SORTS++ Documentation

Release 3.0.0

Daniel Kastinen, Juha Vierinen

Apr 25, 2019

CONTENTS:

CHAPTER ONE

INTRODUCTION

1.1 What is SORTS++

SORTS++ stands for Next-generation (++) Space Object Radar Tracking Simulator (SORTS). It is a collection of modules designed for research purposes concerning the tracking and detection of objects in space. Its ultimate goal is to simulate the tracking and discovery of objects in space using radar systems in a very general fashion.

1.2 Install

1.2.1 System requirements

- Unix (tested on Ubuntu-16.04 LTS, Ubuntu-server-16.04 LTS)
- Python 2.7

1.2.2 Dependencies

```
h5py>=2.7.1
matplotlib>=2.2.2
mpi4py>=3.0.0
numpy>=1.14.3
pyproj>=1.9.5.1
python-dateutil==2.7.3
scipy==1.1.0
sgp4==1.4
pytest>=4.1.1
```

1.3 "I'm feeling lucky" Install

THIS SHOULD BE A MAKEFILE All of the series of install instructions can also be performed by running the `build.sh` file from the `SORTSpp/` folder after cloning the repository:

make <dependancy>

```
make all
make install
```

1.4 Test installation

1.4.1 Modules

Simply navigate to the `SORTSpp` directory and run:

```
pytest
```

And it will automatically use the `pytest.ini` file to discover and run all tests.

1.4.2 Simulation

To test the simulation capabilities (usage of all modules simultaniusly):

- Look in the **SIMULATIONS/** folder
- Configure the files ending with `*_test.py` to output data to desired paths.

Remember, To run the simulation with MPI the file must be executable. To test a capability run the corresponding `test_*` file with

```
python ./SIMULATIONS/test_simulation.py
```

or with

```
mpirun -np 8 ./SIMULATIONS/test_simulation.py
```

if you wish to test the MPI implementation of the simulation. The *-np* specifies how many processes should be launched and should not be larger then the number of cores available.

1.5 General simulation

To perform a general simulation using SORTS++ you need to use the `simulation` class.

To construct a simulation class instance you need:

Radar instance Manually constructed from `radar_system` or a preset instance from *`radar_library`*.

Population instance Manually constructed from `population` or a preset instance from *`population`*.

Simulation root A designated root folder where simulation files will be stored.

These are the bare minimum, it is also recommended to have:

Radar scan Manually constructed from `radar_scan` or a preset instance from *`radar_scan_library`*. The radar_scan instance should be set in the radar_system instance using the `set_scan()` method.

Scheduler instance A scheduler instance is a function declaration located in *`scheduler_library`*. Different schedulers need different configurations and auxiliary functions and instances.

Below is an example simulation of catalogue maintenance:

The simulation can be found in `SIMULATION_EXAMPLES/simple_sim.py`, remember to change the simulation root before running, it can be run using

```
mpirun -np 4 ./SIMULATION_EXAMPLES/simple_sim.py
```

1.6 License

MIT License

Copyright (c) [2019] [Daniel Kastinen, Juha Vierinen]

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

CHAPTER

TWO

COORDINATE CONVENTIONS

2.1 Orbit conventions

Orientation of the ellipse in the coordinate system:

- For zero inclination i: the ellipse is located in the x-y plane.
- The direction of motion as True anoamly
  nu: increases for a zero inclination i: orbit is anti-coockwise, i.e. from +x towards +y.
- If the eccentricity e: is increased, the periapsis will lie in +x direction.
- If the inclination i: is increased, the ellipse will rotate around the x-axis, so that +y is rotated toward +z.
- An increase in Longitude of ascending node Ω: corresponds to a rotation around the z-axis so that +x is rotated toward +y.
- Changing argument of perihelion ω: will not change the plane of the orbit, it will rotate the orbit in the plane.
- The periapsis is shifted in the direction of motion.
- True anomaly measures from the +x axis, i.e
  $nu = 0$ is located at periapsis and
  $nu = \pi$ at apoapsis.
- All anomalies and orientation angles reach between 0 and 2π

Reference: "Orbital Motion" by A.E. Roy.

2.2 Coordinate transformation guide

In general there are 2 classes of coordinate systems:

- Earth Centered Inertial (ECI)
- Earth Centered Earth Fixed (ECEF)

There are several realizations of these classes of coordinate systems that take into account different effects and perturbations. The difference between an Inertial and an Earth Fixed frame is that in an inertial system all motion comes from classical orbit dynamics (N-body solutions) and are not caused by the coordinate frame transformation.

Reference	Type	Coordinate frame name
ITRF	ECEF	International Terrestrial Reference Frame
PEF	ECEF	Pseudo-Earth Fixed reference frame
CIRF	ECI	Celestial Intermediate Reference Frame
MOD	ECI	Mean-Of-Date reference frame
TOD	ECI	True-Of-Data reference frame
GCRF	ECI	Geocentric Celestial Reference Frame (GCRF)
J2000	ECI	J2000 reference frame (Also called EME2000)
TEME	ECI	True Equator, Mean Equinox reference frame

https://www.orekit.org/site-orekit-9.3.1/architecture/frames.html

As an example, consider a Keplerian orbit (i.e. a point moving on a ellipse) around the Earth. An inertial frame here is any barycentric Cartesian fixed frame (barycentric can be approximated as the Earth Centric due to the small mass of the orbiting object). An example of a non-inertial frame could be a translating Cartesian frame, here the object would seem to be "spiraling" away from us. In this frame the movement away from us is not induced by fundamental orbital dynamics but due to the coordinate frame transformation. The same is true in a Earth Fixed system, the orbit would seem to rotate at the speed of the Earths rotation.

Since any barycentric Cartesian fixed frame is Inertial it is customary to choose 2 reference directions to make the frame choice unique. These reference directions are usually the rotational axis of the Earth and the Vernal Equinox, i.e. the direction in space formed by the intersection of the Earths orbital plane around the Sun and the Earth equatorial plane. The direction chosen for the +x axis is usually defined so that it is aligned with the direction when axial tilt of the Earth in +z direction (the Earth moving counter-clockwise) is moving from towards the Sun to away from the Sun. The Vernal Equinox with this definition is also the ascending node of the ecliptic on the celestial equator.

Since the orbital dynamics of the Earth in the solar-system has no analytic solutions due to perturbations, the definition of Vernal Equinox and the Earth ecliptic also changes with respect to time, thus is it customary to choose the common reference direction for the Vernal Equinox at a specific time, called the Epoch of that equinox.

From numerical simulations the drift of the Obliquity of the ecliptic (inclination of ecliptic with respect to the celestial equator) does not vary more then 1 degree on the order of 10,000 years.

Most commonly used ECI's are:

True Equator Mean Equinox (TEME)

This is the frame after a Two-Line Element (TLE) orbit has been converted to an Cartesian state. The Mean Equinox refers to the Vernal Equinox but averaged over time to remove nutation. The Mean Vernal Equinox here is aligned to coincide with the +x axis. Thus the instantaneous Vernal Equinox is different at any point in time and needs to be modeled. True Equator refers to the fact that the instantaneous axis of rotation of the Earth is used to align the +z axis with.

The International Terrestrial Reference Frame (ITRF)

The ITRF contains models of movement of both the Earth and the Equinox. Thus the frame itself is a function of time. As the models are updated it is customary to denote the reference frame by a Epoch, or the time around witch they "center".

Going from a "Mean" element definition to a Instantaneous one requires a model of nutation.

To transform from e.g. TEME to ITRF one would first need to find the difference between the instantanius mean equionox

Then find the instantanius earth rotation. . . .

then find the rotation of the earth, also known as GMST

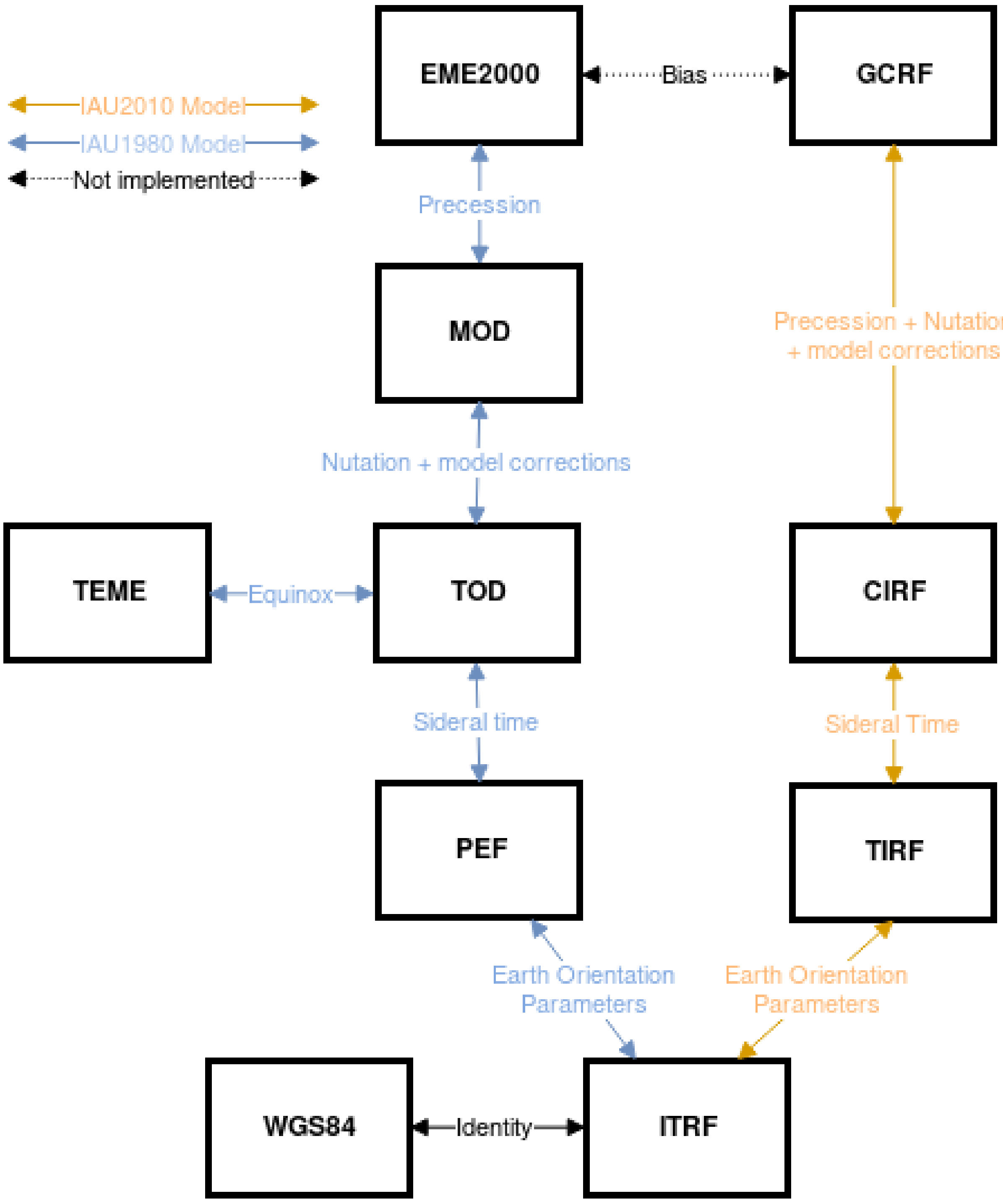

Fig. 1: Flowchart describing the relation between different frames. Original image Copyright (c) 2018 Jules David under the MIT license. Source: beyond.

CHAPTER THREE

OPTIONAL DEPENDENCIES

```
basemap>=1.1.0
ffmpeg>=1.4
sphinx>=1.8.1
```

3.1 Basemap

To install **basemap**:

Install proj-bin:

```
sudo apt-get install proj-bin
```

Get the basemap source

```
wget --no-check-certificate https://github.com/matplotlib/basemap/archive/master.tar.
↪gz
```

Un-tar the basemap version X.Y.Z source tar.gz file, and enter the basemap-X.Y.Z directory

```
export GEOS_DIR=<where you want the libs and headers to go>
```

Then go to the geos distribution in the un-tar'ed basemap and run

```
./configure --prefix=$GEOS_DIR
make; make install
```

Lastly be sure to use the python bin in the virtualenv to install basemap as:: bash

/. . . ./SORTSpp/env2.7/bin/python setup.py install

3.2 Pyglow

Follow the installation guide on Pyglow.

