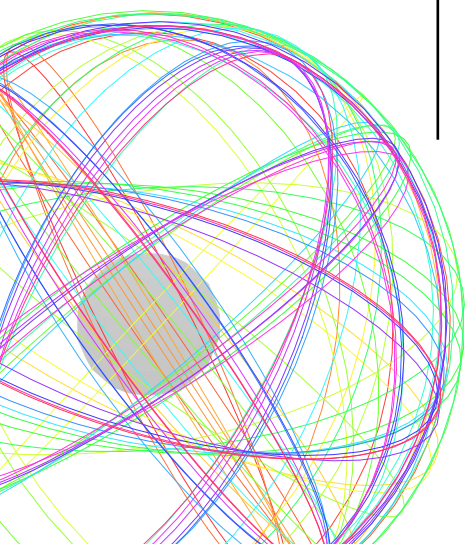
EOS 1.0a1 Documentation

Extendable Orbit System

an orbital mechanics Python library

HAMZA EL-KEBIR
Stud. No.: 4663217



Delft University of Technology, Faculty of Aerospace Engineering

June 15, 2018

Contents

Preface

Throughout my studies, I have been getting increasingly interested in orbital mechanics, especially in locating and tracking spacecraft and celestial bodies. I was, however, surprised by the lack of Python packages that allow for orbit visualization, and orbital mechanics calculations in general. This led me to take up the challenge of developing an easy-to-use Python library which allows for not only orbit visualization, but also for orbital and celestial mechanics calculations. This turned out as EOS, short for 'Extendable Orbit System', named after 'eos', the Latin word for 'dawn'.

This library started out as a personal project, with the goal of plotting 3D Keplerian orbits in an interactive `matplotlib` environment. This turned out to be a rather easy task, leading me to set a number of additional challenges for myself. These included fully modular behavior in adding orbits, along with the ability to view the current location of the orbiting object (an implementation of Kepler's equation). This was soon followed by a full-fledged simulation of the solar system using the ephemeris developed by the Jet Propulsion Laboratory, as well as an orbit visualization of real-life satellites as cataloged in the SPACETRACK database of the United States Air Force.

By means of this documentation, I hope to be able to describe and illustrate the capabilities that are currently available in EOS, and to give a summary of what capabilities are still to come. I hope this library will be of use to many people, and would be grateful to receive any feedback you may have.

HAMZA EL-KEBIR
Delft, the Netherlands
June, 2018

Chapter 1

Introduction

EOS is a versatile Python library that aims to provide a wide variety of orbital mechanics tools, its focus being on effortless visualization in both still and animated form. A brief overview of the various main elements of the library and their dependencies will be introduced in this chapter. The theoretical background and sources of celestial constants used will be described in in Chapter 2. In Chapter 3, the `Eos` class will be documented in detail. Chapter 4 will serve as a showcase of EOS' capabilities by means of several examples. Following this, Chapter 5 will list the functionalities that are to be implemented in future releases. Finally, Chapter 6 will conclude all of the aforementioned.

EOS is principally based on the Kepler's laws of planetary motion, which is applied by default throughout the library. There is, however, also integration of the Simple Generalized Perturbations 4 (SGP4) model, as described by Vallado et al. [1], along with an *n*-body simulation propagator utilizing Newton's laws of gravitational acceleration through numerical integration.

The `Pandas`[1] package is used to order general orbit information and coordinates in the form of DataFrames. These values can easily be accessed through the `kepler` function of the `Eos` class, to allow for effortless manipulation and review of calculation results.

Visualization of the trajectories is achieved by means of the `matplotlib`[2] package. Utilized are mainly the `plot3d`, `scatter` and `text3d` functions. Additional, the `animation` class is used to produce 2D animations of the *n*-body simulations. The `scatter` function is used to allow for a so called *v-line* plot, that visualizes the velocity of the orbiting body by applying a colormap, whereas the `plot3d` function serves as a default for plotting trajectories of uniform color. Finally, the `text3d` function is used to plot markers that designate a.o. the current location, apsides and ascending and descending nodes.

In the following chapters, the role of these dependencies is further elaborated upon. Additionally, the theories that drive the calculations and visualizations will be touched upon accordingly (Chapter 2). Finally, a description of the capabilities of the different functions (Chapter 3) and an overview of the future work will be listed (Chapter 4).

[1] `pandas.pydata.org`
[2] `matplotlib.org`

1.1 Dependencies

EOS is extensively tested on Python 3.6.3, and as such, requires Python 3 to be executed.

Other dependencies are:

- `matplotlib`
- `pycurl`[3]
- `python-sgp4`[4]

[3] pycurl.io
[4] github.com/brandon-rhodes/python-sgp4

Chapter 2

Natural Constants & Theories

This chapters aims to describe the sources of the natural constants used in the EOS, as well as the theories that utilize these constants so as to generate accurate simulations and visualizations. Firstly, the constants used will be described in Section 2.1, with the subsections being dedicated to the fundamental physical constants, planetary properties and ephemeris respectively. Following this, the theories applied are described in Section 2.2.

2.1 Natural Constants

The natural constants applied in this library do not only concern fundamental physical constants, but also the masses, volumes and radii of the various celestial bodies that are present in the Solar System. Additionally, albeit it not so much constant, the ephemerides (models providing the apparent positions of celestial bodies) utilized will be described.

2.1.1 Fundamental Physical Constants

The sole fundamental physical constant utilized in this library is the Newtonian constant of gravitation $\mathcal{G}$. The current best estimate of this value (as of 2018) is

$$\mathcal{G} = 6.674\,08 \pm 0.000\,31 \times 10^{-11}\ \mathrm{m^3\,kg^{-1}\,s^{-2}} \tag{2.1}$$

as found by Mohr et al. [2] under the auspices of the US National Institute of Standards and Technology.

2.1.2 Properties of Celestial Bodies

Following the physical constants, the properties of several main celestial bodies are widely used throughout the library.

