.5 References

Nanos gigantum humeris insidentes.

2.5.1 Books and papers

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

- Vallado, David A., and Wayne D. McClain. *Fundamentals of astrodynamics and applications*. Vol. 12. Springer Science & Business Media, 2001.
- Curtis, Howard. *Orbital mechanics for engineering students*. Butterworth-Heinemann, 2013.
- Bate, Roger R., Donald D. Mueller, William W. Saylor, and Jerry E. White. *Fundamentals of astrodynamics: (dover books on physics)*. Dover publications, 2013.

- Battin, Richard H. *An introduction to the mathematics and methods of astrodynamics*. Aiaa, 1999.
- Edelbaum, Theodore N. "Propulsion requirements for controllable satellites." *ARS Journal* 31, no. 8 (1961): 1079-1089.
- Walker, M. J. H., B. Ireland, and Joyce Owens. "A set modified equinoctial orbit elements." *Celestial Mechanics* 36.4 (1985): 409-419.

2.5.2 Software

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

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

2.6 API Reference

2.6.1 poliastro.twobody package

poliastro.twobody.angles module

Angles and anomalies.

`poliastro.twobody.angles.`**`nu_to_E`**`(`*nu*, *ecc*`)`

Eccentric anomaly from true anomaly.

New in version 0.4.0.

Parameters

- **nu** (*float*) – True anomaly (rad).
- **ecc** (*float*) – Eccentricity.

Returns **E** – Eccentric anomaly.

Return type float

`poliastro.twobody.angles.`**`nu_to_F`**`(`*nu*, *ecc*`)`

Hyperbolic eccentric anomaly from true anomaly.

Parameters

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

Returns **F** – Hyperbolic eccentric anomaly.

Return type float

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

---

poliastro.twobody.angles.**E_to_nu**(*E*, *ecc*)

True anomaly from eccentric anomaly.

New in version 0.4.0.

Parameters

- **E** (*float*) – Eccentric anomaly (rad).
- **ecc** (*float*) – Eccentricity.

Returns **nu** – True anomaly (rad).

Return type float

poliastro.twobody.angles.**F_to_nu**(*F*, *ecc*)

True anomaly from hyperbolic eccentric anomaly.

Parameters

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

Returns **nu** – True anomaly (rad).

Return type float

poliastro.twobody.angles.**M_to_E**(*M*, *ecc*)

Eccentric anomaly from mean anomaly.

New in version 0.4.0.

Parameters

- **M** (*float*) – Mean anomaly (rad).
- **ecc** (*float*) – Eccentricity.

Returns **E** – Eccentric anomaly.

Return type float

poliastro.twobody.angles.**M_to_F**(*M*, *ecc*)

Hyperbolic eccentric anomaly from mean anomaly.

Parameters

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

Returns **F** – Hyperbolic eccentric anomaly.

Return type float

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

New in version 0.4.0.

Parameters

- **E** (*float*) – Eccentric anomaly (rad).
- **ecc** (*float*) – Eccentricity.

Returns **M** – Mean anomaly (rad).

Return type float

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

Parameters

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

Returns **M** – Mean anomaly (rad).

Return type float

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

New in version 0.4.0.

Parameters

- **M** (*float*) – Mean anomaly (rad).
- **ecc** (*float*) – Eccentricity.

Returns **nu** – True anomaly (rad).

Return type float

Examples

```
>>> nu = M_to_nu(np.radians(30.0), 0.06)
>>> np.rad2deg(nu)
33.673284930211658
```

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

New in version 0.4.0.

Parameters

- **nu** (*float*) – True anomaly (rad).
- **ecc** (*float*) – Eccentricity.

Returns **M** – Mean anomaly (rad).

Return type float

`poliastro.twobody.angles.`**`fp_angle`**(*nu*, *ecc*)

Flight path angle.

New in version 0.4.0.

Parameters

- **nu** (*float*) – True anomaly (rad).
- **ecc** (*float*) – Eccentricity.

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

poliastro.twobody.classical module

Functions to define orbits from classical orbital elements.

`poliastro.twobody.classical.`**`rv_pqw`**(*k*, *p*, *ecc*, *nu*)

Returns r and v vectors in perifocal frame.

`poliastro.twobody.classical.`**`coe2rv`**(*k*, *p*, *ecc*, *inc*, *raan*, *argp*, *nu*)

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.twobody.classical.`**`coe2mee`**(*p*, *ecc*, *inc*, *raan*, *argp*, *nu*)

Converts from classical orbital elements to modified equinoctial orbital elements.

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

Parameters

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

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

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

`a`
Semimajor axis.

`ecc`
Eccentricity.

`inc`
Inclination.

`raan`
Right ascension of the ascending node.

`argp`
Argument of the perigee.