CHAPTER

FOUR

INSTALLING PROPAGATORS

4.1 Orekit

Firstly check openJDK version:

```
java -version
```

if OpenJDK not installed:

```
sudo apt-get install openjdk-7-jdk
```

or

```
sudo apt-get install openjdk-8-jdk
```

Then make sure jcc is installed:

```
sudo apt-get install jcc
```

Then create a Python-2.7 environment in an appropriate folder:

```
virtualenv env
```

Activate the environment:

```
source env/bin/activate
```

Depending on your installation, make sure that the `JCC_JDK` variable is set:

```
export JCC_JDK="/usr/lib/jvm/java-8-openjdk-amd64"
```

Again, this DOES NOT work with java-9, needs 8 or 7.

Then install JCC into the environment:

```
pip install jcc
```

go to: Hipparchus and download binary for version 1.3. Extract the .jar files with some archive manager, e.g. *tar*.

Clone the modified orekit including python package java classes: Orekit with python .

Follow the instructions in: Build orekit

Tested building on Ubuntu 16.04:

```
 sudo apt install maven
mvn package
```

If you have problem with some tests failing when building orekit, make sure you check the *petrushy/Orekit.git* repository status and ensure that you have the correct branch checked out before compiling (as of writing, tested branch on Ubuntu 16.04 is *develop*).

After compilation is complete, go to "/Orekit/target/" and to find the **orekit-x.jar**

Clone the python wrapper repository: Orekit python wrapper

Copy the contents of the "python_files" folder (from the python wrapper repository) to the folder where you intend to build the python library.

Then place all the **hipparchus-Y.jar** files and your modified compiled **orekit-x.jar** file in your build folder.

More specifically these files are needed:

- orekit-x.jar
- hipparchus-core-1.3.jar
- hipparchus-filtering-1.3.jar
- hipparchus-fitting-1.3.jar
- hipparchus-geometry-1.3.jar
- hipparchus-ode-1.3.jar
- hipparchus-optim-1.3.jar
- hipparchus-stat-1.3.jar

A summation of these commands are

```
wget https://www.hipparchus.org/downloads/hipparchus-1.3-bin.zip
unzip hipparchus-1.3-bin.zip

git clone https://github.com/petrushy/Orekit.git

cd Orekit
git checkout develop
export _JAVA_OPTIONS="-Dorekit.data.path=/the/path/to/Orekit/"
mvn package

cd ..
mkdir build

git clone https://gitlab.orekit.org/orekit-labs/python-wrapper.git

cp -v Orekit/target/orekit*.jar build/
cp -v hipparchus-1.3-bin/*.jar build/
cp -rv python-wrapper/python_files/* build/
```

Set the environment variable for building:

```
export SRC_DIR="my/orekit/build/folder"
export _JAVA_OPTIONS="-Dorekit.data.path=/full/path/to/Orekit/"
```

In this folder create a build.sh file with the following contents (remember to replace the **x**'es with the correct version compiled):

```bash
#!/bin/bash

python -m jcc \
--use_full_names \
--python orekit \
--version x \
--jar $SRC_DIR/orekit-x.jar \
--jar $SRC_DIR/hipparchus-core-1.3.jar \
--jar $SRC_DIR/hipparchus-filtering-1.3.jar \
--jar $SRC_DIR/hipparchus-fitting-1.3.jar \
--jar $SRC_DIR/hipparchus-geometry-1.3.jar \
--jar $SRC_DIR/hipparchus-ode-1.3.jar \
--jar $SRC_DIR/hipparchus-optim-1.3.jar \
--jar $SRC_DIR/hipparchus-stat-1.3.jar \
--package java.io \
--package java.util \
--package java.text \
--package org.orekit \
java.io.BufferedReader \
java.io.FileInputStream \
java.io.FileOutputStream \
java.io.InputStream \
java.io.InputStreamReader \
java.io.ObjectInputStream \
java.io.ObjectOutputStream \
java.io.PrintStream \
java.io.StringReader \
java.io.StringWriter \
java.lang.System \
java.text.DecimalFormat \
java.text.DecimalFormatSymbols \
java.util.ArrayList \
java.util.Arrays \
java.util.Collection \
java.util.Collections \
java.util.Date \
java.util.HashMap \
java.util.HashSet \
java.util.List \
java.util.Locale \
java.util.Map \
java.util.Set \
java.util.TreeSet \
--module $SRC_DIR/pyhelpers.py \
--reserved INFINITE \
--reserved ERROR \
--reserved OVERFLOW \
--reserved NO_DATA \
--reserved NAN \
--reserved min \
--reserved max \
--reserved mean \
--reserved SNAN \
--build \
--install
```

This command is taken from the *conda-recipe* build sh file.

Make the file executable

```
chmod +x build.sh
```

Run the build file

```
./build.sh
```

This may take some time.

Check installation by

```
pip freeze
```

it should output:

```
JCC==3.4
orekit==9.2
```

Then install some additional libraries

```
pip install scipy
pip install matplotlib
pip install pytest
```

Make sure that you test that the installation and compilation worked. Enter into the "test" folder (should have been part of the "python_files" folder) and run:

```
pytest
```

4.2 SGP4

```
pip install sgp4
```

CHAPTER

FIVE

STEP BY STEP GUIDES

5.1 Step by step: fresh Ubuntu 16.04 LTS

If needed:

```
sudo dpkg --configure -a
```

Proceed to:

```
sudo apt-get install git
cd /my/projects_dir/
git clone https://gitlab.irf.se/danielk/SORTSpp.git
cd SORTSpp/
```

Check your currently installed python versions:

```
python --version
```

If this does NOT return *Python 2.7.x*:

```
sudo apt-get install python-dev
```

Check that pip is installed and bound to your python 2.7:

```
pip --version
```

If pip is NOT installed:

```
sudo apt-get install python-pip
```

At this stage: DO NOT UPGRADE PIP. Do this after the virtualenv is installed and activated.

Then install and create a virtualenv, here the name "env2.7" is used since this name is included in the .gitignore file and will not be detected by git:

```
pip install virtualenv
virtualenv --version
virtualenv env2.7
```

Activate virtualenv:

```
source env2.7/bin/activate
```

If needed (should already be latest version), upgrade the pip inside the virtualenv:

```
pip install --upgrade pip
```

Then make sure additional requirements are fulfilled:

- Used by matplotlib

```
sudo apt-get install libfreetype6-dev
sudo apt-get install libpng12-dev
sudo apt-get install python-tk
```

- Used by mpi4py

```
sudo apt-get install libopenmpi-dev
```

Then install the dependency requirement for SORTS++

```
pip install -r pip_req.txt
```

Then test the installation following the test section below.

CHAPTER

SIX

USAGE EXAMPLES

6.1 Module level usage

6.1.1 Scanning for a single space object

6.1.2 Tracking a single space object

6.1.3 Filtering out detectable population

6.2 Expanding on libraries

6.2.1 Adding radar systems

All radar systems implemented should be added in the *`radar_library`* module.

To create a radar system, define a function that returns a `radar_config.radar_system` instance.

To create a `radar_config.radar_system` instance three objects are needed:

- A list of instances of `radar_config.tx_antenna`
- A list of instances of `radar_config.rx_antenna`
- A radar name

To create a instance of `radar_config.rx_antenna` the following is needed:

- Name
- latitude
- longitude
- minimum elevation (degrees)
- frequency
- reviver noise temperature
- radiation pattern: instance of `antenna.beam_pattern` class

To create a instance of `radar_config.tx_antenna` the following is needed:

- Name
- latitude
- longitude

- minimum elevation (degrees)
- frequency
- reviver noise temperature
- radiation pattern: instance of `antenna.beam_pattern` class
- scanning pattern: instance of `radar_scans.radar_scan` class
- Transmit power (MW)
- Transmit bandwidth (Hz)
- duty cycle

Then optionally it is also possible to supply:

- Transmit pule length
- Transmit inter pule period
- Pules used for coherent integration

Below is a example of an implemented radar system:

6.3 Full simulation usage

example stuff

6.3.1 Parsing output files

Example on how to parse the tracklet files generated by executing the method `run_scan()`

6.4 Not documented yet

6.5 test sims

CHAPTER

SEVEN

DOCUMENTATION

7.1 Simulation handler

`simulation`	Main simulation handler in the form of a class using the capabilities of the entire toolbox.

7.1.1 simulation

Main simulation handler in the form of a class using the capabilities of the entire toolbox.

To construct a simulation instance you need:

Radar instance Manually constructed from `radar_system` or a preset instance from `radar_library`.

Population instance Manually constructed from `population` or a preset instance from `population`.

Simulation root A designated root folder where simulation files will be stored.

These are the bare minimum, it is also recommended to have:

Radar scan Manually constructed from `radar_scan` or a preset instance from `radar_scan_library`. The radar_scan instance should be set in the radar_system instance using the `set_scan()` method.

Scheduler instance A scheduler instance is a function declaration located in `scheduler_library`. Different schedulers need different configurations and auxiliary functions and instances.

class simulation.**ObservationParameters**(*duty_cycle*, *SST_f*, *tracking_f*, *coher_int_t=0.1*, *IPP=0.01*, *interleaving_time_slice=0.4*, *scan_during_interleaved=False*)

Container for observation parameters and the function to calculate them in a consistent manner.

# TODO: Write docstring

calculate_parameters(***kwargs*)

configure_radar_to_observation(*radar*, *mode=None*)

Depending on the current observation mode, set coherent integration bandwidth.

Modes:

- 'scan'
- 'track'

load (*fname*)
Load class data from file.

save (*fname*)
Save data to file.

class simulation.**Simulation** (*radar*, *population*, *root*, *scheduler=<function dynamic_scheduler>*, *simulation_name='SORTS++ Simulation'*)
Bases: `object`

Main simulation handler class.

# TODO: Write docstring

Always iters self.__my_objects[thread id] when doing paralell MPI stuff

branch_simulation (*new_version*)
Branch a copy of the current simulation to a new version.

check_load ()

checkout_simulation (*reference_version*)
Checkout a copy of the given simulation version and replace the current version with it.

clear_detections ()
Delete all files in "detections" folder. Branch specific.

clear_logs ()
Delete all files in "logs" folder. Affects entire Simulation.

clear_orbits ()
Delete all files in "orbits" folder. Branch specific.

clear_plots ()
Delete all files in "plots" folder. Branch specific.

clear_prior ()
Delete all files in "prior" folder. Branch specific.

clear_simulation ()
Clear current version folder of all files.

clear_tracklets ()
Delete all files in "tracklets" folder. Branch specific.

discover_orbits ()

generate_priors ()

generate_tracklets ()

list ()
List all available methods.

load ()

maintain_discovered ()

observation_parameters (***kwargs*)
Calculate and set the necessary observation parameters. If just a subset of parameter is supplied the others keep their old values.

The observation default parameters can be found by looking at the source code to this function.

Parameters **kwargs** (*dict*) – Observation parameters to set before re-calculating and saving observation meta-data.

Keyword arguments:

- duty_cycle [float]: Description
- SST_fraction [float]: Description
- tracking_fraction [float]: Description
- interleaving_time_slice [float]: Description
- SST_time_slice [float]: Description
- IPP [float]: Description
- scan_during_interleaved [bool]: Description

`plot_beams`()
Plot all beam-patterns of all transmitters and receivers.

`plot_radar`(*save_folder*)
Plot radar configuration, includes beam pattern, geographical location and scan.

`plots`()

`print_detections`()

`print_maintenance`()

`print_tracklets`()

`print_tracks`()

`run_observation`(***kwargs*)

`run_scheduler`(***kwargs*)

`save`(*MPI_synch=True*)

`schedule_movie`(*time_len=0.0005555555555555556*, *dt=None*)
NEEDS UPDATING

`set_log_level`(***kwargs*)

`set_logfile_level`(***kwargs*)

`set_scheduler_args`(***kwargs*)

`set_terminal_level`(***kwargs*)

`set_version`(*version*)

`simulation_parameters`(***kwargs*)
Calculate and set the necessary simulation parameters.