The fundamental celestial constants, namely solar, terrestrial and jovian mass parameters and equatorial radii, as defined by Mamajek et al. [3] on behalf of the International Astronomical Union, are as follows:

Table 2.1: IAU (2015) System of Nominal Solar and Planetary Conversion Constants [3]

Meaning	Symbol	Value
Nominal solar mass parameter	$\mathcal{GM}_{☉}^{N}$	$1.327\,124\,4 \times 10^{20}\ \mathrm{m^3\ s^{-2}}$
Nominal jovian mass parameter	$\mathcal{GM}_{♃}^{N}$	$1.266\,865\,3 \times 10^{17}\ \mathrm{m}$
Nominal terrestrial mass parameter	$\mathcal{GM}_{♁}^{N}$	$3.986\,004 \times 10^{14}\ \mathrm{m}$
Nominal solar radius (photospheric)	$\mathcal{R}_{☉}^{N}$	$6.957 \times 10^{8}\ \mathrm{m}$
Nominal terrestrial radius (equatorial)	$\mathcal{R}_{♁}^{N}$	$6.3781 \times 10^{6}\ \mathrm{m}$
Nominal jovian radius (equatorial)	$\mathcal{R}_{♃}^{N}$	$7.1492 \times 10^{7}\ \mathrm{m}$

The remaining masses and radii are also of importance, but these are not defined as constants in contrast to the aforementioned radii and mass parameters. The remaining radii are adapted from Archinal et al. [4], published under the authority of the International Astronomical Union. Similarly, the masses have been adapted from the Solar System Dynamics group at the Jet Propulsion Laboratory [5], with all due credit to the original sources. The lunar radius has been adapted from Roncoli [6], while the lunar mass has been calculated from the current best mass ratio estimate between Moon and Earth [6].

2.1.3 Ephemeris

The ephemeris used in this library is an approximation of the Keplerian elements valid from 3000 BC–3000 AD, as presented by Standish [7]. This is a mathematical fit of the Keplerian elements of the various planets present in the solar system, including corrections for the perturbations experienced by the various bodies. Additional correction factors that account for gravitational perturbations due to the heavier bodies are also present for Jupiter, Saturn, Uranus, Neptune and Pluto.

2.2 Theories

The core part of EOS is driven by Kepler's equations of planetary motion. The obtained cylindrical coordinates are subsequently converted to Cartesian coordinates. Along with the basic functions of the core `Eos` class, this coordinate conversion will be described. The `Nbody` class is driven by Newton's laws, which will be briefly touched upon. Finally, the Simple General Perturbations 4 (SGP4) model will be briefly described, as well as the conversion of NORAD Two-line Elements (TLEs) to Keplerian elements.

2.2.1 Kepler's equation's of motion

Given the mass parameter the central body ($\mu = \mathcal{G}M$), the orbit may be calculated when given the primary orbital (Keplerian) elements. These comprise:

Table 2.2: Primary orbital elements with abbreviations and symbols

Meaning	**Abbreviation**	**Symbol**
Eccentricity	EC	e
Semimajor axis	A	a
Inclination	IN	i
Longitude of the ascending node	W	Ω
Argument of periapsis	OM	ω
True anomaly	TA	θ

All of these are mandatory, except for the true anomaly, which is only needed if the current position is to be known. The abbreviations provided are based on those used in the SPACETRACK program of the United States Air Force (USAF) and the SGP4 code, and are analogous to those used in the library.

The angles described above, i.e. inclination, longitude of the ascending node, argument of periapsis and the true anomaly may be illustrated as per figure 2.1.

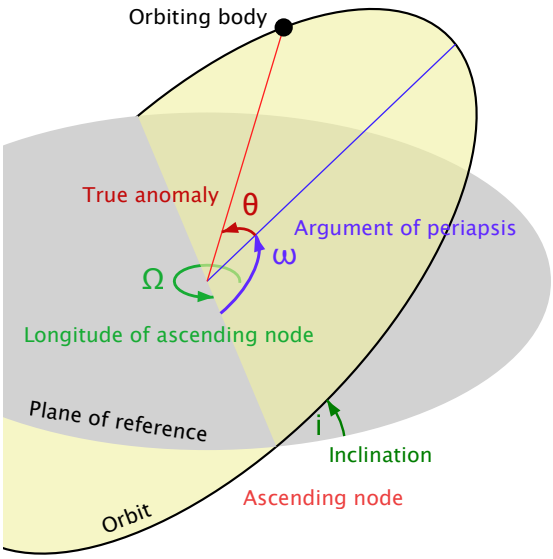


Figure 2.1: Definition of the Keplerian angles, adapted from [8]

Secondary to primary element conversion

The user need not necessarily input all primary orbital elements, as the `Eos` class is capable of deriving these from several secondary orbital elements. The only prerequisite for this is that the mass parameter μ and the central body's radius R are known. The secondary elements are:

Table 2.3: Secondary orbital elements with abbreviations and symbols

Meaning	**Abbreviation**	**Symbol**
Apoapsis	AP	Q
Apoapsis altitude	HAP	h_Q
Periapsis	PE	q
Periapsis altitude	HPE	h_q
Period	T	T
Semimajor axis altitude	HA	h_a
Mean anomaly	MA	M
Mean motion	N	n
Time since periapsis passage	TP	τ

The core class allows for the following conversions of these secondary orbital elements:

$$\begin{aligned} f(h, R) &\rightarrow Q \text{ or } q \text{ or } a \\ f(M) &\rightarrow \theta \\ f(Q, q) &\rightarrow e, a \\ f(\tau) &\rightarrow \theta \end{aligned}$$

The conversion from τ to θ utilizes the following relationship:

$$\sqrt{\frac{a}{\mu^3}}(E - e\sin E) - \tau = 0 \tag{2.2}$$

For which μ is the mass parameter and E the eccentric anomaly. $\sqrt{a/\mu^3}$ is also known as the mean motion n. Scipy's `optimize.fsolve` function is used to minimize the above function, so as to obtain the eccentric anomaly. The same procedure is applied to convert the mean anomaly M to E using the following formula:

$$(E - e\sin E) - M = 0 \tag{2.3}$$

E is subsequently converted to θ by:

$$\theta = 2\arctan\left(\sqrt{\frac{1+e}{1-e}}\tan(E/2)\right) \tag{2.4}$$

Primary to secondary element conversion

The inverse of the operations described above are also of importance. In this case the following conversions from primary elements are possible:

$$\begin{aligned} f(a, e) &\rightarrow Q \text{ or } q \text{ or } a \text{ or their altitude forms} \\ f(\theta, e) &\rightarrow E \\ f(E, e) &\rightarrow M \\ f(a) &\rightarrow n \\ f(n) &\rightarrow T \\ f(M, n) &\rightarrow \tau \end{aligned}$$

