keplertools

Release 1.0.0

Dmitry Savransky

Dec 19, 2022

CONTENTS:

`keplertools` provides a variety of methods for the propagation of two-body orbits.

CHAPTER

ONE

INSTALLATION

To install from PyPI:

```
pip install keplertools
```

To also compile the Cython versions (compiler required, for details see: https://cython.readthedocs.io/en/latest/src/quickstart/install.html):

```
pip install --no-binary keplertools keplertools[C]
```

If using a zsh shell (or depending on your specific shell setup), you may need to escape the square brackets (i.e., the last bit of the previous command would be `keplertools\[C\]`.

CHAPTER

TWO

CLOSED-ORBIT METHODS

The following methods are for use with closed orbits:

- *eccanom()*
- *trueanom()*
- *vec2orbElem()*
- *vec2orbElem2()*
- *orbElem2vec()*

CHAPTER

THREE

ALL-ORBIT METHODS

The following methods support all orbits (open and closed):

- *invKepler()*
- *kepler2orbstate()*
- *orbstate2kepler()*
- *universalfg()*
- *planSys*

3.1 keplertools

3.1.1 keplertools package

Submodules

keplertools.CyKeplerSTM module

keplertools.Cyeccanom module

keplertools.fun module

keplertools.fun.**c2c3**(*psi: Union[Sequence[Sequence[Sequence[Sequence[Sequence[Any]]]]], _SupportsArray[dtype], Sequence[_SupportsArray[dtype]], Sequence[Sequence[_SupportsArray[dtype]]], Sequence[Sequence[Sequence[_SupportsArray[dtype]]]], Sequence[Sequence[Sequence[Sequence[_SupportsArray[dtype]]]]], bool, int, float, complex, str, bytes, Sequence[Union[bool, int, float, complex, str, bytes]], Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]], Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]], Sequence[Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]]]]*) → Tuple[ndarray[Any, dtype[float64]], ndarray[Any, dtype[float64]]]

Calculate the c2, c3 coefficients for the universal variable

Parameters

psi (*iterable or float*) – psi = chi^2/a for universal variable chi and semi-major axis a

Returns

c2 (numpy.ndarray):
c2 coefficients (same size as input)

c3 (numpy.ndarray):
c3 coefficients (same size as input)

Return type
tuple

keplertools.fun.**calcAB**(*a: ndarray[Any, dtype[float64]]*, *e: ndarray[Any, dtype[float64]]*, *O: ndarray[Any, dtype[float64]]*, *I: ndarray[Any, dtype[float64]]*, *w: ndarray[Any, dtype[float64]]*) → Tuple[ndarray[Any, dtype[float64]], ndarray[Any, dtype[float64]]]

Calculate inertial frame components of perifocal frame unit vectors scaled by orbit semi-major and semi-minor axes.

Note that these quantities are closely related to the Thiele-Innes constants

Parameters

- **a** (*ndarray*) – Semi-major axes
- **e** (*ndarray*) – eccentricities
- **O** (*ndarray*) – longitudes of ascending nodes (rad)
- **I** (*ndarray*) – inclinations (rad)
- **w** (*ndarray*) – arguments of pericenter (rad)

Returns

A (ndarray):
Components of eccentricity vector scaled by a

B (ndarray):
Components of q vector (orthogonal to e and h) scaled by b (=asqrt{1-e^2})

Return type
tuple

Notes

All inputs must be of same size. Outputs are 3xn for n input points. See Vinti (1998) for details on element/coord sys defintions.

keplertools.fun.**eccanom**(*M: Union[Sequence[Sequence[Sequence[Sequence[Sequence[Any]]]]], _SupportsArray[dtype], Sequence[_SupportsArray[dtype]], Sequence[Sequence[_SupportsArray[dtype]]], Sequence[Sequence[Sequence[_SupportsArray[dtype]]]], Sequence[Sequence[Sequence[Sequence[_SupportsArray[dtype]]]]], bool, int, float, complex, str, bytes, Sequence[Union[bool, int, float, complex, str, bytes]], Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]], Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]], Sequence[Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]]]*, *e: Union[Sequence[Sequence[Sequence[Sequence[Sequence[Any]]]]], _SupportsArray[dtype], Sequence[_SupportsArray[dtype]], Sequence[Sequence[_SupportsArray[dtype]]], Sequence[Sequence[Sequence[_SupportsArray[dtype]]]], Sequence[Sequence[Sequence[Sequence[_SupportsArray[dtype]]]]], bool, int, float, complex, str, bytes, Sequence[Union[bool, int, float, complex, str, bytes]], Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]], Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]], Sequence[Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]]]*, *epsmult: float = 4.01*, *maxIter: int = 100*, *returnIter: bool = False*, *noc: bool = False*, *verb: bool = False*) → Union[Tuple[ndarray[Any, dtype[float64]], int], ndarray[Any, dtype[float64]]]

Finds eccentric anomaly from mean anomaly and eccentricity

This method uses Newton-Raphson iteration to find the eccentric anomaly from mean anomaly and eccentricity, assuming a closed (0<e<1) orbit.