The simulation default parameters can be found by looking at the source code to this function.

Parameters **`kwargs`** (`dict`) – Simulation parameters to set before re-calculating and saving simulation meta-data. Keyword arguments not in the list of supported parameters will be ignored.

Keyword arguments:

- max_dpos [float]: Description
- tracklet_noise [bool]: Description
- auto_synchronize [bool]: Determines if threads should be automatically synchronized after state changing commands (like run_observations)

status (*fout=None*)
Print summary status of the simulation.

7.2 Class modules

radar_config	This module is used to define the radar network configuration.
antenna	Defines an antenna's or entire radar system's radiation pattern, also defines physical antennas for RX and TX.
propagator_base	A parent class used for interfacing any propagator.
population	Defines a population of space objects in the form of a class.
radar_scans	Defines what a radar observation schema is in the form of a class.
space_object	Defines a space object.
catalogue	Catalogue class.

7.2.1 radar_config

This module is used to define the radar network configuration.

# TODO: Change all attribute names according to convention: 'var' public, '_var' internal, '__var' private. # TODO: It would make sens to change it so that a rx antenna is always a reciver but a TX antenna inherrents RX antenna as it is now but that every TX antenna is also automatically counted as a RX antenna so that you do not e.g. have to specify 'skibotten RX' and 'skibotten TX', instead it would only be 'skibotten' but a instance of TX instead of RX and then you intead only loop over stations in radar system and know that all have RX capabilities but at least one have to have TX capabilities. # TODO: Change name of this module to radar.py

class radar_config.**RadarSystem** (*tx_lst*, *rx_lst*, *name*, *max_on_axis=90.0*, *min_SNRdb=1.0*)
Bases: `object`

A network of transmitting and receiving radar systems.

Variables

- **_tx** (*list*) – List of transmitting sites, i.e. instances of *antenna.AntennaTX*
- **_rx** (*list*) – List of receiving sites, i.e. instances of *antenna.AntennaRX*
- **max_on_axis** (*float*) – Maximum angle between pointing direction and a received signal.
- **name** (*string*) – Verbose name of the radar system
- **_horizon_elevation** (*float*) – Elevation in degrees of the horizon, i.e. minimum elevation the radar system can measure and point.
- **min_SNRdb** (*float*) – Minimum SNR detectable by radar system in dB.

Parameters

- **tx_lst** (*list*) – List of transmitting sites, i.e. instances of *antenna.AntennaTX*
- **rx_lst** (*list*) – List of receiving sites, i.e. instances of *antenna.AntennaRX*
- **name** (*string*) – Verbose name of the radar system

- **max_on_axis** (*float*) – Maximum angle between pointing direction and a received signal.
- **min_SNRdb** (*float*) – Minimum SNR detectable by radar system in dB.

draw3d(*ax*)

set_FOV(*max_on_axis*, *horizon_elevation*)
Set the Field of View (FOV) for this radar system. The FOV is imposed on every receiving station and transmitting station in the network. The FOV is assumed to be azimutally symmetric.

Parameters

- **max_on_axis** (*float*) – Maximum angle in degrees from the pointing direction at witch a detection can be made.
- **horizon_elevation** (*float*) – The elevation angle in degrees of the FOV.

set_SNR_limits(*min_total_SNRdb*, *min_pair_SNRdb*)
Set the Signal to Noise Ratio (SNR) limits for the system.

Parameters

- **min_total_SNRdb** (*float*) – The minimum SNR in dB that is required on at least one transmitter-receiver pair for a detection to be made.
- **min_pair_SNRdb** (*float*) – The minimum SNR in dB that is required for a transmitter-receiver pair to have a detection.

set_TX_bandwith(*bw*)
Set the transmission bandwidth in Hz of all transmitters in the radar system.

Parameters bw (*float*) – Transmission bandwidth in Hz. This is basically what range of frequencies available for wave forming the transmission, e.g. how fast bit-key-shifting code can switch from 0 to π and can then be calculated as the inverse of the baud length.

set_beam(*beam*, *mode='all'*)
Sets the radiation pattern for transmitters, receivers or entire radar system.

To manually set custom beams for each transmitter and receiver in the radar system, set the attributes directly using instances of `antenna.BeamPattern`.

Parameters

- **beam** (`BeamPattern`) – The radiation pattern to set for radar system.
- **mode** (*str*) – String describing what part of radar system to set beam for: Options are `'TX'` for transmission, `'RX'` for reception, or both when left unset.

Example:

```
import antenna_library as alib
from my_radar import radar

#radar is a instance of RadarSystem
radar.set_beam(
    alib.planar_beam(az0=0, el0=90, lat=68, lon=0, I_0=10**4.5, a0=40.0,␣
↪az1=0.0, el1=90.0, f=233e6),
    'TX'
)
```

set_scan(*SST*, *secondary_list=None*)
Set the observation schema that the radar system will use.

Parameters

- **SST** (*radar_scan*) – Sets the main SST observation schema.
- **secondary_list** (*list*) – Sets a list of other observation schema's, i.e. instances of `radar_scans.radar_scan`, that are interleaved with the main SST scan.

`radar_config.`**`plot_radar`**(*radar*, *save_folder=None*)

Plots aspects of the radar system.

Current plots:

- Geographical locations.
- Antenna patterns.
- Scan patterns.

`radar_config.`**`plot_radar_earth`**(*ax*, *radar*)

`radar_config.`**`plot_radar_geo`**(*radar*)

Plot the geographical location of the radar system using the GeoPandas library.

To get:

pip install git+git://github.com/geopandas/geopandas.git pip install descartes

include in basic SORTS++ install?

7.2.2 antenna

Defines an antenna's or entire radar system's radiation pattern, also defines physical antennas for RX and TX.

(c) 2016-2019 Juha Vierinen, Daniel Kastinen

class `antenna.`**`AntennaRX`**(*name*, *lat*, *lon*, *alt*, *el_thresh*, *freq*, *rx_noise*, *beam*)

Bases: `object`

A receiving radar system (antenna or array of antennas).

Parameters

- **name** (*str*) – Name of transmitting radar.
- **lat** (*float*) – Geographical latitude of radar system in decimal degrees (North+).
- **lon** (*float*) – Geographical longitude of radar system in decimal degrees (East+).
- **alt** (*float*) – Geographical altitude above geoid surface of radar system in meter.
- **el_thresh** (*float*) – Elevation threshold for radar station, i.e. it cannot detect or point below this elevation.
- **freq** (*float*) – Operating frequency of radar station in Hz, i.e. carrier wave frequncy.
- **rx_noise** (*float*) – Receiver noise in Kelvin, i.e. system temperature.
- **ant** (`BeamPattern`) – Radiation pattern for radar station.

Variables

- **name** (*str*) – Name of transmitting radar.
- **lat** (*float*) – Geographical latitude of radar system in decimal degrees (North+).
- **lon** (*float*) – Geographical longitude of radar system in decimal degrees (East+).
- **alt** (*float*) – Geographical altitude above geoid surface of radar system in meter.

- **el_thresh** (*float*) – Elevation threshold for radar station, i.e. it cannot detect or point below this elevation.
- **freq** (*float*) – Operating frequency of radar station in Hz, i.e. carrier wave frequncy.
- **wavelength** (*float*) – Operating wavelength of radar station in meter.
- **rx_noise** (*float*) – Reviver noise in Kelvin, i.e. system temperature.
- **beam** (BeamPattern) – Radiation pattern for radar station.
- **ecef** (*numpy.array*) – The ECEF coordinates of the radar system calculated using *coord.geodetic2ecef()*.

point_ecef (*point*)
Point antenna beam in location of ECEF coordinate. Returns local pointing direction.

class antenna.**AntennaTX** (*name*, *lat*, *lon*, *alt*, *el_thresh*, *freq*, *rx_noise*, *beam*, *scan*, *tx_power*, *tx_bandwidth*, *duty_cycle*, *pulse_length=0.001*, *ipp=0.01*, *n_ipp=20*)
Bases: *antenna.AntennaRX*

A transmitting radar system (antenna or array of antennas)

Parameters

- **name** (*str*) – Name of transmitting radar.
- **lat** (*float*) – Geographical latitude of radar system in decimal degrees (North+).
- **lon** (*float*) – Geographical longitude of radar system in decimal degrees (East+).
- **alt** (*float*) – Geographical altitude above geoid surface of radar system in meter.
- **el_tresh** (*float*) – Elevation threshold for radar station, i.e. it cannot detect or point below this elevation.
- **freq** (*float*) – Operating frequency of radar station in Hz, i.e. carrier wave frequency.
- **rx_noise** (*float*) – Receiver noise in Kelvin, i.e. system temperature.
- **beam** (BeamPattern) – Radiation pattern for radar station.
- **tx_bandwidth** (*float*) – Transmissions bandwidth.
- **duty_cycle** (*float*) – Maximum duty cycle, i.e. fraction of time transmission can occur at maximum power.
- **tx_power** (*float*) – Transmissions power in watts.
- **pulse_length** (*float*) – Length of transmission pulse.
- **ipp** (*float*) – Time between consecutive pulses.
- **n_ipp** (*int*) – Number of pulses to coherently integrate.

Variables

- **name** (*str*) – Name of transmitting radar.
- **lat** (*float*) – Geographical latitude of radar system in decimal degrees (North+).
- **lon** (*float*) – Geographical longitude of radar system in decimal degrees (East+).
- **alt** (*float*) – Geographical altitude above geoid surface of radar system in meter.
- **el_thresh** (*float*) – Elevation threshold for radar station, i.e. it cannot detect or point below this elevation.
- **freq** (*float*) – Operating frequency of radar station in Hz, i.e. carrier wave frequency.

- **wavelength** (*float*) – Operating wavelength of radar station in meter.
- **rx_noise** (*float*) – Reviver noise in Kelvin, i.e. system temperature.
- **beam** (BeamPattern) – Radiation pattern for radar station.
- **ecef** (*numpy.array*) – The ECEF coordinates of the radar system calculated using *coord.geodetic2ecef()*.
- **tx_bandwidth** (*float*) – Transmissions bandwidth.
- **duty_cycle** (*float*) – Maximum duty cycle, i.e. fraction of time transmission can occur at maximum power.
- **tx_power** (*float*) – Transmissions power in watts.
- **enr_thresh** (*float*) – Minimum detectable target SNR (after coherent integration)
- **pulse_length** (*float*) – Length of transmission pulse.
- **ipp** (*float*) – Time between consecutive pulses.
- **n_ipp** (*int*) – Number of pulses to coherently integrate.
- **coh_int_bandwidth** (*float*) – Effective bandwidth of receiver noise after coherent integration.
- **extra_scans** (*list*) – List of additional observation schemes the transmitter will switch between, i.e. instances of `radar_scans.radar_scan`.
- **scan** (*radar_scan*) – The main observation mode of the transmitter.
- **scan_controler** (*function*) – The scan_controler function takes the *antenna.AntennaTX* instance and the time as arguments. The function should, based on the time, return either the `antenna.AntennaTX.scan` attribute, or one of the scans in the list `antenna.AntennaTX.extra_scans` attribute. If the function pointer is set to `None`, it is assumed only one scan exists and by default `antenna.AntennaTX.scan` is returned.

get_pointing (*t*)

Return the instantanius pointing of the TX antenna based on the currently running scan. Uses *antenna.AntennaTX.get_scan()*.

Parameters **t** (*float*) – Current time.

Returns Current TX-location in WGS84 ECEF and current pointing direction in ECEF. Both are 1-D arrays of 3 elements (lists, tuples or numpy.ndarray).

get_scan (*t*)

Return the current scan at a particular time.

Depending on the scan_controler function return the current observation schema that the system is running. If no scan_controler function is set, return the default scan.

The `antenna.AntennaTX.scan_controler` function takes the *antenna.AntennaTX* instance and a time as arguments.

Parameters **t** (*float*) – Current time.

Returns The currently running radar scan at time `t`.

Return type *RadarScan*

set_scan (*scan=None*, *extra_scans=None*, *scan_controler=None*)

Set the scan this TX-antenna will use.