The formulae associated with these conversions are as follows. For apoapsis and periapsis calculations:

$$Q = (1 + e)a \; ; \; q = (1 - e)a \tag{2.5}$$

Altitude h conversions from radial length r, require the central body's radius R:

$$h = r - R \tag{2.6}$$

Mean anomaly is calculated using the eccentricity and eccentric anomaly:

$$M = E - e \sin E \tag{2.7}$$

Mean motion is derived using the semimajor axis:

$$n = \sqrt{\frac{\mu}{a^3}} \tag{2.8}$$

The orbital period is calculated using the mean motion:

$$T = \frac{2\pi}{n} \tag{2.9}$$

Finally, the time from periapsis is obtained using both the mean motion and mean anomaly:

$$\tau = \frac{M}{n} \tag{2.10}$$

Coordinate transformation

After determination of the primary orbital elements, the Cartesian coordinates may be calculated. To this end, the radius as a function of true anomaly must first be obtained:

$$r(\theta) = \frac{a\left(1 - e^2\right)}{1 + e\cos\theta} \tag{2.11}$$

Following this, the position vector $\vec{o}$ and velocity vector $\dot{\vec{o}}$ in the orbital frame may be obtained, as described by Schwarz [9]:

$$\vec{o}(\theta) = r(\theta)\begin{bmatrix}\cos\theta \\ \sin\theta \\ 0\end{bmatrix} \tag{2.12}$$

$$\dot{\vec{o}}(\theta) = \frac{\sqrt{\mu a}}{r(\theta)}\begin{bmatrix}-\sin E \\ \sqrt{1 - e^2}\cos E \\ 0\end{bmatrix} \tag{2.13}$$

These coordinates must, however, still be converted to a bodycentric Cartesian coordinate system. The transformation matrix for this operation was derived individually using rotational matrix transformations:

$$A = \begin{bmatrix} \cos(\omega)\cos(i)\cos(\Omega) - \sin(\omega)\sin(\Omega) & -\sin(\omega)\cos(i)\cos(\Omega) - \cos(\omega)\sin(\Omega) & 0 \\ \cos(\omega)\cos(i)\sin(\Omega) + \sin(\omega)\cos(\Omega) & -\sin(\omega)\cos(i)\sin(\Omega) + \cos(\omega)\cos(\Omega) & 0 \\ -\cos(\omega)\sin(i) & \sin(\omega)\sin(i) & 0 \end{bmatrix} \tag{2.14}$$

The position and velocity vectors are both ultimately function of the true anomaly θ, while the transformation matrix is a function of the inclination i, right ascension ω and longitude of the ascending node Ω. Finally, the state vectors may be left-multiplied with the transformation vector to give the state vectors in bodycentric Cartesian coordinates:

$$\vec{r}(\theta) = A\vec{o}(\theta) \tag{2.15}$$

$$\dot{\vec{r}}(\theta) = A\dot{\vec{o}}(\theta) \tag{2.16}$$

One must note that most approaches take $\vec{r}$ as a function of time t, but this necessitates additional calculations that are not deemed worthwhile when aiming for orbit visualization. Moreover, the local time from periapsis at every calculated point is obtained afterwards, using much less labor-intensive formulae that do not require numerical optimization (cf. equation 2.2).

2.2.2 N-body Simulations

N-body simulations are enabled by the `Nbody` class. This class also allows for on-the-spot animated visualizations of the simulation results. The backbone of this class lies in the numerical integration of Newton's law of universal gravitation:

$$\vec{F}_{2\to1} = \mathcal{G}\frac{M_1 M_2}{r^2}\hat{\vec{r}}_{2\to1} \tag{2.17}$$

With M_1 and M_2 being the masses of two bodies, r the radial distance between the centers of mass (CoM), and $\hat{\vec{r}}_{2\to1}$ the unit vector from the CoM of the second body to the CoM of the first body.

As we are dealing with a numerical integration starting from acceleration, we may apply the fact that:

$$\vec{a}_{2\to1} = \frac{\vec{F}_{2\to1}}{M_2} = \mathcal{G}\frac{M_1}{r^2}\hat{\vec{r}}_{2\to1} \tag{2.18}$$

Through numerical integration of a using a sufficiently small time step Δt, the position at a certain point in time may be obtained.

For a body n attracted by body $n-1$, with initial conditions $\vec{v_{n,0}}$ and $\vec{x_{n,0}}$, the following integration may be applied.

$$\vec{a}_{n,1} = \mathcal{G}\frac{M_{n-1}}{r^2}\hat{\vec{r}}_{n\to n-1} \tag{2.19}$$

$$\vec{v}_{n,1} = \vec{v}_{n,0} + \frac{\vec{F}_{n\to n-1,1}}{M_n}\Delta t \tag{2.20}$$

$$\vec{x}_{n,1} = \vec{x}_{n,0} + \vec{v}_{n,1}\Delta t \tag{2.21}$$

Generalizing this for a body p, part of the set of bodies N at time step j:

$$\vec{a}_{p,j} = \sum_{i\in N\setminus\{p\}}^{n} \mathcal{G}\frac{M_i}{r^2}\hat{\vec{r}}_{a\to i} \tag{2.22}$$

$$\vec{v}_{p,j} = \vec{v}_{n,j-1} + \vec{a}_{p,j}\Delta t \tag{2.23}$$

$$\vec{x}_{p,j} = \vec{x}_{n,j-1} + \vec{v}_{p,j}\Delta t \tag{2.24}$$

This recurrence relation forms the basis of any n-body simulation. This relation is applied to every body $p \in N$. The results at every time step j are stored to enable plotting of the position at time $t = j\Delta t$ and the points traversed before t.

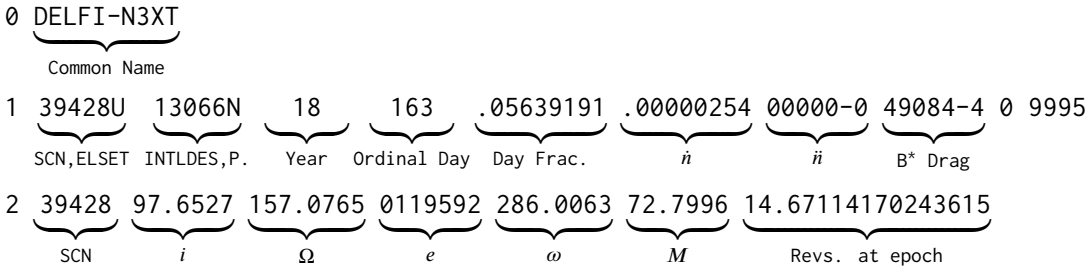
2.2.3 SPACETRACK TLEs & SGP4

The forte of this library, is that it allows for direct orbit calculation and visualization of real-life Earth-orbiting satellites, as tracked in the SPACETRACK catalog [10]. SPACETRACK is a declassified mission of the United States Air Force (USAF), currently performed by the Joint Space Operations Center (JSOC) at Vandenberg AFB. The first satellite to have been tracked, was the Soviet Sputnik I in 1957, and operations have continued ever since. Through use of advanced tracking techniques, satellite position and velocity ephemerides are supplied nearly daily to the general public.