Parameters

- **M** (*float or ndarray*) – mean anomaly (rad)
- **e** (*float or ndarray*) – eccentricity (eccentricity may be a scalar if M is given as an array, but otherwise must match the size of M.)
- **epsmult** (*float*) – Precision of convergence (multiplied by precision of floating data type). Optional, defaults to 4.01.
- **maxiter** (*int*) – Maximum numbr of iterations. Optional, defaults to 100.
- **returnIter** (*bool*) – Return number of iterations (defaults false, only available in python version, ignored if using C version)
- **noc** (*bool*) – Don't use C version even if it can be loaded.
- **verb** (*bool*) – Print exactly which version (C or Python is being used)

Returns

E (float or ndarray):
eccentric anomaly (rad)

numIter (int):
Number of iterations (returned only if returnIter=True)

Return type
tuple

Notes

If either M or e are scalar, and the other input is an array, the scalar input will be expanded to the same size array as the other input. So, a scalar M and array e will result in the calculation of the eccentric anomaly for one mean anomaly at a variety of eccentricities, and a scalar e and array M input will result in the calculation of eccentric anomalies for one eccentricity at a variety of mean anomalies. If both inputs are arrays then they are matched element by element.

keplertools.fun.**forcendarray**(*x: Union[float, ndarray[Any, dtype[float64]]]*) → ndarray[Any, dtype[float64]]

Convert any numerical value into 1-D ndarray

Parameters
x (*float or numpy.ndarray*) – Input

Returns
Same size as input but in ndarray form

Return type
numpy.ndarray

keplertools.fun.**invKepler**(*M: Union[float, ndarray[Any, dtype[float64]]]*, *e: Union[float, ndarray[Any, dtype[float64]]]*, *tol: Optional[float] = None*, *E0: Optional[Union[float, ndarray[Any, dtype[float64]]]] = None*, *maxIter: int = 100*, *return_nu: bool = False*, *convergence_error: bool = True*) → Tuple[ndarray[Any, dtype[float64]], ...]

Finds eccentric/hyperbolic/parabolic anomaly from mean anomaly and eccentricity

This method uses Newton-Raphson iteration to find the eccentric anomaly from mean anomaly and eccentricity, assuming a closed (0<e<1) orbit.

Parameters

- **M** (*float or ndarray*) – mean anomaly (rad)
- **e** (*float or ndarray*) – eccentricity (eccentricity may be a scalar if M is given as an array, but otherwise must match the size of M.)
- **tolerance** (*float*) – Convergence of tolerance. Defaults to eps(2*pi)
- **E0** (*float or ndarray*) – Initial guess for iteration. Defaults to Taylor-expansion based value for closed orbits and Vallado-derived heuristic for open orbits. If set, must match size of M.
- **maxiter** (*int*) – Maximum numbr of iterations. Optional, defaults to 100.
- **return_nu** (*bool*) – Return true anomaly (defaults false)
- **convergence_error** (*bool*) – Raise error on convergence failure. Defaults True. If false, throws a warning.

Returns

E (ndarray):
eccentric/parabolic/hyperbolic anomaly (rad)

numIter (ndarray):
Number of iterations

nu (ndarray):
True anomaly (returned only if return_nu=True)

Return type
tuple

Notes

If either M or e are scalar, and the other input is an array, the scalar input will be expanded to the same size array as the other input. So, a scalar M and array e will result in the calculation of the eccentric anomaly for one mean anomaly at a variety of eccentricities, and a scalar e and array M input will result in the calculation of eccentric anomalies for one eccentricity at a variety of mean anomalies. If both inputs are arrays then they are matched element by element.

keplertools.fun.**kepler2orbstate**(*a: Union[float, ndarray[Any, dtype[float64]]]*, *e: Union[float, ndarray[Any, dtype[float64]]]*, *O: Union[float, ndarray[Any, dtype[float64]]]*, *I: Union[float, ndarray[Any, dtype[float64]]]*, *w: Union[float, ndarray[Any, dtype[float64]]]*, *mu: Union[float, ndarray[Any, dtype[float64]]]*, *nu: Union[float, ndarray[Any, dtype[float64]]]*) → Tuple[ndarray[Any, dtype[float64]], ndarray[Any, dtype[float64]]]