Parameters

- **scan** (RadarScan) – The main observation mode of the transmitter. If not given or `None` the scan set at initialization will be used.
- **extra_scans** (*list*) – List of additional observation schemes the transmitter will switch between, i.e. instances of `radar_scans.radar_scan`.
- **scan_controler** (*function*) – The scan_controler function takes the *antenna.AntennaTX* instance and the time as arguments. The function should, based on the time, return either the `antenna.AntennaTX.scan` attribute, or one of the scans in the list `antenna.AntennaTX.extra_scans` attribute. If the function pointer is set to `None`, it is assumed only one scan exists and by default `antenna.AntennaTX.scan` is returned.

class antenna.**BeamPattern**(*gain_func*, *az0*, *el0*, *I_0*, *f*, *beam_name=''*)
Bases: `object`

Defines the radiation pattern of a radar station.

Parameters

- **I_0** (*float*) – Peak intensity of radiation pattern in linear scale, i.e. the peak gain.
- **f** (*float*) – Frequency of radiation pattern.
- **az0** (*float*) – Azimuth of pointing direction in dgreees.
- **el0** (*float*) – Elevation of pointing direction in degrees.
- **gain_func** (*function*) – Function describing gain as a function of incoming wave vector direction.
- **beam_name** (*str*) – Name of the radiation pattern model.

Attr numpy.array on_axis Cartesian vector in ECEF describing pointing direction.

Variables

- **I_0** (*float*) – Peak intensity of radiation pattern in linear scale, i.e. the peak gain.
- **f** (*float*) – Frequency of radiation pattern.
- **az0** (*float*) – Azimuth of pointing direction in dgreees.
- **el0** (*float*) – Elevation of pointing direction in degrees.
- **on_axis** (*numpy.array*) – Cartesian vector in local coordinates describing pointing direction.
- **gain_func** (*function*) – Function describing gain as a function of incoming wave vector direction.
- **beam_name** (*str*) – Name of the radiation pattern model.

angle(*az*, *el*)
Get angle between azimuth and elevation and pointing direction.

Parameters

- **az** (*float*) – Azimuth in dgreees east of north to measure from.
- **el** (*float*) – Elevation in degrees from horizon to measure from.

Returns Angle in degrees.

Return type float

angle_k(*k*)
Get angle between azimuth and elevation and pointing direction.

Parameters **k** (*numpy.array*) – Direction to evaluate angle to.

Returns Angle in degrees.

Return type float

copy()
Return a copy of the current instance.

gain(*k*)
Return the gain using the gain-function. The gain function may change gain result for a specific direction based on the instance state, i.e. pointing direction.

Parameters **k** (*numpy.array*) – Direction in local coordinates to evaluate gain in.

Return float Gain evaluated using current configuration.

point(*az0*, *el0*)
Point beam towards azimuth and elevation coordinate.

Parameters

- **az0** (*float*) – Azimuth of pointing direction in dgreees east of north.
- **el0** (*float*) – Elevation of pointing direction in degrees from horizon.

point_k0(*k0*)
Point beam in local direction.

Parameters **k0** (*numpy.ndarray*) – Pointing direction in local coordinates.

antenna.**full_gain2inst_gain**(*gain*, *groups*, *N_IPP*, *IPP_scale=1.0*, *units='dB'*)
Using pulse encoding schema, subgroup setup and coherrent integration setup; convert from coherrently integrated gain to instantanius gain.

Parameters

- **gain** (*float*) – Coherrently integrated gain, linear units or in dB.
- **groups** (*int*) – Number of subgroups from witch signals are coherrently combined, assumes subgroups are identical.
- **N_IPP** (*int*) – Number of pulses to coherrently integrate.
- **IPP_scale** (*float*) – Scale the IPP effective length in case e.g. the IPP is the same but the actual TX length is lowered.
- **units** (*str*) – If string equals 'dB', assume input and output units should be dB, else use linear scale.

Return float Instantanius gain, linear units or in dB.

antenna.**inst_gain2full_gain**(*gain*, *groups*, *N_IPP*, *IPP_scale=1.0*, *units='dB'*)
Using pulse encoding schema, subgroup setup and coherrent integration setup; convert from instantanius gain to coherrently integrated gain.

Parameters

- **gain** (*float*) – Instantanius gain, linear units or in dB.
- **groups** (*int*) – Number of subgroups from witch signals are coherrently combined, assumes subgroups are identical.
- **N_IPP** (*int*) – Number of pulses to coherrently integrate.

- **IPP_scale** (*float*) – Scale the IPP effective length in case e.g. the IPP is the same but the actual TX length is lowered.
- **units** (*str*) – If string equals 'dB', assume input and output units should be dB, else use linear scale.

Return float Gain after coherent integration, linear units or in dB.

antenna.**plot_gain**(*beam*, *res=1000*, *min_el=0.0*)
Plot the gain of a beam patterns as a function of elevation at 0° degrees azimuth.

Parameters

- **beam** (BeamPattern) – Beam pattern to plot.
- **res** (*int*) – Number of points to devide the set elevation range into.
- **min_el** (*float*) – Minimum elevation in degrees, elevation range is from this number to 90°.

antenna.**plot_gain3d**(*beam*, *res=200*, *min_el=0.0*)
Creates a 3d plot of the beam-patters as a function of azimuth and elevation in terms of wave vector ground projection coordinates.

Parameters

- **beam** (BeamPattern) – Beam pattern to plot.
- **res** (*int*) – Number of points to devide the wave vector x and y component range into, total number of caluclation points is the square of this number.
- **min_el** (*float*) – Minimum elevation in degrees, elevation range is from this number to 90°. This number defines the half the length of the square that the gain is calculated over, i.e. $\cos(el_{min})$.

antenna.**plot_gain_heatmap**(*beam*, *res=201*, *min_el=0.0*, *title=None*, *title_size=28*, *ax=None*)
Creates a heatmap of the beam-patters as a function of azimuth and elevation in terms of wave vector ground projection coordinates.

Parameters

- **beam** (BeamPattern) – Beam pattern to plot.
- **res** (*int*) – Number of points to devide the wave vector x and y component range into, total number of caluclation points is the square of this number.
- **min_el** (*float*) – Minimum elevation in degrees, elevation range is from this number to 90°. This number defines the half the length of the square that the gain is calculated over, i.e. $\cos(el_{min})$.

antenna.**plot_gains**(*beams*, *res=1000*, *min_el=0.0*, *alpha=0.5*)
Plot the gain of a list of beam patterns as a function of elevation at 0° degrees azimuth.

Parameters

- **beams** (*list*) – List of instances of *antenna.BeamPattern*.
- **res** (*int*) – Number of points to devide the set elevation range into.
- **min_el** (*float*) – Minimum elevation in degrees, elevation range is from this number to 90°.

7.2.3 base_propagator

A parent class used for interfacing any propagator.

class propagator_base.**PropagatorBase**

Bases: `object`

get_orbit (*t*, *a*, *e*, *inc*, *raan*, *aop*, *mu0*, *mjd0*, ***kwargs*)

Propagate a Keplerian state forward in time.

This function uses key-word argument to supply additional information to the propagator, such as area or mass.

It is a good idea to only implement *propagator_base.PropagatorBase.get_orbit()* or *propagator_base.PropagatorBase.get_orbit_cart()* and then link one to the other by simply using *dpt_tools.kep2cart()* or *dpt_tools.cart2kep()*.

The coordinate frames used should be documented in the child class docstring.

SI units are assumed unless implementation states otherwise.

Parameters

- **t** (*float/list/numpy.ndarray*) – Time in seconds to propagate relative the initial state epoch.
- **mjd0** (*float*) – The epoch of the initial state in fractional Julian Days.
- **a** (*float*) – Semi-major axis
- **e** (*float*) – Eccentricity
- **inc** (*float*) – Inclination
- **aop** (*float*) – Argument of perihelion
- **raan** (*float*) – Longitude (right ascension) of ascending node
- **mu0** (*float*) – Mean anomaly

Returns 6-D Cartesian state vector in SI-units.

get_orbit_cart (*t*, *x*, *y*, *z*, *vx*, *vy*, *vz*, *mjd0*, ***kwargs*)

Propagate a Cartesian state forward in time.

This function uses key-word argument to supply additional information to the propagator, such as area or mass.

It is a good idea to only implement *propagator_base.PropagatorBase.get_orbit()* or *propagator_base.PropagatorBase.get_orbit_cart()* and then link one to the other by simply using *dpt_tools.kep2cart()* or *dpt_tools.cart2kep()*.

The coordinate frames used should be documented in the child class docstring.

Parameters

- **t** (*float/list/numpy.ndarray*) – Time in seconds to propagate relative the initial state epoch.
- **mjd0** (*float*) – The epoch of the initial state in fractional Julian Days.
- **x** (*float*) – X position
- **y** (*float*) – Y position
- **z** (*float*) – Z position

- **vx** (*float*) – X-direction velocity
- **vy** (*float*) – Y-direction velocity
- **vz** (*float*) – Z-direction velocity

Returns 6-D Cartesian state vector in SI-units.

`propagator_base.`**`plot_orbit_3d`**`(`*ecefs*`)`

Plot a set of ECEF's in 3D using matplotlib.

7.2.4 population

Defines a population of space objects in the form of a class.

`class` `population.`**`Population`**`(`*name='Unnamed population', extra_columns=[], dtypes=[], space_object_uses=[], propagator=<class 'propagator_sgp4.PropagatorSGP4'>, propagator_options={}*`)`

Encapsulates a population of space objects as an array and functions for returning instances of space objects.

Default columns:

- 0: oid - Object ID
- 1: a - Semi-major axis in km
- 2: e - Eccentricity
- 3: i - Inclination in degrees
- 4: raan - Right Ascension of ascending node in degrees
- 5: aop - Argument of perihelion in degrees
- 6: mu0 - Mean anoamly in degrees
- 7: mjd0 - Epoch of object given in Modified Julian Days

Any column that is added will have its name used in initializing the Space object.

A population's column data can also be accessed as a python dictionary or a table according to row number, e.g.

```
#this returns all Right Ascension of ascending node as a numpy vector
vector = my_population['raan']

#this gets row number 3 (since we use 0 indexing)
row = my_population[2]
```

but it is also configured to be able to try to convert to uniform type array and perform numpy like slices. If a column data type cannot be converted to the default data type numpy.nan is inserted instead.

```
#This will convert the internal data structure to a uniform type array and select␣
↪all rows and columns 4 and onwards. This is time-consuming on large populations␣
↪as it actually copies to data.
vector = my_population[:,4:]

# This is significantly faster as a single column is easy to extract and no␣
↪conversion is needed
data_point = my_population[123,2]
```

This indexing system can also be used for data manipulation:

```
my_population['raan'] = vector
my_population[2] = row
my_population[:,4:] = matrix
my_population[123,2] = 2.3
my_population[123,11] = 'test'
```

Notice that in the above example the value to be assigned always has the correct size corresponding to the index and slices, a statement like `x[:,3:7] = 3` is not possible, instead one would write `x[:,3:7] = np.full((len(pop), 4), 3.0, dtype='f')`.

Variables

- **objs** (*numpy.ndarray*) – Array containing population data. Rows correspond to objects and columns to variables.
- **name** (*str*) – Name of population.
- **header** (*list*) – List of strings containing column descriptions.
- **space_object_uses** (*list*) – List of booleans describing what columns should be included when initializing a space object. This allows for extra data to be stored in the population without passing it to the space object.
- **propagator** (PropagatorBase) – Propagator class pointer used for *space_object.SpaceObject*.
- **propagator_options** (*dict*) – Propagator initialization keyword arguments.

Parameters

- **name** (*str*) – Name of population.
- **extra_columns** (*list*) – List of strings containing column descriptions for addition data besides the default columns.
- **dtypes** (*list*) – List of strings containing numpy data type description. Defaults to 'f'.
- **space_object_uses** (*list*) – List of booleans describing what columns should be included when initializing a space object. This allows for extra data to be stored in the population without passing it to the space object.
- **propagator** (PropagatorBase) – Propagator class pointer used for *space_object.SpaceObject*.
- **propagator_options** (*dict*) – Propagator initialization keyword arguments.

add_column (*name*, *dtype='float64'*, *space_object_uses=False*)
Add a column to the population data.

allocate (*length*)
Allocate the internal data array for assignment of objects.