Using the SPACETRACK REST API and `pycurl`, the raw Two-Line Elements (TLEs) of all satellites are downloaded and store for further processing. In a later paragraph, position determination through the Simple General Perturbations 4 (SGP4) model will be described, but for now the focus lies on extracting Keplerian elements from the TLEs. An example of a TLE is as follows:

```
0 DELFI-N3XT
1 39428U 13066N 18163.05639191 .00000254 00000-0 49084-4 0 9995
2 39428 97.6527 157.0765 0119592 286.0063 72.7996 14.67114170243615
```

These three lines consist of the following elements:



For which SCN is the NORAD Satellite Catalog Number, INTLDES the Committee on Space Research's (COSPAR) International Designator along with P, being the launch piece designator (A--Z). ELSET is the classification of the object (almost always U for unclassified). The B^*-term is a drag term that is used in SGP4 to account for atmospheric conditions, more on which will follow later. Terms that have not been annotated, are used only for internal analysis and serve no real meaning to the end-user.

Many of the constituents of TLEs are encoded in a variety of ways to reduce the space they require (TLEs have a fixed length). The first and second derivate of mean motion ($\dot{n}$ and $\ddot{n}$) have units of revolutions per day, and have been divided by 2 and 6 respectively. A somewhat more sophisticated method has been employed for the B^*-term, which has units of (earth radii)$^{-1}$. This term is further more encoded by ways of a convention inherited from FORTRAN. That is to say:

$$\texttt{49084-4} = 0.49084 \cdot 10^{-4} \ \left(\mathcal{R}_{\delta}^{N}\right)^{-1} \tag{2.25}$$

The above mentioned caveats are only the tip of the iceberg. More information may be found in Kelso [11].

TLE processing

The getTLE function is capable of retrieving the most recent TLEs, Satellite Catalog (SATCAT) and launch site list. This is done by accessing the SPACETRACK API using pycurl, after entering the credentials stored in eos_config.py. These raw elements are stored as .3le and .csv files. The .3le file is read by python, after which it is manually parsed using the parseTLE function. Both the converted and raw elements are store as Pandas DataFrames for later access. The .csv files subsequently imported as DataFrames using Pandas' read_csv function.

SGP4

The Simple General Perturbations 4 (SGP4) model is used in combination with the aforementioned TLEs. This model accounts for perturbations caused by the nonuniformity of the Earth and the influence of the Earth's atmosphere on a satellites orbit (B^*-term corrections).

To account for the irregular gravitational field and oblateness of the Earth, the 1984 World Geodetic System (WGS84) is used [1]. This model comprises an improved gravitational and geometric model of the Earth. Although WGS72 (the 1972 version) is still the de facto model used in SGP4, WGS84 is steadily taking over its role [11]. For this reason, EOS uses the WGS84 model.

Initially it was attempted to code the SGP4 model following the formulae as found in the original manual. However, as SGP4 is rather complex, it is very prone to computational inefficiencies if not coded optimally. For this reason, EOS uses the `python-sgp4` package by Rhodes [12] to do the heavy lifting. This package is a one-on-one Python port of the C++ version, as presented by Vallado et al. [1]. This C++ version is, on its turn, based on the original FORTRAN implementation.

All in all, the combination of SPACETRACK TLEs and SGP4 is known to produce positional errors in the range of $1\,\mathrm{km\,d^{-1}}$ to $3\,\mathrm{km\,d^{-1}}$ (kilometers per day) [1], which makes for an excellent means of determining satellite positions, as TLEs are updated as often as every other quarter day [10].

Chapter 3

Classes & Functions

The EOS library consists of several independent classes, most of which have a certain degree of intercompatibility. The default EOS release directory has the following layout:

```
/release
├── assets
│   ├── current.3le
│   ├── launchsite.csv
│   ├── satcatcurrent.csv
│   └── satcatrecent.csv
├── eos
│   ├── __init__.py
│   ├── eos_aedata.py
│   ├── eos_config.py
│   ├── eos_conv.py
│   ├── eos_core.py
│   └── nbody.py
├── eos_core_showcase.py
└── nbody_showcase.py
```

EOS depends on several ancillary files for nominal operation. These include the `eos_aedata.py` and `eos_config.py` files. The `eos_aedata` is a storage medium containing the various natural constants, planetary properties and ephemerides as used by EOS.`eos_config` on the other hand, contains the login credentials as used by the `getTLE` function to access the SPACETRACK API. The structure of this file will be elaborated on later in this document.

The remainder of this chapter will briefly deal with the various classes and the functions they contain, in the order displayed in the directory tree above.

3.1 Eos class

The Eos class forms the core of EOS. It contains all necessary algorithms for orbit calculation and propagation, as well as the visualization system based on matplotlib. This class resides in the eos_core.py file and has the following structure (in alphabetic order):

Eos
- degrees ..... Convert orbital elements from degrees to radians
- getTLE ..... Downloads latest TLEs and SATCAT from SPACETRACK
- init ..... Initializes core functionality (orbit center)
- kepler ..... Calculates and plots orbits defined using orbit
- M_solve ..... Optimalization function for $M \rightarrow E$ calculation
- orbit ..... Defines new orbit and converts secondary orbital elements
- parseTLE ..... Reads and parses TLE and SATCAT databases
- satorbit ..... Calculates and plot satellite orbit as found in SATCAT
- sgp4 ..... Calculates satellite position at specified time. Requires raw TLE strings
- solarsystem .... Calculates and plots positions of major celestial bodies in Solar System
- ta2t ..... Calculates local time since epoch, taking true anomaly as input
- tp_solve ..... Optimalization function for $\tau \rightarrow E$ calculation
- units ..... Convert input values to output values with output unit
- view ..... Views or saves added orbits
- view_init ..... Initializes matplotlib figure and axes

Henceforth, the main functions will be described and illustrated by examples. Documentations of secondary functions will follow in a later release.

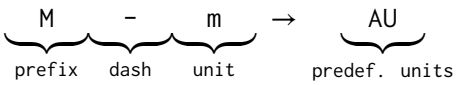
3.1.1 orbit function

The orbit function takes the following parameters as default input:

```
(self, a=None, h_a=None, e=None, ta=None, tp=None, i=0, om=0, w=0, Q=None,
h_Q=None, q=None, h_q=None, unit_i='km', unit_o='km', deg=True, body='Earth')
```

In addition to the elements described in Subsection 2.2.1, this function also allows the user to choose whether or not degrees are used (deg bool). The central body is specified using Earth, and only takes predefined bodies at the moment. The input and output units may also be defined (unit_i and unit_o).