Calculate orbital state vectors from Keplerian elements

Parameters

- **a** (*float or numpy.ndarray*) – Semi-major axis (or semi-parameter is e = 1)
- **e** (*float or numpy.ndarray*) – eccentricity
- **O** (*float or numpy.ndarray*) – longitude of ascending node (rad)
- **I** (*float or numpy.ndarray*) – inclination (rad)
- **w** (*float or numpy.ndarray*) – arguments of periapsis (rad)
- **mu** (*float or numpy.ndarray*) – Gravitational parameters. If float, assuming all state vectors belong to the same system.
- **nu** (*float or numpy.ndarray*) – True anomaly (rad)

Returns

r (numpy.ndarray):
Components of orbital radius (n x 3)

v (numpy.ndarray):
Components of orbital velocity (n x 3)

Return type
tuple

Notes

r.flatten() and v.flatten() will automatically stack elements in the proper order in a 1D array

keplertools.fun.**orbElem2vec**(*E: ndarray[Any, dtype[float64]]*, *mus: Union[float, ndarray[Any, dtype[float64]]]*, *orbElem: Optional[Tuple[ndarray[Any, dtype[float64]], ndarray[Any, dtype[float64]], ndarray[Any, dtype[float64]], ndarray[Any, dtype[float64]], ndarray[Any, dtype[float64]]]] = None*, *AB: Optional[Tuple[ndarray[Any, dtype[float64]], ndarray[Any, dtype[float64]]]] = None*, *returnAB: bool = False*) → Union[Tuple[ndarray[Any, dtype[float64]], ndarray[Any, dtype[float64]], Tuple[ndarray[Any, dtype[float64]], ndarray[Any, dtype[float64]]]], Tuple[ndarray[Any, dtype[float64]], ndarray[Any, dtype[float64]]]]

Convert Keplerian orbital elements to position and velocity vectors

Parameters

- **E** (*ndarray*) – nx1 array of eccentric anomalies (rad)
- **mus** (*ndarray or float*) – nx1 array of gravitational parameters (G*m_i) where G is the gravitational constant and m_i is the mass of the ith body. if all vectors represent the same body, mus may be a scalar.
- **orbElem** (*tuple*) – (a,e,O,I,w) Exact inputs to calcAB. Either this or AB input must be set
- **AB** (*tuple*) – (A,B) Exact outpus from calcAB
- **returnAB** (*bool*) – Default False. If True, returns (A,B) as thrid output.

Returns

rs (ndarray):
3 x n stacked position vectors

vs (ndarray):
3 x n stacked velocity vectors

AB (tuple):
(A,B)

Return type
tuple

Notes

All units are complementary, i.e., if mus are in AU^3/day^2 then positions will be in AU, and velocities will be AU/day.

Possible combinations or inputs are:

1. E scalar, mu scalar - single body, single position. A, B should be 3x1 (or orbElem should be all scalars).
2. E vector, mu scalar - single body, many orbital positions. A, B should be 3x1 (or orbElem should be all scalars).
3. E vector, mu vector - multiple bodies at varying orbital positions. A, B should be 3xn where E.size==n (or all orbElem should be size n) and mus.size must equal E.size.

keplertools.fun.**orbstate2kepler**(*r: ndarray[Any, dtype[float64]], v: ndarray[Any, dtype[float64]], mu: Union[float, ndarray[Any, dtype[float64]]]*) → Tuple[ndarray[Any, dtype[float64]], ndarray[Any, dtype[float64]], ndarray[Any, dtype[float64]], ndarray[Any, dtype[float64]], ndarray[Any, dtype[float64]], ndarray[Any, dtype[float64]]]

Calculate Keplerian elements given orbital state vectors

Parameters

- **r** (*numpy.ndarray*) – Components of orbital radius. 3n elements in 1D as [r1(1);r1(2);r1(3);r2(1);r2(2)r2(3);...;rn(1);rn(2);rn(3)] or in 2D as nx3 or 3xn
- **v** (*numpy.ndarray*) – Components of orbital velocity. Same stacking as r
- **mu** (*float or numpy.ndarray*) – Gravitational parameters. If float, assuming all state vectors belong to the same system.