Warning: This removes all internal data.

Example:

Create a population with two objects. Here the `load_data` function is a fictional function that creates a row with the needed data.

```
from population import Population
from my_data_loader import load_data

my_pop = Population(
```

(continues on next page)

(continued from previous page)

```
    name='two objects',
    extra_columns = ['m', 'color'],
    dtypes = ['Float64', 'U20'],
    space_object_uses = [True, False],
)

print(len(my_pop)) #will output 0
my_pop.allocate(2)
print(len(my_pop)) #will output 2

my_pop.objs[0] = load_data('obj1')
my_pop.objs[1] = load_data('obj2')
```

delete(*inds*)
Remove the rows according to the given indices. Supports single index, iterable of indices and slices.

filter(*col, fun*)
Filters the population using a boolean function, keeping true values.

Parameters

- **col** (*str*) – Column to filter, must match exactly one entry in the `header` attribute.
- **fun** (*function*) – Function that returns boolean array used for filtering.

Example:

Filter Master population keeping only objects below 45.0 degrees inclination.

```
from population_library import master_catalog

master = master_catalog()
master.filter(
    col='i',
    fun=lambda inc: inc < 45.0,
)
```

get_all_orbits(*order_angs=False*)
Get the orbital elements for all rows from internal data array.

Parameters **order_angs** (*bool*) – Order the orbital element angles according to aop before raan or not.

get_object(*n*)
Get the one row from the population as a *space_object.SpaceObject* instance.

get_orbit(*n, order_angs=False*)
Get the orbital elements for one row from internal data array.

Parameters

- **n** (*int*) – Row number.
- **order_angs** (*bool*) – Order the orbital element angles according to aop before raan or not.

get_states(*M_cent=5.972370723755184e+24*)
Use the orbital parameters and get the state.

classmethod load(*fname, propagator=<class 'propagator_sgp4.PropagatorSGP4'>, propagator_options={}*)

next()

object_generator()
Return a generator that iterates trough the entire population returning space objects.

plot_distribution(*dist*, *label=None*, *logx=False*, *logy=False*, *log_freq=False*)
Plot the distribution of parameter(s) or all orbits of this population.

Parameters

- **dist** (*str/list*) – Name of parameter as given by `population.header` or `'orbits'` to plot all orbits. If a list, length of list must be exactly 2 and will produce a 2d distribution instead.
- **label** (*str/list*) – Used if parameter(s) distribution is plotted to label the axis.
- **logx** (*bool*) – Determines if x-axis is logarithmic or not.
- **logy** (*bool*) – Determines if y-axis is logarithmic or not.
- **log_freq** (*bool*) – Determines if frequency is logarithmic or not.

print_row(*n*)
Print a specific row with Header information.

save(*fname*)

shape
This is the shape of the internal data matrix

7.2.5 radar_scans

Defines what a radar observation schema is in the form of a class.

A scan needs to return:

- radar position and radar pointing direction at any given time t (seconds since epoch)
- short name of scan
- title describing the scan

class radar_scans.**RadarScan**(*lat*, *lon*, *alt*, *pointing_function*, *min_dwell_time*, *pointing_coord='azel'*, *name='generic scan'*)
Bases: `object`

Encapsulates the observation schema of a radar system, i.e. its “scan”.

Variables

- **_lat** (*float*) – Geographical latitude of radar system in decimal degrees (North+).
- **_lon** (*float*) – Geographical longitude of radar system in decimal degrees (East+).
- **_alt** (*float*) – Geographical altitude above geoid surface of radar system in meter.
- **_pointing_function** (*function*) – A function that takes in time as the first argument and then any number of keyword arguments and returns the pointing of the radar in a system specified by `pointing_coord`.
- **_pointing_coord** (*str*) – The coordinate system used by the `_pointing_function`, may be ‘azel’ or ‘ned’.
- **name** (*str*) – Name of the scan.

- **_function_data** (*dict*) – A dictionary contaning the data to be expanded as keyword parameters to the `_pointing_function`.
- **_info_str** (*str*) – A string describing the scan.
- **_scan_time** (*float*) – If the scan has a repeating deterministic sequence, it is he time it takes to complete one sequence.
- **_pulse_n** (*float*) – Number of pulses in a repeating pulse sequence.
- **_min_el** (*float*) – Minimum elevation of the scanning sequence.

Parameters

- **lat** (*float*) – Geographical latitude of radar system in decimal degrees (North+).
- **lon** (*float*) – Geographical longitude of radar system in decimal degrees (East+).
- **alt** (*float*) – Geographical altitude above geoid surface of radar system in meter.
- **pointing_function** (*float*) – A function that takes in time as the first argument and then any number of keyword arguments and returns the pointing of the radar in a system specified by `pointing_coord`.
- **pointing_coord** (*str*) – The coordinate system used by the `_pointing_function`, may be 'azel' or 'ned'.
- **name** (*str*) – Name of the scan.

Pointing function:

The pointing function must follow the following standard:

- Take in time in seconds past reference epoch in seconds as first argument
- Take any number of keyword arguments, these arguments must be defined in the `_function_data` dictionary.
- It must return the pointing coordinates as a object with get-item implemented (list, tuple, 1-D numpy array, ect) of 3 elements.
- Units are in meters or degrees.

Example pointing function:

```
import numpy as np

def point_east_west_fence(t, dwell_time, angles):
    """Pointing function for a east-to-west fence scan returning pointing␣
↪coordinates in a NED (North-East-Down) cartesian coordinate system.
    """
        ind = np.floor(t/dwell_time % len(angles))
        angle = int(ind)
        e = np.cos(np.pi*angle/180.0)
        d = -np.sin(np.pi*angle/180.0)
        return 0.0, e, d
```

Coordinate systems:

azel Azimuth and Elevation in degrees east of north and above horizon.

ned Cartesian coordinates in North, East, Down in meters.

enu Cartesian coordinates in East, North, Up in meters.

antenna_pointing(*t*)
Returns the instantaneous WGS84 ECEF pointing direction and the radar geographical location in WGS84 ECEF coordinates.

Parameters **t** (*float*) – Seconds past a reference epoch to retrieve the pointing at.

check_tx_compatibility(*tx*)
Checks if the transmitting antenna pusle pattern and coherrent integration schema is compatible with the observation schema. Raises an Exception if not.

Parameters **tx** (AntennaTX) – The antenna that should perform this scan.

copy()
Return a copy of the current instance of *radar_scans.RadarScan*.

dwell_time()
If dwell time is a applicable concept for this scan, return that time.

info()
Return a descriptive string.

keyword_arguments(***kw*)
Adds or modifies all the input keyword arguments of this call to the function data used in calling the pointing function.

local_pointing(*t*)
Returns the instantaneous pointing in local coordinates (ENU).

Parameters **t** (*float*) – Seconds past a reference epoch to retrieve the pointing at.

min_dwell_time
The dwell time of the scan. If there are dynamic dwell times, this is the minimum dwell time.

set_tx_location(*tx*)
Set the geographic location of this scan to coencide with the input *antenna.AntennaTX*.

Parameters **tx** (AntennaTX) – The antenna that should perform this scan.

radar_scans.**plot_radar_scan**(*SC*, *earth=False*)
Plot a full cycle of the scan pattern based on the `_scan_time` and the `_function_data['dwell_time']` variable.

Parameters

- **SC** (RadarScan) – Scan to plot.
- **earth** (*bool*) – Plot the surface of the Earth.

radar_scans.**plot_radar_scan_movie**(*SC*, *earth=False*, *rotate=False*, *save_str=''*)
Create a animation of the scan pattern based on the `_scan_time` and the `_function_data['dwell_time']` variable.

Parameters

- **SC** (RadarScan) – Scan to plot.
- **earth** (*bool*) – Plot the surface of the Earth.
- **save_str** (*str*) – String of path to output movie file. Requers an avalible ffmpeg encoder on the system. If string is empty no movie is saved.

7.2.6 space_object

Defines a space object. Encapsulates orbital elements, propagation and related methods.

Example:

Using space object for propagation.

```
import numpy as n
import matplotlib.pyplot as plt
import SpaceObject as so
import plothelp

o = so.SpaceObject(
    a=7000, e=0.0, i=69,
    raan=0, aop=0, mu0=0,
    C_D=2.3, A=1.0, m=1.0,
    C_R=1.0, oid=42,
    mjd0=57125.7729,
)

t=n.linspace(0,24*3600,num=1000, dtype=n.float)
ecefs=o.get_state(t)

fig = plt.figure(figsize=(15,15))
ax = fig.add_subplot(111, projection='3d')
ax.view_init(15, 5)
plothelp.draw_earth_grid(ax)

ax.plot(ecefs[0,:],ecefs[1,:],ecefs[2,:],'-',alpha=0.5,color="black")
plt.title("Orbital propagation test")
plt.show()
```

Using space object with a different propagator.

```
import numpy as n
import matplotlib.pyplot as plt
import SpaceObject as so
import plothelp
from propagator_orekit import PropagatorOrekit

o = so.SpaceObject(
    a=7000, e=0.0, i=69,
    raan=0, aop=0, mu0=0,
    C_D=2.3, A=1.0, m=1.0,
    C_R=1.0, oid=42,
    mjd0=57125.7729,
    propagator = PropagatorOrekit,
    propagator_options = {
        'in_frame': 'TEME',
        'out_frame': 'ITRF',
    },
)

t=n.linspace(0,24*3600,num=1000, dtype=n.float)
ecefs=o.get_state(t)

fig = plt.figure(figsize=(15,15))
```

(continues on next page)

(continued from previous page)

```
ax = fig.add_subplot(111, projection='3d')
ax.view_init(15, 5)
plothelp.draw_earth_grid(ax)

ax.plot(ecefs[0,:],ecefs[1,:],ecefs[2,:],'-',alpha=0.5,color="black")
plt.title("Orbital propagation test")
plt.show()
```

`space_object.`**`M_e = 5.972e+24`**
float: Mass of the Earth

`space_object.`**`R_E = 6371000.0`**
float: Radius of the Earth

`class` `space_object.`**`SpaceObject`**(*a, e, i, raan, aop, mu0, d=0.01, C_D=2.3, A=1.0, m=1.0, mjd0=57125.7729, oid=42, M_cent=5.972e+24, C_R=1.0, propagator=<class 'propagator_sgp4.PropagatorSGP4'>, propagator_options={}, **kwargs*)

Bases: `object`

Encapsulates a object in space who's dynamics is governed in time by a propagator.

The relation between the Cartesian and Kepler states are a direct transformation according to the below orientation rules. If the Kepler elements are given in a Inertial system, to reference the Cartesian to a Earth-fixed system a earth rotation transformation must be applied externally of the method.

Orientation of the ellipse in the coordinate system:

- For zero inclination i: the ellipse is located in the x-y plane.
- The direction of motion as True anoamly ν: increases for a zero inclination i: orbit is anti-coockwise, i.e. from +x towards +y.
- If the eccentricity e: is increased, the periapsis will lie in +x direction.
- If the inclination i: is increased, the ellipse will rotate around the x-axis, so that +y is rotated toward +z.
- An increase in Longitude of ascending node Ω: corresponds to a rotation around the z-axis so that +x is rotated toward +y.
- Changing argument of perihelion ω: will not change the plane of the orbit, it will rotate the orbit in the plane.
- The periapsis is shifted in the direction of motion.
- True anomaly measures from the +x axis, i.e $\nu = 0$ is located at periapsis and $\nu = \pi$ at apoapsis.
- All anomalies and orientation angles reach between 0 and 2π

Reference: "Orbital Motion" by A.E. Roy.