`nu`
True anomaly.

`to_vectors`()
Converts to position and velocity vector representation.

`to_classical`()
Converts to classical orbital elements representation.

`to_equinoctial`()
Converts to modified equinoctial elements representation.

poliastro.twobody.decorators module

Decorators.

poliastro.twobody.equinoctial module

Functions to define orbits from modified equinoctial orbital elements.

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

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

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

poliastro.twobody.orbit module

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

`state`
Position and velocity or orbital elements.

`epoch`
Epoch of the orbit.

classmethod from_vectors(*attractor*, *r*, *v*, *epoch=<Time object: scale='tdb' format='jyear_str' value=J2000.000>*)
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.

classmethod from_classical(*attractor*, *a*, *ecc*, *inc*, *raan*, *argp*, *nu*, *epoch=<Time object: scale='tdb' format='jyear_str' value=J2000.000>*)
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.

classmethod from_equinoctial(*attractor*, *p*, *f*, *g*, *h*, *k*, *L*, *epoch=<Time object: scale='tdb' format='jyear_str' value=J2000.000>*)
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.

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

classmethod circular(*attractor*, *alt*, *inc=<Quantity 0.0 deg>*, *raan=<Quantity 0.0 deg>*, *arglat=<Quantity 0.0 deg>*, *epoch=<Time object: scale='tdb' format='jyear_str' value=J2000.000>*)
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.

classmethod parabolic (*attractor, p, inc, raan, argp, nu, epoch=<Time object: scale='tdb' format='jyear_str' value=J2000.000>*)

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.

propagate (*epoch_or_duration, rtol=1e-10*)

Propagate this *Orbit* some *time* and return the result.

Parameters

- **epoch_or_duration** (*Time, TimeDelta or equivalent*) – Final epoch or time of flight.
- **rtol** (*float, optional*) – Relative tolerance for the propagation algorithm, default to 1e-10.

sample (*values=100*)

Samples an orbit to some specified time values.

New in version 0.8.0.

Parameters values (*Multiple options*) – Number of interval points (default to 100), True anomaly values, Time values.

Returns Position vector in each given value.

Return type CartesianRepresentation

Notes

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

Examples

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

apply_maneuver (*maneuver*, *intermediate=False*)
Returns resulting *Orbit* after applying maneuver to self.

Optionally return intermediate states (default to False).

Parameters

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

poliastro.twobody.propagation module

Propagation algorithms.

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

poliastro.twobody.propagation.**cowell** (*k*, *r0*, *v0*, *tof*, *rtol=1e-10*, ***, *ad=None*, *callback=None*, *nsteps=1000*)
Propagates orbit using Cowell's formulation.

Parameters

- **k** (*float*) – Gravitational constant of main attractor (km^3 / s^2).
- **r0** (*array*) – Initial position (km).
- **v0** (*array*) – Initial velocity (km).
- **ad** (*function(t0, u, k), optional*) – Non Keplerian acceleration (km/s2), default to None.
- **tof** (*float*) – Time of flight (s).
- **rtol** (*float, optional*) – Maximum relative error permitted, default to 1e-10.
- **nsteps** (*int, optional*) – Maximum number of internal steps, default to 1000.
- **callback** (*callable, optional*) – Function called at each internal integrator step.

Raises RuntimeError – If the algorithm didn't converge.

Note: This method uses a Dormand & Prince method of order 8(5,3) available in the `scipy.integrate.ode` module.

`poliastro.twobody.propagation.`**`kepler`**(*k*, *r0*, *v0*, *tof*, *rtol=1e-10*, *, *numiter=35*)

Propagates Keplerian orbit.

Parameters

- **`k`** (*`float`*) – Gravitational constant of main attractor (km^3 / s^2).
- **`r0`** (*`array`*) – Initial position (km).
- **`v0`** (*`array`*) – Initial velocity (km).
- **`tof`** (*`float`*) – Time of flight (s).
- **`rtol`** (*`float, optional`*) – Maximum relative error permitted, default to 1e-10.
- **`numiter`** (*`int, optional`*) – Maximum number of iterations, default to 35.

Raises `RuntimeError` – If the algorithm didn't converge.

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.

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

Propagate an orbit some time and return the result.

poliastro.twobody.rv module

Functions to define orbits from position and velocity vectors.

`poliastro.twobody.rv.`**`rv2coe`**(*k*, *r*, *v*, *tol=1e-08*)

Converts from vectors to classical orbital elements.

Parameters

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

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

State defined by its position and velocity vectors.

`r`

Position vector.

`v`

Velocity vector.

`to_vectors`()

Converts to position and velocity vector representation.

to_classical()
Converts to classical orbital elements representation.