Unit conversion is handled by the conv function imported from eos_conv.py. In general, the following structure for unit input has been defined:



Prefixes are separated from units by a dash (-), whereas several mainstream units, such as km, are also allowed as 'predefined units'. The astronomical unit is one such units. Other predefined units supported are amongst others: jovian radii, earth radii, nautical miles, etc. Both SI units and US

Customary units are fully supported. A more exhaustive list of supported units will follow in a later release. In the meantime, please consult the `eos_conv.py` for a direct overview.

3.1.2 `view` function

The `view` function invokes the `matplotlib` `plt.show()` function after defining axes with an aspect ratio of one, as well as a legend and titles. Furthermore, the central body may be customized. The default options of this function are:

```
(self, x=9, y=9, el=30, az=45, cust_txt=False, bodysurface=False,
bodycolor=None, bodyalpha=0.5, show=True, save=False, name='EOS Output',
extension='pdf', legend=False)
```

The options containing the `body` prefix allow for toggling use of a surface (default is wireframe), choosing a custom color and changing the transparency (alpha). The elevation (`el`) and azimuth (`az`) angle may also be altered. Furthermore, a custom title text (`cust_txt`) may be defined. Finally, the end result may be shown or saved, with the option to display a legend and change the output name and extension.

3.1.3 `kepler` function

After specifying an orbit using the `orbit` function, the `kepler` function may be called. This function propagates the most recently defined orbit, allowing for different resolutions and visualization options:

```
(self, rev=1, data=False, vis='v-line', marker=True, curr_background=True,
curr_marker=True, curr_symbol='', curr_color='#D62829',
curr_label='_nolegend_', curr_fontdict=None, linewidth=1.5, sgp=False,
l1=None, l2=None, time='now', designation='NA', res=250)
```

The `rev` variable defines the amount of revolutions to be calculated. The `vis` option takes either `None`, for no visualization, `v-line` for a velocity gradient line, `gradient` for a fading gradient shade or any other color that `matplotlib` can handle (e.g. `'green'`)

The `curr`-prefix marks functions that deal with the current marker position, which is controlled by the `marker` boolean. The `data` boolean allows the user to toggle raw orbit data to be returned when the function is called.

Molniya Orbit Example

An example plotting a Soviet Molniya orbit will follow:

Example Code 3.1: Molniya orbit example

```
from eos.eos_core import Eos

eos1 = Eos()
eos1.view_init()
eos1.orbit(a=26554, e=0.72, tp=4000, i=63.4, om=0, w=-90, unit_i='km',
 ↪  unit_o='km', body='Earth')
eos1.kepler(vis='v-line', data=True, marker=True, curr_marker=True)
eos1.view(x=8,y=8,bodysurface=True,bodyalpha=0.2,cust_txt='Molniya Orbit
 ↪  Visualization')
```

This results in the following output (Fig. 3.1):

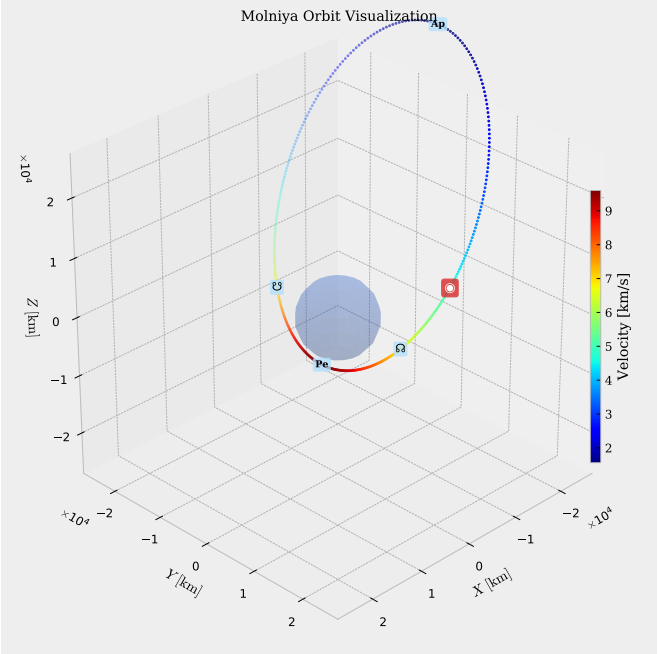


Figure 3.1: Molniya orbit visualization (code 3.1)

In addition to this visualization, notice how the `data` option has been enabled. Data is ordered in a dictionary of `DataFrames` with the following structure:

DATA
- GNRL
- ORBT
- CRDS
- SPEC

An more exhaustive data structure can be found in Appendix A.

3.1.4 `satorbit` function

Now that the `kepler` function has been described, we may look at the `satorbit` function. This function does, however, require the following steps to be performed:

`view_init` → (`getTLE`) → `parseTLE` → `satorbit`

As indicated above, `getTLE` is optional, as one may wish to not update to the latest TLEs. The capabilities of `satorbit` will be illustrated by two examples.

Sentinel Orbit Search Example

The first example (code 3.2) demonstrates a search by `SATNAME`, for which the `cont` option is enabled, meaning the results may partially of the query and need not be exactly the same. Several `nocont` (not containing) strings have also been defined. Most common is to exclude `'R/B'` (rocket bodies) and `'DEB'` (debris). Finally, SGP4 propagation is disabled, and both the orbit and current position are shown.

Example Code 3.2: `satorbt` multiple search example

```
from eos.eos_core import Eos

eos1 = Eos()
eos1.view_init()
eos1.parseTLE()
eos1.satorbit(SATNAME=['SENTINEL'], cont=True, nocont=['DEB','R/B'],
↪  res=25, nums=False, info=True, marker=False, abbrev=False,
↪  vline=False, label='SATNAME', legend=True, bg=False, orbit=True,
↪  orbitcolor=None, markercolor=None, SGP=False, curr_marker=True)
eos1.view(x=8, y=8, bodysurface=True, bodyalpha=0.2,
↪  bodycolor='white', cust_txt='Sentinel Sattelite Orbits')
```

This results in the following output (Fig. 3.2):