Variables:

- a: Semi-major axis
- e: Eccentricity
- i: Inclination
- ω: Argument of perihelion
- Ω: Longitude of ascending node

- ν: True anoamly

Uses:

- *kep2cart()*
- *cart2kep()*
- *mean2true()*
- *true2mean()*
- *rot_mat_z()*
- *gmst()*
- *write_oem()*

Variables

- **a** (*float*) – Semi-major axis [km]
- **e** (*float*) – Eccentricity
- **i** (*float*) – Inclination [deg]
- **aop** (*float*) – Argument of periapsis [deg]
- **raan** (*float*) – Right ascension of the ascending node [deg]
- **mu0** (*float*) – Mean anomaly [deg]
- **x** (*float*) – X position [km]
- **y** (*float*) – Y position [km]
- **z** (*float*) – Z position [km]
- **vx** (*float*) – X-direction velocity [km/s]
- **vy** (*float*) – Y-direction velocity [km/s]
- **vz** (*float*) – Z-direction velocity [km/s]
- **oid** (*int*) – Identifying object ID
- **C_D** (*float*) – Drag coefficient
- **C_R** (*float*) – Radiation pressure coefficient
- **A** (*float*) – Area [m^2]
- **m** (*float*) – Mass [kg]
- **mjd0** (*float*) – Epoch for state [BC-relative JD]
- **prop** (*float*) – Propagator instance, child of `PropagatorBase`
- **d** (*float*) – Diameter [m]
- **M_cent** (*float*) – Mass of central body [kg]
- **state_cart** (*numpy.ndarray*) – 6-D vector containing the Cartesian state vector.
- **propagator_options** (*dict*) – Propagator initialization keyword arguments
- **kwargs** (*dict*) – All additional keyword arguments will be passed to the propagator call.

The constructor creates a space object using Kepler elements.

Parameters

- **a** (*float*) – Semi-major axis in km
- **e** (*float*) – Eccentricity
- **i** (*float*) – Inclination in degrees
- **aop** (*float*) – Argument of perigee in degrees
- **raan** (*float*) – Right ascension of the ascending node in degrees
- **mu0** (*float*) – Mean anomaly in degrees
- **C_D** (*float*) – Drag coefficient
- **C_R** (*float*) – Radiation pressure coefficient
- **A** (*float*) – Area in square meters
- **m** (*float*) – Mass in kg
- **mjd0** (*float*) – Epoch for state
- **oid** (*int*) – Identifying object ID
- **d** (*float*) – Diameter in meters
- **M_cent** (*float*) – Mass of central body
- **propagator** (PropagatorBase) – Propagator class pointer
- **propagator_options** (*dict*) – Propagator initialization keyword arguments
- **kwargs** (*dict*) – All additional keyword arguments will be passed to the propagator call.

classmethod cartesian(*x*, *y*, *z*, *vx*, *vy*, *vz*, *d=0.01*, *C_D=2.3*, *A=1.0*, *m=1.0*, *mjd0=57125.7729*, *oid=42*, *M_cent=5.972e+24*, *C_R=1.0*, *propagator=<class 'propagator_sgp4.PropagatorSGP4'>*, *propagator_options={}*, ***kwargs*)

Creates a space object using Cartesian elements.

Parameters

- **x** (*float*) – X position in km
- **y** (*float*) – Y position in km
- **z** (*float*) – Z position in km
- **vx** (*float*) – X-direction velocity in km/s
- **vy** (*float*) – Y-direction velocity in km/s
- **vz** (*float*) – Z-direction velocity in km/s
- **C_D** (*float*) – Drag coefficient
- **C_R** (*float*) – Radiation pressure coefficient
- **A** (*float*) – Area
- **m** (*float*) – Mass in kg
- **mjd0** (*float*) – Epoch for state
- **oid** (*int*) – Identifying object ID
- **d** (*float*) – Diameter in meters
- **M_cent** (*float*) – Mass of central body

- **propagator** (PropagatorBase) – Propagator class pointer
- **propagator_options** (*dict*) – Propagator initialization keyword arguments
- **kwargs** (*dict*) – All additional keyword arguments will be passed to the propagator call.

diam

get_orbit(*t*)

Gets ECEF position at specified times using propagator instance.

Parameters **t** (*float/list/numpy.ndarray*) – Time relative epoch in seconds.

Returns Array of positions as a function of time.

Return type numpy.ndarray of size (3,len(t))

get_state(*t*)

Gets ECEF state at specified times using propagator instance.

Parameters **t** (*float/list/numpy.ndarray*) – Time relative epoch in seconds.

Returns Array of state (position and velocity) as a function of time.

Return type numpy.ndarray of size (6,len(t))

update(***kwargs*)

Updates the orbital elements and Cartesian state vector of the space object.

Can update any of the related state parameters, all others will automatically update.

Cannot update Keplerian and Cartesian elements simultaneously.

Parameters

- **a** (*float*) – Semi-major axis in km
- **e** (*float*) – Eccentricity
- **i** (*float*) – Inclination in degrees
- **aop** (*float*) – Argument of perigee in degrees
- **raan** (*float*) – Right ascension of the ascending node in degrees
- **mu0** (*float*) – Mean anomaly in degrees
- **x** (*float*) – X position in km
- **y** (*float*) – Y position in km
- **z** (*float*) – Z position in km
- **vx** (*float*) – X-direction velocity in km/s
- **vy** (*float*) – Y-direction velocity in km/s
- **vz** (*float*) – Z-direction velocity in km/s

write_oem(*t0*, *t1*, *n_items*, *fname*)

Writes OEM format file of orbital state for specific time interval. The states are linearly spaced in the specified time interval.

Parameters

- **t0** (*float*) – Start time.
- **t1** (*float*) – End time.

- **n_items** (*int*) – State points between start and end times.

7.2.7 catalogue

Catalogue class.

NOTES: If a object is detected it automatically produces a tracklet with one point (the point of detection) It can then get more tracklet points from the scheduler

We will have to think like this: * we run 1 set of obsrevation confugrations for one certain time * if a object is discovered, it can only get tracklet points that pass * Correlation between unknown objects will be AFTERWARDS, not in read time, thus it can be "rediscovered" * These "rediscoveries" will help imporve orbital elelemnts when it is added to the catalouge.

class catalogue.**Catalogue**(*population*, *known=False*)

# TODO: Write proper documentation for this class.

** BELOW IS OLD DOCS ** _detections format: [the object id] -> dict dict: "t0" initial detection

"t1" passes below horizon "snr" the snr's "tm" time of max SNR

each dict is a vector where one entry is one detection e.g _detections[obj id 4]['t0'][detection 2]

_maintinence format:, None indicate fail at that slot [the object id] -> dict dict: "t" list of all above to below horizon rimes, tx lst ['t'][tx 0][pass 2][above horizon time = 0, below = 1]

"snr" the list of max snrs of all rx tx pairs, i,e ["snr"][tx 0][pass 0][rx 1][0=SNR,1=time]

each dict is a vector where one entry is one detection e.g _maintinence[obj id 4]['t'][tx 0][pass 2][0] = above horizon t

track format [track nr], is list: 0 : t0 (scan: detection time, track: above horizon) 1 : dt (time untill horizon) 2 : detected/measured? 3 : SNR dB (scan: best detection posibility, track: peak snr) 4 : OID [not pop-id] 5 : number of baselines, e.g. 3=tristatic 6 : track is maintenance "track" or discovery "scan" 7 : time of SNR dB (col 3)

e.g _tracks[track nr 4][3] = SNR dB

_discovered format: [object id] [True or false, track number] e.g _discovered[object id 4][1] = track of detection

_tracklets format: rows = tracks

cols: ...fnames... [one col for each name]

#known objects DO NOT NEED TO BE SCANED FOR

add_tracklet(***kwargs*)

Add a tracklet to the internal list.

add_tracks(*num*, *data=None*)

Add more tracks to data array.

compile_tracks(*radar*, *t0*, *t1*, *radar_control=None*)

Takes a radar system and uses the cashed maintenance and detections data to fill the track-data array.

detection_summary()

detection_summary_plot(*save_folder=None*)

detection_tracks_plot(*save_folder=None*)

classmethod from_file(*population*, *fname*)

get_orbit(*ind*, *t0*, *t1*)

Get the orbit-determination for object considering tracks between certain times.

load(*fname*)
: Create a instance of Catalogue using a saved file and a population.

maintain(*inds*)
: Set object(s) to be maintained.

maintenance_tracks_plot(*save_folder=None*)

maintinece_summary()
: Compute summary statistics about maintenance.

maintinece_summary_plot(*save_folder=None*)

plots(*save_folder=None*)

save(*fname*)
: Save all data related to the catalog to a hdf5 file.

track_statistics()

track_statistics_plot(*save_folder=None*)

7.3 Function modules

`debris`	Radar signal to noise calculations for hard targets.
`ccsds_write`	Simple CCSDS TDM file writer.
`coord`	Collection of common coordinate transformations.
`dpt_tools`	Functions from Daniel's-Python-tools package
`TLE_tools`	Collection of useful functions for handling TLE's.
`population_filter`	Investigate what fraction of objects can be detected with a radar system.
`ray_trace`	
`simulate_scan`	Simulate discovery observations with user-defined scan pattern.
`simulate_tracking`	Functions for simulating the tracking of an object in space.
`simulate_tracklet`	Given scheduled observations of an object simulate the generated tracklet-data.
`simulate_scaning_snr`	Functions for single object propagation and SNR examination.
`logging_setup`	Sets up a logging framework that can be imported and used anywhere.
`orbit_accuracy`	Linearized error determination for orbital elements.
`orbital_estimation`	Estimates a space objects state vector from a set of ranges and range-rates.
`plothelp`	Functions for making plots quicker.
`lgeom`	Collection of simple geometric functions.
`correlator`	Correlate measurement time series with a population of objects to find the best match.

7.3.1 debris

Radar signal to noise calculations for hard targets.

Hard target radar echo signal to noise calculations Range and range-rate error analysis `debris.`**`debug_bw_sweep`**`(`*bw=array([10000., 564444.44444444, 1118888.88888889, 1673333.33333333, 2227777.77777778, 2782222.22222222, 3336666.66666667, 3891111.11111111, 4445555.55555556, 5000000.]), txlen=1000.0, enr=100.0, n_ipp=20, ipp=0.02*`)`

`debris.`**`debug_debris_filter`**`()`

Debug plot

Simulate the effect of the planned strong SNR satellite filter for EISCAT 3D.

`debris.`**`debug_enr_sweep`**`(`*enrs=array([1.00000000e+00, 4.64158883e+00, 2.15443469e+01, 1.00000000e+02, 4.64158883e+02, 2.15443469e+03, 1.00000000e+04, 4.64158883e+04, 2.15443469e+05, 1.00000000e+06]), txlen=1000.0, bw=1000000.0, n_ipp=10, ipp=0.02*`)`

Debug plot

Sweep through different energy-to-noise ratios and determine range and Doppler error variance.

`debris.`**`debug_ipp_sweep`**`(`*n_ipps=array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]), bw=1000000.0, enr=1000.0, ipp=0.02, txlen=2000.0*`)`

Debug IPP sweep plot.

Determine the measurement variance of range and range-rate estimates for a different number of transmit pulses that are coherently integrated.

`debris.`**`debug_rcs_sweep`**`()`

Debug plot.

Sweep through different diameters and calculate radar cross-section. The aim is to validate the perfectly conducting sphere model. Plot result.