2.6.2 poliastro.iod package

poliastro.iod.izzo module

Izzo's algorithm for Lambert's problem

poliastro.iod.izzo.**lambert**(*k*, *r0*, *r*, *tof*, *M=0*, *numiter=35*, *rtol=1e-08*)
Solves the Lambert problem using the Izzo algorithm.

New in version 0.5.0.

Parameters

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

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

poliastro.iod.vallado module

Initial orbit determination.

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

New in version 0.3.0.

Parameters

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

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

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

2.6.3 poliastro.neos package

Code related to NEOs.

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

Notes

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

poliastro.neos.dastcom5 module

NEOs orbit from DASTCOM5 database.

`poliastro.neos.dastcom5.`**`asteroid_db`**`()`

Return complete DASTCOM5 asteroid database.

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

Return type numpy.ndarray

`poliastro.neos.dastcom5.`**`comet_db`**`()`

Return complete DASTCOM5 comet database.

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

Return type numpy.ndarray

`poliastro.neos.dastcom5.`**`orbit_from_name`**`(`*name*`)`

Return `Orbit` given a name.

Retrieve info from JPL DASTCOM5 database.

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

Returns **orbit** – NEO orbits.

Return type list (*Orbit*)

`poliastro.neos.dastcom5.`**`orbit_from_record`**`(`*record*`)`

Return `Orbit` given a record.

Retrieve info from JPL DASTCOM5 database.

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

Returns **orbit** – NEO orbit.

Return type *Orbit*

`poliastro.neos.dastcom5.`**`record_from_name`**`(`*name*`)`

Search *dastcom.idx* and return logical records that match a given string.

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

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

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

Return type list (int)

`poliastro.neos.dastcom5.`**`string_record_from_name`**`(`*name*`)`

Search *dastcom.idx* and return body full record.

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

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

Returns **lines** – Body records

Return type list(str)

`poliastro.neos.dastcom5.`**`read_headers`**`()`

Read *DASTCOM5* headers and return asteroid and comet headers.

Headers are two numpy arrays with custom dtype.

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

Return type tuple (numpy.ndarray)

`poliastro.neos.dastcom5.`**`read_record`**`(`*record*`)`

Read *DASTCOM5* record and return body data.

Body data consists of numpy array with custom dtype.

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

Returns **body_data** – Body information.

Return type numpy.ndarray

`poliastro.neos.dastcom5.`**`download_dastcom5`**`()`

Downloads DASTCOM5 database.

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

`poliastro.neos.dastcom5.`**`entire_db`**`()`

Return complete DASTCOM5 database.

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

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

Return type numpy.ndarray

dastcom5 parameters

Dastcom5 parameters

avail:

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

avail	Label	Definition
ac/3+	EPOCH	Time of osc. orbital elements solution, JD (CT,TDB)
ac/3+	CALEPO	Time of osc. orbital elements solution, YYYYDDMM.ffff
ac/3+	MA	Mean anomaly at EPOCH, deg (elliptical & hyperbolic cases "9.999999E99" if not available)

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

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

poliastro.neos.neows module

NEOs orbit from NEOWS and JPL SBDB

`poliastro.neos.neows.`**`orbit_from_spk_id`**(*spk_id*, *api_key='DEMO_KEY'*)

Return `Orbit` given a SPK-ID.

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

Parameters

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

Returns **orbit** – NEA orbit.

Return type *Orbit*

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

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

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

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

Returns **spk_id** – SPK-ID number.

Return type str

`poliastro.neos.neows.orbit_from_name(name, api_key='DEMO_KEY')`

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*

2.6.4 poliastro.bodies module

Bodies of the Solar System.

Contains some predefined bodies of the Solar System:

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

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

Data references can be found in `constants`

class `poliastro.bodies.`**`Body`**(*parent*, *k*, *name*, *symbol=None*, *R=<Quantity 0.0 km>*, ***kwargs*)
Class to represent a generic body.

`__init__`(*parent*, *k*, *name*, *symbol=None*, *R=<Quantity 0.0 km>*, ***kwargs*)
Constructor.

Parameters

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

2.6.5 poliastro.constants module

Astronomical and physics constants.

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

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

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

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

and radii were obtained from:

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

2.6.6 poliastro.coordinates module

Functions related to coordinate systems and transformations.

This module complements `astropy.coordinates`.

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

Parameters

- **`r`** (*`Quantity`*) – Position vector in a body-centered reference frame.
- **`v`** (*`Quantity`*) – Velocity vector in a body-centered reference frame.
- **`source_body`** (`Body`) – Source body.
- **`epoch`** (*`Time, optional`*) – Epoch, default to J2000.
- **`rotate_meridian`** (*`bool, optional`*) – Whether to apply the rotation of the meridian too, default to False.