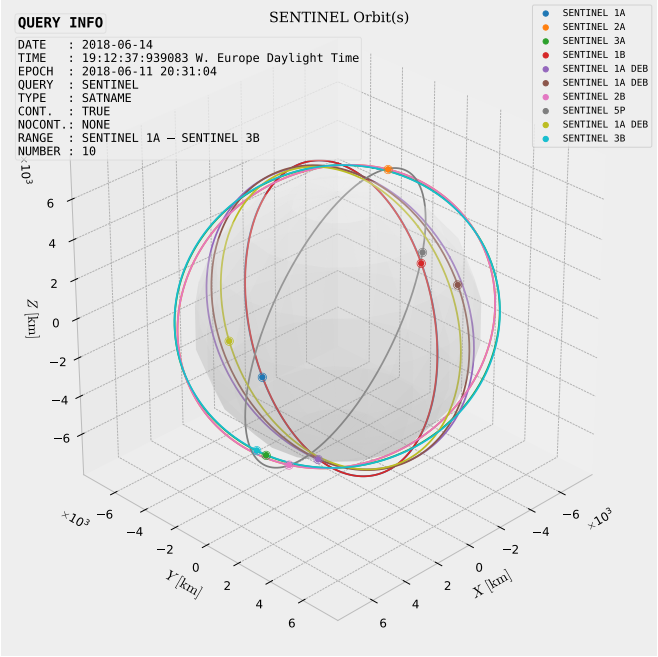


Figure 3.2: ESA's Sentinel satellite orbits (code 3.2)

Vanguard 1 SGP4 Example

If one wishes to get the true current position as calculated using SGP4, the following example (code 3.3) will present a solution. It must, however, be noted that SGP4 uses a different coordinate system than EOS; the X-axis is aligned with the vernal equinox, while the Z-axis is aligned with the Geographic North Pole. Correction for this fact will be implemented in a later release.

Example Code 3.3: `satorbt` SGP4 example

```
from eos.eos_core import Eos

eos1 = Eos()
eos1.view_init()
eos1.parseTLE()
eos1.satorbit(SATNAME=['VANGUARD 1'], cont=True, nocont=['DEB','R/B'],
↪  res=25, nums=False, info=True, marker=False, abbrev=False, vline=False,
↪  label='SATNAME', legend=True, bg=False, orbit=False, orbitcolor=None,
↪  markercolor=None, SGP=True, curr_marker=True)
eos1.view(x=8, y=8, bodysurface=True, bodyalpha=0.2, bodycolor='white',
↪  cust_txt='VANGUARD 1 Current Position')
```

Which yields the following visualization (Fig. 3.3):

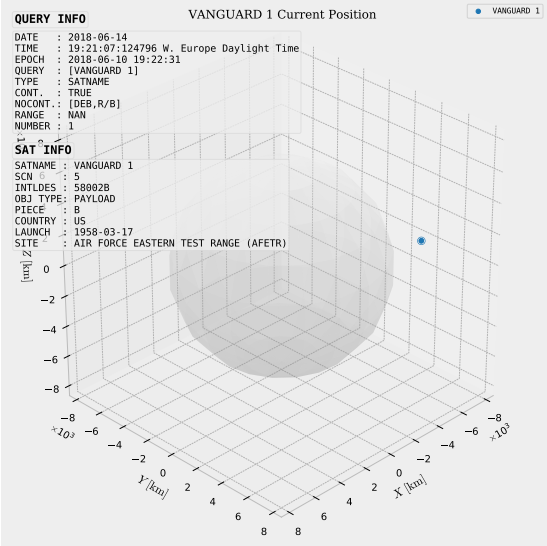


Figure 3.3: Current Vanguard 1 position (code 3.3)

As may been seen in the figure above, `satorbit` plots with a single satellite display additional satellite properties. The SGP4 propagation may be applied at larger scale, for example to visualize the current position of all space debris (Fig. 3.4):

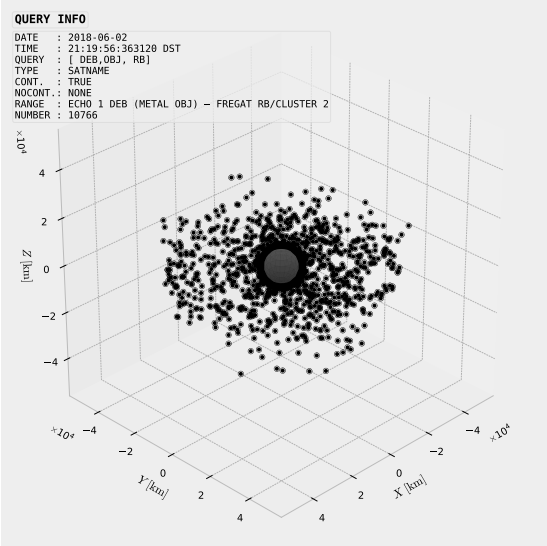


Figure 3.4: Space debris visualization

3.1.5 `solarsystem` function

The `solarsystem` function is fairly simple. It only requires a `datetime` object for its variable `time`. `time` may also be `'now'`, for the current time. After calculation, the current locations and orbits of the various planets are stored in a list called `self.ssdata`. `solarsystem` is best illustrated by means of an example.

Solar System Example

This example will serve to illustrate `solarsystem`'s intercompabtibility with the aforementioned functions. In addition to plotting the solar system, an additional orbit will be displayed.

Example Code 3.4: `solarsystem` example

```
from eos.eos_core import Eos

eos1 = Eos()
eos1.view_init(center='Earth')
eos1.solarsystem(now=True,view=True,res=50,unit='AU',sphere=True)
eos1.orbit(q=1.35,Q=5.4,i=79.11,unit_i='AU',unit_o='AU',body='Sun')
eos1.kepler(vis='v-line',res=200,curr_marker=False,curr_label='Ulysses')
eos1.view(x=8, y=8,cust_txt='Ulysses Orbit',legend=True)
```

When manually changing the view, the following result may be obtained (Fig. 3.5)

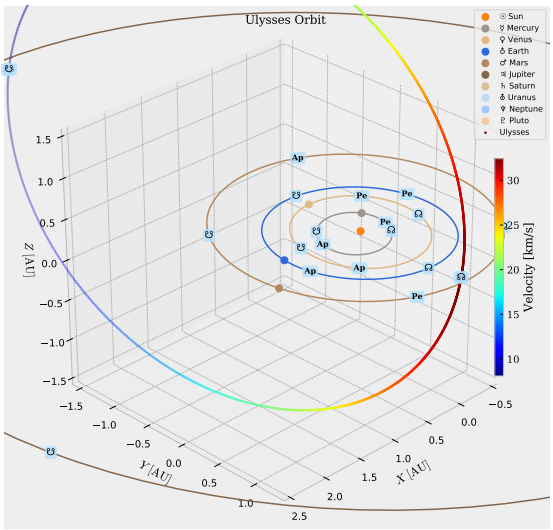


Figure 3.5: Ulysses orbit as seen with Earth in center (code 3.4)

3.2 Nbody & Body classes

In order to allow for n-body simulations, to classes have been constructed: the Nbody and Body class. The Body class serves to define bodies that will be propagated by the Nbody class. A Body object is essentially a storage devices for the body's properties and position over time. Its variable structure of such an object is as roughly follows:

```
Body
├── color
├── impulse
├── mass
├── name
├── objs
├── posx
├── posy
├── posz
└── radius
```

The various pos variables are lists that hold the different coordinate positions at every time step. The objs variable holds all other Body objects with which the current body will react. Finally, the impulse variable contains a list with the various $\Delta\vec{V}$ impulses and the time at which they take place.