Returns

a (ndarray):
Semi-major axis (or semi-parameter where e = 1)

e (ndarray):
eccentricity

O (ndarray):
longitude of ascending node (rad)

I (ndarray):
inclination (rad)

w (ndarray):
arguments of periapsis (rad)

tp (ndarray):
time of periapsis passage

Return type
tuple

keplertools.fun.**trueanom**(*E: Union[Sequence[Sequence[Sequence[Sequence[Sequence[Any]]]]], _SupportsArray[dtype], Sequence[_SupportsArray[dtype]], Sequence[Sequence[_SupportsArray[dtype]]], Sequence[Sequence[Sequence[_SupportsArray[dtype]]]], Sequence[Sequence[Sequence[Sequence[_SupportsArray[dtype]]]]], bool, int, float, complex, str, bytes, Sequence[Union[bool, int, float, complex, str, bytes]], Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]], Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]], Sequence[Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]]]*, *e: Union[Sequence[Sequence[Sequence[Sequence[Sequence[Any]]]]], _SupportsArray[dtype], Sequence[_SupportsArray[dtype]], Sequence[Sequence[_SupportsArray[dtype]]], Sequence[Sequence[Sequence[_SupportsArray[dtype]]]], Sequence[Sequence[Sequence[Sequence[_SupportsArray[dtype]]]]], bool, int, float, complex, str, bytes, Sequence[Union[bool, int, float, complex, str, bytes]], Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]], Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]], Sequence[Sequence[Sequence[Sequence[Union[bool, int, float, complex, str, bytes]]]]]*) → ndarray[Any, dtype[float64]]

Finds true anomaly from eccentric anomaly and eccentricity

The implemented method corresponds to Eq. 6.28 in Green assuming a closed (0<e<1) orbit.

Parameters

- **E** (*float or ndarray*) – eccentric anomaly (rad)
- **e** (*float or ndarray*) – eccentricity (eccentricity may be a scalar if M is given as an array, but otherwise must match the size of M.)

Returns
true anomaly (rad)

Return type
ndarray

Notes

If either E or e are scalar, and the other input is an array, the scalar input will be expanded to the same size array as the other input.

keplertools.fun.**unitvector**(*vec: ndarray[Any, dtype[float64]]*, *mag: ndarray[Any, dtype[float64]]*) → ndarray[Any, dtype[float64]]

Return the unit vectors of an array of vectors

Parameters

- **vec** (*numpy.ndarray*) – Vectors as nx3
- **mag** (*numpy.ndarray*) – Vector magnitudes as nx1

Returns
Unit vectors in the same layout as input

Return type
numpy.ndarray

keplertools.fun.**universalfg**(*r0: ndarray[Any, dtype[float64]]*, *v0: ndarray[Any, dtype[float64]]*, *mu: Union[float, ndarray[Any, dtype[float64]]]*, *dt: Union[float, ndarray[Any, dtype[float64]]]*, *maxIter: int = 100*, *return_counter: bool = False*, *convergence_error: bool = True*) → Tuple[ndarray[Any, dtype[float64]], ...]

Propagate orbital state vectors by delta t via universal variable-based f and g

Parameters

- **r0** (*numpy.ndarray*) – Components of orbital radius. 3n elements in 1D as [r1(1);r1(2);r1(3);r2(1);r2(2)r2(3);...;rn(1);rn(2);rn(3)] or in 2D as nx3 or 3xn
- **v0** (*numpy.ndarray*) – Components of orbital velocity. Same stacking as r
- **mu** (*float or numpy.ndarray*) – Gravitational parameters. If float, assuming all state vectors belong to the same system.
- **dt** (*float or numpy.ndarray*) – Propagation time. If float, assuming all states are propagated for the same time
- **return_counter** (*bool*) – If True, returns the number of iterations for each input state. Defaults False.
- **convergence_error** (*bool*) – Raise error on convergence failure if True. Defaults True.

Returns

r (numpy.ndarray):
Components of orbital radius (n x 3)

v (numpy.ndarray):
Components of orbital velocity (n x 3)

counter(numpy.ndarray):
Number of required iterations (size n). Only returned if return_counter is True

Return type
tuple

Notes

r.flatten() and v.flatten() will automatically stack elements in the proper order in a 1D array

keplertools.fun.**validateOrbitalStateInputs**(*r: ndarray[Any, dtype[float64]]*, *v: ndarray[Any, dtype[float64]]*, *mu: Union[float, ndarray[Any, dtype[float64]]]*) → Tuple[ndarray[Any, dtype[float64]], ndarray[Any, dtype[float64]], ndarray[Any, dtype[float64]]]

Validate and standardize dimensionality of orbital state vector inputs

Parameters

- **r** (*numpy.ndarray*) – Components of orbital radius. 3n elements in 1D as [r1(1);r1(2);r1(3);r2(1);r2(2)r2(3);...;rn(1);rn(2);rn(3)] or in 2D as nx3 or 3xn
- **v** (*numpy.ndarray*) – Components of orbital velocity. Same stacking as r
- **mu** (*float or numpy.ndarray*) – Gravitational parameters. If float, assuming all state vectors belong to the same system.