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

Return type tuple (Quantity)

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

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

Parameters

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

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

Return type tuple (Quantity)

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

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

Parameters

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

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

Return type tuple (Quantity)

2.6.7 poliastro.cli module

Command line functions.

2.6.8 poliastro.ephem module

Planetary ephemerides.

poliastro.ephem.**get_body_ephem**(*body*, *epoch*)

Position and velocity vectors of a given body at a certain time.

The vectors are computed with respect to the Solar System barycenter.

New in version 0.3.0.

Parameters

- **body** (`str`) – Name of the body.
- **epoch** (`Time`) – Computation time. Can be scalar or vector.

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

Return type Quantity

2.6.9 poliastro.examples module

Example data.

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

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

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

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

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

poliastro.examples.**churi = 1 x 6 AU x 7.0 deg orbit around Sun ()**
Comet 67P/Churyumov–Gerasimenko orbit example

2.6.10 poliastro.frames module

Coordinate frames definitions.

class poliastro.frames.**HeliocentricEclipticJ2000**(**args*, ***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).

2.6.11 poliastro.hyper module

Utility hypergeometric functions.

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

2.6.12 poliastro.maneuver module

Orbital maneuvers.

class poliastro.maneuver.**Maneuver**(**impulses*)
Class to represent a Maneuver.

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

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

__init__(**impulses*)
Constructor.

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

Notes

TODO: Fix docstring, *args convention

classmethod **impulse** (*dv*)
Single impulse at current time.

classmethod **hohmann** (*orbit_i*, *r_f*)
Compute a Hohmann transfer between two circular orbits.

classmethod **bielliptic** (*orbit_i*, *r_b*, *r_f*)
Compute a bielliptic transfer between two circular orbits.

get_total_time ()
Returns total time of the maneuver.

get_total_cost ()
Returns total cost of the maneuver.

2.6.13 poliastro.patched_conics module

Patched Conics Computations

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

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

Parameters

- **body** (*~poliastro.bodies.Body*) – Astronomical body which the SOI's radius is computed for
- **a** (*float or None, optional*) – Semimajor Axis of the body's orbit

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

Return type astropy.units.quantity.Quantity

2.6.14 poliastro.plotting module

Plotting utilities.

poliastro.plotting.**plot** (*state*, *label=None*, *color=None*)
Plots a State.

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

class poliastro.plotting.**OrbitPlotter** (*ax=None*, *num_points=100*)
OrbitPlotter class.

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

__init__ (*ax=None*, *num_points=100*)
Constructor.

Parameters

- **ax** (*Axes*) – Axes in which to plot. If not given, new ones will be created.
- **num_points** (*int, optional*) – Number of points to use in plots, default to 100.

set_frame (*p_vec*, *q_vec*, *w_vec*)
Sets perifocal frame if not existing.

Raises

- `ValueError` – If the vectors are not a set of mutually orthogonal unit vectors.
- `NotImplementedError` – If the frame was already set up.

set_attractor (*orbit*)
Sets plotting attractor.

Parameters **orbit** (`Orbit`) – orbit with attractor to plot.

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

2.6.15 poliastro.stumpff module

Stumpff functions.

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

For positive arguments:

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

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

For positive arguments:

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

2.6.16 poliastro.util module

Function helpers.

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

`poliastro.util.`**rotate** (*vector*, *angle*, *axis='z'*, *unit=None*)
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** (*array_like*) – Dimension 3 vector.
- **angle** (*convertible to Angle*) – Angle of rotation.
- **axis** (*str or 3-sequence*) – Either 'x','y', 'z', or a (x,y,z) specifying an axis to rotate about. If 'x','y', or 'z', the rotation sense is counterclockwise looking down the + axis (e.g. positive rotations obey left-hand-rule).

- **unit** (*UnitBase, optional*) – If *angle* does not have associated units, they are in this unit. If neither are provided, it is assumed to be degrees.

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

References

`poliastro.util.transform(vector, angle, axis='z', unit=None)`
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.

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

New in version 0.8.0.

Parameters

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

Returns Array of time values.

Return type Time

2.7 What's new

2.7.1 poliastro 0.8 (Unreleased)

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.

2.7.2 poliastro 0.7.0 - 2017-09-15

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

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

New features:

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

Other highlights:

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

New contributors

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

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

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

Bugs fixed:

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

Backward incompatible changes:

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

2.7.3 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.

2.7.4 poliastro 0.5.0 - 2016-03-06

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

New features:

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

Other highlights:

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

Bugs fixed:

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

Backward incompatible changes:

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

2.7.5 poliastro 0.4.2 - 2015-12-24

Fixed packaging problems.

2.7.6 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.

2.7.7 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.

2.7.8 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.

2.7.9 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.

2.7.10 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.

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

Python Module Index

p

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

V