The Nbody class acts as a mediator between the various Body objects and propagates them for a specified amount of frames, each frame having a time step dt. The setup is roughly as follows:

1. Define Body objects
2. Add impulses using the addimpulse method
3. Define Nbody object
4. Add list of Body objects to Nbody using Nbody.addobj
5. Link the objects using Nbody.connectobjs
6. Animate (propagate & visualize) using Nbody.animate

3.2.1 Solar System Example

The following example (code 3.5) will show a collaboration between the Eos and Nbody classes to realize a life Solar System simulation.

Example Code 3.5: Nbody example using Eos' solarsystem

```
from eos.eos_core import Eos
from eos.nbody import Nbody,Body
from datetime import datetime

frame0 = 0
interval = 1
```

```
time = datetime.datetime.now()
frames = 5000
timestep = 10*24*3600
lims = [[-35*AU,35*AU], [-35*AU,35*AU]]

eos1 = Eos()

eos1.solarsystem(time='now',view=False,res=1)

Mercury_dat = eos1.ssdata[0]['SPEC']
Venus_dat = eos1.ssdata[1]['SPEC']
Earth_dat = eos1.ssdata[2]['SPEC']
Mars_dat = eos1.ssdata[3]['SPEC']
Jupiter_dat = eos1.ssdata[4]['SPEC']
Saturn_dat = eos1.ssdata[5]['SPEC']
Uranus_dat = eos1.ssdata[6]['SPEC']
Neptune_dat = eos1.ssdata[7]['SPEC']
Pluto_dat = eos1.ssdata[8]['SPEC']

Sun = Body([0.,0.,0.], [0.,0.,0.], dict_R['Sun'], dict_M['Sun'],
↪  'Sun',dict_c['Sun'])

Mercury = Body([Mercury_dat['X'][0]*1e3, Mercury_dat['Y'][0]*1e3,
↪  Mercury_dat['Z'][0]*1e3], [Mercury_dat['VX'][0]*1e3,
↪  Mercury_dat['VY'][0]*1e3, Mercury_dat['VZ'][0]*1e3], dict_R['Mercury'],
↪  dict_M['Mercury'], 'Mercury',dict_c['Mercury'])

Venus = Body([Venus_dat['X'][0]*1e3, Venus_dat['Y'][0]*1e3,
↪  Venus_dat['Z'][0]*1e3], [Venus_dat['VX'][0]*1e3, Venus_dat['VY'][0]*1e3,
↪  Venus_dat['VZ'][0]*1e3], dict_R['Venus'], dict_M['Venus'],
↪  'Venus',dict_c['Venus'])

Earth = Body([Earth_dat['X'][0]*1e3, Earth_dat['Y'][0]*1e3,
↪  Earth_dat['Z'][0]*1e3], [Earth_dat['VX'][0]*1e3, Earth_dat['VY'][0]*1e3,
↪  Earth_dat['VZ'][0]*1e3], dict_R['Earth'], dict_M['Earth'],
↪  'Earth',dict_c['Earth'])

Mars = Body([Mars_dat['X'][0]*1e3, Mars_dat['Y'][0]*1e3,
↪  Mars_dat['Z'][0]*1e3], [Mars_dat['VX'][0]*1e3, Mars_dat['VY'][0]*1e3,
↪  Mars_dat['VZ'][0]*1e3], dict_R['Mars'], dict_M['Mars'],
↪  'Mars',dict_c['Mars'])

```

```
36 Jupiter = Body([Jupiter_dat['X'][0]*1e3, Jupiter_dat['Y'][0]*1e3,
37 Jupiter_dat['Z'][0]*1e3], [Jupiter_dat['VX'][0]*1e3,
   ↪ Jupiter_dat['VY'][0]*1e3, Jupiter_dat['VZ'][0]*1e3], dict_R['Jupiter'],
   ↪ dict_M['Jupiter'], 'Jupiter',dict_c['Jupiter'])
38
39 Saturn = Body([Saturn_dat['X'][0]*1e3, Saturn_dat['Y'][0]*1e3,
   ↪ Saturn_dat['Z'][0]*1e3], [Saturn_dat['VX'][0]*1e3,
   ↪ Saturn_dat['VY'][0]*1e3, Saturn_dat['VZ'][0]*1e3], dict_R['Saturn'],
   ↪ dict_M['Saturn'], 'Saturn',dict_c['Saturn'])
40
41 Uranus = Body([Uranus_dat['X'][0]*1e3, Uranus_dat['Y'][0]*1e3,
   ↪ Uranus_dat['Z'][0]*1e3], [Uranus_dat['VX'][0]*1e3,
   ↪ Uranus_dat['VY'][0]*1e3, Uranus_dat['VZ'][0]*1e3], dict_R['Uranus'],
   ↪ dict_M['Uranus'], 'Uranus',dict_c['Uranus'])
42
43 Neptune = Body([Neptune_dat['X'][0]*1e3, Neptune_dat['Y'][0]*1e3,
   ↪ Neptune_dat['Z'][0]*1e3], [Neptune_dat['VX'][0]*1e3,
   ↪ Neptune_dat['VY'][0]*1e3, Neptune_dat['VZ'][0]*1e3], dict_R['Neptune'],
   ↪ dict_M['Neptune'], 'Neptune',dict_c['Neptune'])
44
45 Pluto = Body([Pluto_dat['X'][0]*1e3, Pluto_dat['Y'][0]*1e3,
   ↪ Pluto_dat['Z'][0]*1e3], [Pluto_dat['VX'][0]*1e3, Pluto_dat['VY'][0]*1e3,
   ↪ Pluto_dat['VZ'][0]*1e3], dict_R['Pluto'], dict_M['Pluto'],
   ↪ 'Pluto',dict_c['Pluto'])
46
47 objs = [Sun,Mercury,Venus,Earth,Mars,Jupiter,Saturn,Uranus,Neptune,Pluto]
48
49 nbdy = Nbody(timestep=timestep, frames=frames, frame0=frame0,
   ↪ interval=interval, lims=lims, time=time)
50 nbdy.addobj(objs)
51 nbdy.connectobjs()
52 nbdy.animate()
```

This is a somewhat more complex example, as it utilizes the stored `ssdata` list to retrieve the current location and velocity vector of all planets. The rest should be evident from the example. A more in-depth example of an *n*-body simulation can be found in the `lunartrajectory.pdf` document in the installation folder.