`debris.`**`debug_tx_len_sweep`**`(`*tlen=array([10., 231.11111111, 452.22222222, 673.33333333, 894.44444444, 1115.55555556, 1336.66666667, 1557.77777778, 1778.88888889, 2000.]), bw=1000000.0, enr=1000.0, n_ipp=10, ipp=0.02*`)`

Debug plot

Sweep through different transmit pulse lengths and determine the measurement variance of range and range-rate estimates. Consider all echoes to have constant SNR

`debris.`**`debug_tx_len_sweep2`**`(`*tlen=array([10., 231.11111111, 452.22222222, 673.33333333, 894.44444444, 1115.55555556, 1336.66666667, 1557.77777778, 1778.88888889, 2000.]), bw=1000000.0, enr0=1000.0, n_ipp=20*`)`

Debug plot

Determine variance for different transmit pulse lengths with fixed peak power (less SNR for shorter pulses)

`debris.`**`hard_target_enr`**`(`*gain_tx, gain_rx, wavelength_m, power_tx, range_tx_m, range_rx_m, diameter_m=0.01, bandwidth=10, rx_noise_temp=150.0*`)`

Deterine the energy-to-noise ratio for a hard target (signal-to-noise ratio). Assume a smooth transition between Rayleigh and optical scattering.

gain_tx - transmit antenna gain, linear gain_rx - receiver antenna gain, linear wavelength_m - radar wavelength (meters) power_tx - transmit power (W) range_tx_m - range from transmitter to target (meters) range_rx_m - range from target to receiver (meters) diameter_m - object diameter (meters) bandwidth - effective receiver noise bandwidth for incoherent integration (tx_len*n_ipp/sample_rate) rx_noise_temp - receiver noise temperature (K) (Markkanen et.al., 1999)

`debris.`**`ionospheric_error_fun`**`()`

`debris.`**`lin_error`**`(`*enr=10.0, txlen=1000.0, n_ipp=10, ipp=0.02, bw=1000000.0, dr=10.0, ddop=1.0, sr=100000000.0, plot=False*`)`

Determine linearized errors for range and range-rate error for a psuedorandom binary phase coded radar transmit pulse with a certain transmit bandwidth (inverse of bit length)

calculate line of sight range and range-rate error, given ENR after coherent integration (pulse compression) txlen in microseconds.

Simulate a measurement and do a linearized error estimate.

debris.**precalculate_dr**(*txlen*, *bw*, *ipp=0.02*, *n_ipp=20*, *n_interp=20*)

debris.**simulate_echo**(*codes*, *t_vecs*, *bw=1000000.0*, *dop_Hz=0.0*, *range_m=1000.0*, *plot=False*, *sr=5000000.0*)

Simulate a radar echo with range and doppler. Use windowing to simulate a continuous finite bandwidth signal. This is used for linearized error estimates of range and range-rate errors.

debris.**target_diameter**(*gain_tx*, *gain_rx*, *wavelength_m*, *power_tx*, *range_tx_m*, *range_rx_m*, *enr=1.0*, *bandwidth=10.0*, *rx_noise_temp=150.0*)

Given SNR, determine the diameter of the target

determine smallest sphere detactable with a certain ENR Ignore Mie regime and use either optical or Rayleigh scatter.

(Markkanen et.al., 1999)

7.3.2 ccsds_write

Simple CCSDS TDM file writer.

EISCAT UHF antenna coordinates: 05 SENTER1 7725445.727 664387.767 71.354+14.2 (pedestal foot alt + antenna height)

WGS84: 69.58649229 N 19.22592538 E, 71.354+14.2 m (pedestal foot alt + antenna height) distance = d - 20.0 + 5.0 m [- subreflector round trip + elevation arm length] +/- 4 m

https://public.ccsds.org/Pubs/503x0b1c1.pdf

ccsds_write.**date2unix**(*year*, *month*, *day*, *hour*, *minute*, *second*)

Convert date to unix time in seconds

Parameters

- **year** (*int*) – Year as integer. Years preceding 1 A.D. should be 0 or negative. The year before 1 A.D. is 0, 10 B.C. is year -9.
- **month** (*int*) – Month as integer, Jan = 1, Feb. = 2, etc.
- **day** (*int*) – Day
- **hour** (*int*) – Hour in 24h format
- **minute** (*int*) – Minute
- **second** (*float*) – Second, may contain fractional part.

Returns Unix time in seconds

Return type float

ccsds_write.**read_ccsds**(*fname*)

Just get the range data # TODO: the rest

ccsds_write.**unix2date**(*unix*)

Convert unix time in seconds to UTC date datetime object

Parameters **unix** (*float*) – Unix time in seconds.

Returns Datetime object in UTC

Return type datetime.datetime

`ccsds_write.`**`unix2datestr`**(*unix*)

Convert unix time in seconds to Gregorian calendar UTC date-time formatted string

Uses:

- `unix2date()`

Parameters **unix** (*float*) – Unix time in seconds.

Returns Gregorian calendar UTC date-time formatted string

Return type str

`ccsds_write.`**`unix2datestrf`**(*x*)

Convert unix time in seconds to Gregorian calendar UTC date-time formatted string

Different implementation?

Uses:

- `unix2date()`

Parameters **unix** (*float*) – Unix time in seconds.

Returns Gregorian calendar UTC date-time formatted string

Return type str

`ccsds_write.`**`write_ccsds`**(*t_pulse, m_range, m_range_rate, m_range_std, m_range_rate_std, freq=230000000.0, tx_ecef=[0, 0, 0], rx_ecef=[0, 0, 0], tx='EISCAT UHF', rx='EISCAT UHF', oid='ERS-1', tdm_type='track', fname='track.tdm'*)

# TODO: Document function.

`ccsds_write.`**`write_oem`**(*t, state, oid=42, fname='oems/state.oem'*)

Uses a series of unix-times and state vectors in ITRF2000 to create a CCSDS OEM file.

Uses:

- `unix2datestrf()`
- `unix2datestr()`

Parameters

- **t** (*list/numpy.ndarray*) – Vector of unix-times
- **state** (*numpy.ndarray*) – 6-D states given in SI units in the ITRF2000 frame. Rows correspond to different states and columns to dimensions.
- **OID** (*int*) – Object ID and name (written in OEM as the same but with different formating)
- **fname** (*str*) – Output file-path for OEM.

7.3.3 coord

Collection of common coordinate transformations.

Some bits and pieces are from PySatel, but changed to work with numpy. There was a bug in geodetic2ecef in PySatel that is fixed. The other functions are implemented using information from wikipedia.

Juha Vierinen 2013. Daniel Kastinen 2019: Bug-fixes and updates

`coord.`**`angle_deg`**(*a*, *b*)
Angle in degrees between two vectors.

Parameters

- **a** (*numpy.ndarray*) – Vector a
- **a** – Vector b

Returns Angle in degrees between vectors a and b

Return type float

`coord.`**`az_el_r2geodetic`**(*obs_lat*, *obs_lon*, *obs_h*, *az*, *el*, *r*)
When given a observer lat,long,h and az,el and r, return lat,long,h of target

`coord.`**`azel_ecef`**(*lat*, *lon*, *alt*, *az*, *el*)
Radar pointing (az,el) degrees to unit vector in ECEF.

`coord.`**`azel_to_cart`**(*az*, *el*, *r*)
Convert from spherical coordinates to Cartesian in a degrees east of north and elevation fashion

`coord.`**`cart_to_azel`**(*vec*)
Convert from Cartesian coordinates to spherical in a degrees east of north and elevation fashion

`coord.`**`cbrt`**(*x*)

`coord.`**`ecef2geodetic`**(*x*, *y*, *z*)
Convert ECEF coordinates to geodetic. J. Zhu, "Conversion of Earth-centered Earth-fixed coordinates to geodetic coordinates," IEEE Transactions on Aerospace and Electronic Systems, vol. 30, pp. 957-961, 1994.

According to WGS84.

`coord.`**`ecef2local`**(*lat*, *lon*, *alt*, *x*, *y*, *z*)
NED (east,north,up) from ECEF coordinate system conversion.

`coord.`**`enu2ecef`**(*lat*, *lon*, *alt*, *n*, *e*, *u*)
NEU (north/east/up) to ECEF coordinate system conversion. Degrees are used.

`coord.`**`geodetic2ecef`**(*lat*, *lon*, *alt*)
Convert geodetic coordinates to ECEF. @lat, @lon in decimal degrees @alt in meters

Uses WGS84.

`coord.`**`geodetic_to_az_el_r`**(*obs_lat*, *obs_lon*, *obs_h*, *target_lat*, *target_lon*, *target_h*)
When given a observer lat,long,h and target lat,long,h, provide azimuth, elevation, and range to target

`coord.`**`ned2ecef`**(*lat*, *lon*, *alt*, *n*, *e*, *d*)
NED (north/east/down) to ECEF coordinate system conversion. Degrees are used.

7.3.4 dpt_tools

Functions from Daniel's-Python-tools package

This is a module to enable simple plotting with kwargs as configuration and includes coordinate transformations and other useful functions.

# TODO: Fix so orbits work with hyperbolic # TODO: Fix the 0-e 0-i errors

`dpt_tools.`**`M_earth = 5.972366228753626e+24`**

float: Mass of the Earth using the WGS84 convention.

`dpt_tools.`**`M_sol = 1.98847e+30`**

float: The mass of the sun $M_\odot$ given in kg, used in kepler transformations

`dpt_tools.`**`ascending_node_from_statevector`**(*sv*, *m*, ***kw*)

keywords include M_cent = 'central mass'

`dpt_tools.`**`cart2kep`**(*x*, *m=0.0*, *M_cent=1.98847e+30*, *radians=True*)

Converts set of Cartesian state vectors to set of Keplerian orbital elements.

Units: All units are SI-units: SI Units

Angles are by default given as radians, all angles are radians internally in functions, input and output angles can be both radians and degrees depending on the `radians` boolean.

Orientation of the ellipse in the coordinate system:

- For zero inclination i: the ellipse is located in the x-y plane.
- The direction of motion as True anoamly ν: increases for a zero inclination i: orbit is anti-coockwise, i.e. from +x towards +y.
- If the eccentricity e: is increased, the periapsis will lie in +x direction.
- If the inclination i: is increased, the ellipse will rotate around the x-axis, so that +y is rotated toward +z.
- An increase in Longitude of ascending node Ω: corresponds to a rotation around the z-axis so that +x is rotated toward +y.
- Changing argument of perihelion ω: will not change the plane of the orbit, it will rotate the orbit in the plane.
- The periapsis is shifted in the direction of motion.
- True anomaly measures from the +x axis, i.e $\nu = 0$ is located at periapsis and $\nu = \pi$ at apoapsis.
- All anomalies and orientation angles reach between 0 and 2π

Reference: "Orbital Motion" by A.E. Roy.

Constants:

- *`e_lim`*: Used to determine circular orbits
- *`i_lim`*: Used to determine non-inclined orbits

Variables:

- a: Semi-major axis
- e: Eccentricity
- i: Inclination
- ω: Argument of perihelion
- Ω: Longitude of ascending node
- ν: True anoamly

Parameters

- **x** (*numpy.ndarray*) – Cartesian state vectors where rows 1-6 correspond to x, y, z, v_x, v_y, v_z and columns correspond to different objects.
- **m** (*float/numpy.ndarray*) – Masses of objects. If m is a numpy vector of masses, the gravitational μ parameter will be calculated also as a vector.
- **M_cent** (*float*) – Is the mass of the central massive body, default value is the mass of the sun parameter in *M_sol*
- **radians** (*bool*) – If true radians are used, else all angles are given in degree.

Returns Keplerian orbital elements where rows 1-6 correspond to a, e, i, ω, Ω, ν and columns correspond to different objects.

Return type numpy.ndarray

Example:

Convert 1 AU distance object of Earth mass traveling at 30 km/s tangential velocity to Kepler elements.

```
import dpt_tools as dpt
import numpy as n
import scipy.constants as c

state = n.array([
  c.au*1.0, #x
  0, #y
  0, #z
  0, #vx
  30e3, #vy
  0, #vz
], dtype=n.float)

orbit = dpt.cart2kep(state, m=5.97237e24, M_cent=1.9885e30, radians=False)
print('Orbit: a={} AU, e={}'.format(orbit[0]/c.au, orbit[1]))
print('(i, omega, Omega, nu)={} deg '.format(orbit[2:]))
```

Reference: Daniel Kastinen Master Thesis: Meteors and Celestial Dynamics

dpt_tools.**date_to_jd**(*year*, *month*, *day*)

Convert a date to Julian Day.

Parameters

- **year** (*int*) – Year as integer. Years preceding 1 A.D. should be 0 or negative. The year before 1 A.D. is 0, 10 B.C. is year -9.
- **month** (*int*) – Month as integer, Jan = 1, Feb. = 2, etc.
- **day** (*float*) – Day, may contain fractional part.

Returns (float) Julian Day

Example:

Convert 6 a.m., February 17, 1985 to Julian Day

```
>>> date_to_jd(1985,2,17.25)
2446113.75
```

Reference: 'Practical Astronomy with your Calculator or Spreadsheet', 4th ed., Duffet-Smith and Zwart, 2011.