Returns

r (numpy.ndarray):
Components of orbital radius. (n x 3)

v (numpy.ndarray):
Components of orbital velocity. (n x 3)

mu (numpy.ndarray):
Gravitational parameters. (size 1 or n)

Return type
tuple

keplertools.fun.**vec2orbElem**(*rs: ndarray[Any, dtype[float64]]*, *vs: ndarray[Any, dtype[float64]]*, *mus: Union[float, ndarray[Any, dtype[float64]]]*) → Tuple[ndarray[Any, dtype[float64]], ndarray[Any, dtype[float64]], ndarray[Any, dtype[float64]], ndarray[Any, dtype[float64]], ndarray[Any, dtype[float64]], ndarray[Any, dtype[float64]], ndarray[Any, dtype[float64]]]

Convert position and velocity vectors to Keplerian orbital elements

Implements the (corrected) algorithm from Vinti

Parameters

- **rs** (*ndarray*) – 3n x 1 stacked initial position vectors: [r1(1);r1(2);r1(3);r2(1);r2(2)r2(3);...;rn(1);rn(2);rn(3)] or 3 x n or n x 3 matrix of position vectors.
- **vs** (*ndarray*) – 3n x 1 stacked initial velocity vectors or 3 x n or n x3 matrix
- **mus** (*ndarray or float*) – nx1 array of gravitational parameters (G*m_i) where G is the gravitational constant and m_i is the mass of the ith body. if all vectors represent the same body, mus may be a scalar.

Returns

a (ndarray):
Semi-major axes

e (ndarray):
eccentricities

E (ndarray):
eccentric anomalies

O (ndarray):
longitudes of ascending nodes (rad)

I (ndarray):
inclinations (rad)

w (ndarray):
arguments of pericenter (rad)

P (ndarray):
orbital periods

tau (ndarray):
time of periapsis crossing

Return type
tuple

Notes

All units must be complementary, i.e., if positions are in AU, and time is in days, vs must be in AU/day, mus must be in AU^3/day^2

keplertools.fun.**vec2orbElem2**(*rs: ndarray[Any, dtype[float64]]*, *vs: ndarray[Any, dtype[float64]]*, *mus: Union[float, ndarray[Any, dtype[float64]]]*) → Tuple[ndarray[Any, dtype[float64]], ndarray[Any, dtype[float64]], ndarray[Any, dtype[float64]], ndarray[Any, dtype[float64]], ndarray[Any, dtype[float64]], ndarray[Any, dtype[float64]], ndarray[Any, dtype[float64]]]

Convert position and velocity vectors to Keplerian orbital elements

Implements the algorithm from Vallado

Parameters

- **rs** (*ndarray*) – 3n x 1 stacked initial position vectors: [r1(1);r1(2);r1(3);r2(1);r2(2)r2(3);...;rn(1);rn(2);rn(3)] or 3 x n or n x 3 matrix of position vectprs.
- **vs** (*ndarray*) – 3n x 1 stacked initial velocity vectors or 3 x n or n x3 matrix
- **mus** (*ndarray or float*) – nx1 array of gravitational parameters (G*m_i) where G is the gravitational constant and m_i is the mass of the ith body. if all vectors represent the same body, mus may be a scalar.

Returns

a (ndarray):
Semi-major axes

e (ndarray):
eccentricities

E (ndarray):
eccentric anomalies

O (ndarray):
longitudes of ascending nodes (rad)

I (ndarray):
inclinations (rad)

w (ndarray):
arguments of pericenter (rad)

P (ndarray):
orbital periods

tau (ndarray):
time of periapsis crossing

Return type
tuple

Notes

All units must be complementary, i.e., if positions are in AU, and time is in days, vs must be in AU/day, mus must be in AU^3/day^2

keplertools.keplerSTM module

class keplertools.keplerSTM.**planSys**(*x0*, *mu*, *epsmult=4.0*, *prefVallado=False*, *noc=False*)

Bases: object

calcSTM(*dt*)

calcSTM_vallado(*dt*)

contFrac(*x*, *a=5.0*, *b=0.0*, *c=2.5*)

psi2c2c3(*psi0*)

takeStep(*dt*)

updateState(*x0*)

Module contents

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

k

INDEX