Chapter 4

Future Work

In future releases, the features mentioned in the following lists are hoped to be gradually implemented into EOS. These are sorted by the amount of time they require to be implemented (short to long-term goals):

1. Greenwich mean and apparent sidereal time calculation (GMST and GAST) for correct longitude placement
2. Addition of a text-based interface (TUI) in `asciimatics` for ease-of-access
3. Ground track visualization (WIP)
4. Real-time satellite position update (blit)
5. Satellite field of view visualization
6. Improved planet visuals (satellite imagery)
7. Support for moon ephemerides (JPL SSD ephemeris)
8. Implementation of JPL's Center of Near Earth Object Studies' (CNEOS) NEO database
9. Integration of NASA's Trajectory Database to allow for interplanetary flight simulations
10. Integration of JPL HORIZONS ephemeris database for precise planet, moon and asteroid positioning
11. Implementation of Earth gravitational model
12. Realization of a hybrid simulation system utilizing both Kepler's equations as well as n-body simulation
13. Implementation of patch-conic approximation and spheres of influence to alleviate computational load
14. Ability to calculate least-energy interplanetary trajectories and construct ΔV porkchop plots
15. Functionality to analyze planetary positions as seen in the night sky at specified location on planet
16. Addition of GAIA star catalog for star tracking
17. Simulation of launch phase, perhaps with simple aerodynamic model
18. Improved visuals with addition of sunlight, shadow, nightsky and atmosphere
19. Port of the entire script to C++ and implement OpenGL visualization to allow for improved versatility

20. Generation of a GUI using Qt5, allowing for real-time operation

Appendix A

`kepler` Data Structure

A detailed data structure of the output of the `kepler` function can be found below. Do note, however, that `SPEC` (specified point) data is only included if either the true anomaly or time from periapsis is specified.

- DATA
  - GNRL
    - Body Mass
    - Body Radius
    - Celestial Object
    - Designation
    - Graviational Parameter
    - Unit
  - ORBT
    - Apoapsis Altitude
    - Apoapsis Radius
    - Apoapsis Velocity
    - Argument of pereapsis
    - Ascending node longitude
    - Eccentricity
    - Inclination
    - Periapsis Altitude
    - Periapsis Radius
    - Periapsis Velocity
    - Period
    - Semi-latus Rectum
    - Semi-major Axis
    - Unit
  - CRDS
    - Angular Velocity
    - Color

```
    ├── Radius
    ├── Revolution
    ├── Time
    ├── True Anomaly
    ├── Velocity
    ├── X
    ├── Y
    └── Z
└── SPEC
    ├── Eccentric Anomaly
    ├── Escape Velocity
    ├── Mean Anomaly
    ├── Mean Motion
    ├── Radius
    ├── Time from Periapsis
    ├── Time to Periapsis
    ├── True Anomaly
    ├── VX
    ├── VY
    ├── VZ
    ├── Velocity
    ├── X
    ├── Y
    └── Z
```

Bibliography

[1] David Vallado, Paul Crawford, Ricahrd Hujsak, and T. S. Kelso. Revisiting spacetrack report #3. In *AIAA/AAS Astrodynamics Specialist Conference and Exhibit*. American Institute of Aeronautics and Astronautics, Aug 2006. doi: 10.2514/6.2006-6753. URL https://doi.org/10.2514/6.2006-6753.

[2] Peter J. Mohr, David B. Newell, and Barry N. Taylor. CODATA recommended values of the fundamental physical constants: 2014. *Journal of Physical and Chemical Reference Data*, 45(4):043102, Dec 2016. doi: 10.1063/1.4954402. URL https://doi.org/10.1063/1.4954402.

[3] E. E. Mamajek, A. Prsa, G. Torres, P. Harmanec, M. Asplund, P. D. Bennett, N. Capitaine, J. Christensen-Dalsgaard, E. Depagne, W. M. Folkner, M. Haberreiter, S. Hekker, J. L. Hilton, V. Kostov, D. W. Kurtz, J. Laskar, B. D. Mason, E. F. Milone, M. M. Montgomery, M. T. Richards, J. Schou, and S. G. Stewart. IAU 2015 Resolution B3 on Recommended Nominal Conversion Constants for Selected Solar and Planetary Properties. Technical report, International Astronomical Union, 2015.

[4] B. A. Archinal, C. H. Acton, M. F. A'Hearn, A. Conrad, G. J. Consolmagno, T. Duxbury, D. Hestroffer, J. L. Hilton, R. L. Kirk, S. A. Klioner, D. McCarthy, K. Meech, J. Oberst, J. Ping, P. K. Seidelmann, D. J. Tholen, P. C. Thomas, and I. P. Williams. Report of the iau working group on cartographic coordinates and rotational elements: 2015. *Celestial Mechanics and Dynamical Astronomy*, 130(3):22, Feb 2018. ISSN 1572-9478. doi: 10.1007/s10569-017-9805-5. URL https://doi.org/10.1007/s10569-017-9805-5.

[5] Ryan S. Park. Planets and Pluto: Physical Characteristics, 2018. URL https://ssd.jpl.nasa.gov/?planet_phys_par.

[6] Ralph B. Roncoli. Lunar Constants and Models Document. Technical Report JPL Technical Document D-32296, Jet Propulsion Laboratory/California Institute of Technology, Sep 2005.

[7] E. M. Standish. Keplerian Elements for Approximate Positions of Major Planets. Technical report, Jet Propulsion Laboratory/California Institute of Technology, May 2011.

[8] Lasunncty. Diagram illustrating and explaining various terms in relation to Orbits of Celestial bodies., Oct 2007. URL https://commons.wikimedia.org/wiki/File:Orbit1.svg.

[9] R. Schwarz. Memorandum No. 1 Keplerian Orbit Elements → Cartesian State Vectors. Technical report, Oct 2017.

[10] 614th Air & Space Operations Center. SPACETRACK API, Jun 2018. URL https://www.space-track.org/documentation.

[11] T.S. Kelso. Frequently Asked Questions: Two-Line Element Set Format. *Satellite Times*, Jan 1998. URL https://www.celestrak.com/columns/v04n03/.

[12] B. Rhodes. python-sgp4, Jun 2018. URL https://github.com/brandon-rhodes/python-sgp4.