`dpt_tools.`**`e_lim = 1e-09`**
float: The limit on eccentricity below witch an orbit is considered circular

`dpt_tools.`**`eccentric2mean`**(*E*, *e*, *radians=True*)
Calculates the mean anomaly from the eccentric anomaly using Kepler equation.

Parameters

- **`E`** (*`float/numpy.ndarray`*) – Eccentric anomaly.
- **`e`** (*`float/numpy.ndarray`*) – Eccentricity of ellipse.
- **`radians`** (*`bool`*) – If true radians are used, else all angles are given in degrees

Returns Mean anomaly.

Return type numpy.ndarray or float

`dpt_tools.`**`eccentric2true`**(*E*, *e*, *radians=True*)
Calculates the true anomaly from the eccentric anomaly.

Parameters

- **`E`** (*`float/numpy.ndarray`*) – Eccentric anomaly.
- **`e`** (*`float/numpy.ndarray`*) – Eccentricity of ellipse.
- **`radians`** (*`bool`*) – If true radians are used, else all angles are given in degrees

Returns True anomaly.

Return type numpy.ndarray or float

`dpt_tools.`**`elliptic_radius`**(*E*, *a*, *e*, *radians=True*)
Calculates the distance between the left focus point of an ellipse and a point on the ellipse defined by the eccentric anomaly.

Parameters

- **`E`** (*`float/numpy.ndarray`*) – Eccentric anomaly.
- **`a`** (*`float/numpy.ndarray`*) – Semi-major axis of ellipse.
- **`e`** (*`float/numpy.ndarray`*) – Eccentricity of ellipse.
- **`radians`** (*`bool`*) – If true radians are used, else all angles are given in degrees

Returns Radius from left focus point.

Return type numpy.ndarray or float

`dpt_tools.`**`find_ascending_node_time`**(*a*, *e*, *aop*, *mu0*, *m*, *radians=False*)
Find the time past the crossing of the ascending node.

`dpt_tools.`**`gmst`**(*mjd_UT1*)
Returns the Greenwich Mean Sidereal Time (rotation of the earth) at a specific UTC Modified Julian Date. Defined as the hour angle between the meridian of Greenwich and mean equinox of date at 0 h UT1

Parameters **`mjd_UT1`** (*`float/numpy.ndarray`*) – UTC Modified Julian Date.

Returns Greenwich Mean Sidereal Time in radians between 0 and 2π.

Reference: Montenbruck & Gill: Satellite orbits

`dpt_tools.`**`gps0_tai = numpy.datetime64('1980-01-06T00:00:19')`**
numpy.datetime64: Epoch of GPS time, in TAI

`dpt_tools.`**`hist`**`(`*x*, ***options*`)`

This function creates a histogram plot with lots of nice pre-configured settings unless they are overridden

Parameters

- **x** (*numpy.ndarray*) – data to histogram over, if x is not a vector it is flattened
- **options** (*dict*) – All keyword arguments as a dictionary containing all the optional settings.

Currently the keyword arguments that can be used in the options:

bins [int] the number of bins

density [bool] convert counts to density in [0,1]

edges [float] bin edge line width, set to 0 to remove edges

title [string] the title of the plot

title_font_size [int] the title font size

xlabel [string] the label for the x axis

ylabel [string] the label for the y axis

tick_font_size [int] the axis tick font size

window [tuple/list] the size of the plot window in pixels (assuming dpi = 80)

save [string] will not display figure and will instead save it to this path

show [bool] if False will do draw() instead of show() allowing script to continue

plot [tuple] A tuple with the `(fig, ax)` objects from matplotlib. Then no new figure and axis will be created.

logx [bool] Determines if x-axis should be the logarithmic.

logy [bool] Determines if y-axis should be the logarithmic.

Example:

```
import dpt_tools as dpt
import numpy as np
np.random.seed(19680221)

x = 10*np.random.randn(1000)

dpt.hist(x,
   title = "My first plot",
)
```

`dpt_tools.`**`hist2d`**`(`*x*, *y*, ***options*`)`

This function creates a histogram plot with lots of nice pre-configured settings unless they are overridden

Parameters

- **x** (*numpy.ndarray*) – data to histogram over, if x is not a vector it is flattened
- **options** (*dict*) – All keyword arguments as a dictionary containing all the optional settings.

Currently the keyword arguments that can be used in the options:

bins [int] the number of bins

colormap [str] Name of colormap to use.

title [string] the title of the plot

title_font_size [int] the title font size

xlabel [string] the label for the x axis

ylabel [string] the label for the y axis

tick_font_size [int] the axis tick font size

window [tuple/list] the size of the plot window in pixels (assuming dpi = 80)

save [string] will not display figure and will instead save it to this path

show [bool] if False will do draw() instead of show() allowing script to continue

plot [tuple] A tuple with the `(fig, ax)` objects from matplotlib. Then no new figure and axis will be created.

logx [bool] Determines if x-axis should be the logarithmic.

logy [bool] Determines if y-axis should be the logarithmic.

log_freq [bool] Determines if frequency should be the logarithmic.

Example:

```
import dpt_tools as dpt
import numpy as np
np.random.seed(19680221)

x = 10*np.random.randn(1000)

dpt.hist(x,
   title = "My first plot",
)
```

`dpt_tools.`**`i_lim = 3.141592653589793e-09`**

float: The limit on inclination below witch an orbit is considered not inclined.

`dpt_tools.`**`jd_to_date`**(*jd*)

Convert Julian Day to date.

Parameters **jd** (*float*) – Julian Day

Returns Tuple consisting of year, month and day

Return type tuple

Return tuple:

year (int) Year as integer. Years preceding 1 A.D. should be 0 or negative. The year before 1 A.D. is 0, 10 B.C. is year -9.

month (int) Month as integer, Jan = 1, Feb. = 2, etc.

day (float) Day, may contain fractional part.

Example:

Convert Julian Day 2446113.75 to year, month, and day.

```
>>> jd_to_date(2446113.75)
(1985, 2, 17.25)
```

Reference: 'Practical Astronomy with your Calculator or Spreadsheet', 4th ed., Duffet-Smith and Zwart, 2011.

`dpt_tools.`**`jd_to_mjd`**(*jd*)

Convert Julian Date (relative 12h Jan 1, 4713 BC) to Modified Julian Date (relative 0h Nov 17, 1858)

`dpt_tools.`**`jd_to_unix`**(*jd_ut1*)

Convert JD UT1 time to Unix time

Constant is due to 0h Jan 1, 1970 = 2440587.5 JD

Parameters **`jd_ut1`** (*`float/numpy.ndarray`*) – Julian Date UT1

Returns Unix time in seconds

Return type float/numpy.ndarray

`dpt_tools.`**`kep2cart`**(*o*, *m=0.0*, *M_cent=1.98847e+30*, *radians=True*)

Converts set of Keplerian orbital elements to set of Cartesian state vectors.

Units: All units are SI-units: SI Units

Angles are by default given as radians, all angles are radians internally in functions, input and output angles can be both radians and degrees depending on the `radians` boolean.

To use custom units, simply change the definition of `mu = scipy.constants.G*(m + M_cent)` to an input parameter for the function as this is the only unit dependent calculation.

Orientation of the ellipse in the coordinate system:

- For zero inclination i: the ellipse is located in the x-y plane.
- The direction of motion as True anoamly ν: increases for a zero inclination i: orbit is anti-coockwise, i.e. from +x towards +y.
- If the eccentricity e: is increased, the periapsis will lie in +x direction.
- If the inclination i: is increased, the ellipse will rotate around the x-axis, so that +y is rotated toward +z.
- An increase in Longitude of ascending node Ω: corresponds to a rotation around the z-axis so that +x is rotated toward +y.
- Changing argument of perihelion ω: will not change the plane of the orbit, it will rotate the orbit in the plane.
- The periapsis is shifted in the direction of motion.

Reference: "Orbital Motion" by A.E. Roy.

Variables:

- a: Semi-major axis
- e: Eccentricity
- i: Inclination
- ω: Argument of perihelion
- Ω: Longitude of ascending node
- ν: True anoamly

Uses:

- *true2eccentric()*
- *elliptic_radius()*

Parameters

- **o** (*numpy.ndarray*) – Keplerian orbital elements where rows 1-6 correspond to a, e, i, ω, Ω, ν and columns correspond to different objects.
- **m** (*float/numpy.ndarray*) – Masses of objects. If m is a numpy vector of masses, the gravitational μ parameter will be calculated also as a vector.
- **M_cent** (*float*) – Is the mass of the central massive body, default value is the mass of the sun parameter in *M_sol*
- **radians** (*bool*) – If true radians are used, else all angles are given in degree.

Returns Cartesian state vectors where rows 1-6 correspond to x, y, z, v_x, v_y, v_z and columns correspond to different objects.

Return type numpy.ndarray

Example:

Convert Earth J2000.0 orbital parameters to Cartesian position.

```
import dpt_tools as dpt
import numpy as n
import scipy.constants as c

#Periapsis approx 3 Jan
orbit = n.array([
  c.au*1.000001018, #a
  0.0167086, #e
  7.155, #i
  288.1, #omega
  174.9, #Omega
  0.0, #nu
], dtype=n.float)

state = dpt.kep2cart(orbit, m=5.97237e24, M_cent=1.9885e30, radians=False)
print('Position: {} AU '.format(state[:3]/c.au))
print('Velocity: {} km/s '.format(state[3:]/1e3))
```

Reference: Daniel Kastinen Master Thesis: Meteors and Celestial Dynamics, "Orbital Motion" by A.E. Roy.

dpt_tools.**kepler_guess**(*M*, *e*)

Guess the initial iteration point for newtons method.

Parameters

- **M** (*float/numpy.ndarray*) – Mean anomaly.
- **e** (*float/numpy.ndarray*) – Eccentricity of ellipse.

Returns Guess for eccentric anomaly.

Return type numpy.ndarray or float

Reference: Esmaelzadeh, R., & Ghadiri, H. (2014). Appropriate starter for solving the Kepler's equation. International Journal of Computer Applications, 89(7).

`dpt_tools.`**`laguerre_solve_kepler`**(*E0*, *M*, *e*, *tol=1e-12*, *degree=5*)

Solve the Kepler equation using the The Laguerre Algorithm, a algorithm that guarantees global convergence. Adjusted for solving only real roots (non-hyperbolic orbits)

Absolute numerical tolerance is defined as $|f(E)| < tol$ where $f(E) = M - E + e\sin(E)$.

# TODO: implement hyperbolic solving.

Note: Choice of polynomial degree does not matter significantly for convergence rate.

Parameters

- **M** (*float*) – Initial guess for eccentric anomaly.
- **M** – Mean anomaly.
- **e** (*float*) – Eccentricity of ellipse.
- **tol** (*float*) – Absolute numerical tolerance eccentric anomaly.
- **degree** (*int*) – Polynomial degree in derivation of Laguerre Algorithm.

Returns Eccentric anomaly and number of iterations.

Return type tuple of (float, int)

Reference: Conway, B. A. (1986). An improved algorithm due to Laguerre for the solution of Kepler's equation. Celestial mechanics, 39(2), 199-211.

Example:

```
import dpt_tools as dpt
M = 3.14
e = 0.8

#Use mean anomaly as initial guess
E, iterations = dpt.laguerre_solve_kepler(
   E0 = M,
   M = M,
   e = e,
   tol = 1e-12,
)
```

`dpt_tools.`**`leapseconds = array(['1972-01-01T00:00:00.000000000', '1972-07-01T00:00:00.000000`**

numpy.ndarray: Leapseconds added since 1972.

Must be maintained manually.

Source: tai-utc

`dpt_tools.`**`leapseconds_before`**(*ytime*, *tai=False*)

Calculate the number of leapseconds has been added before given date.

`dpt_tools.`**`mean2eccentric`**(*M*, *e*, *tol=1e-12*, *radians=True*)

Calculates the eccentric anomaly from the mean anomaly by solving the Kepler equation.

Parameters

- **M** (*float/numpy.ndarray*) – Mean anomaly.
- **e** (*float/numpy.ndarray*) – Eccentricity of ellipse.
- **tol** (*float*) – Numerical tolerance for solving Keplers equation in units of radians.
- **radians** (*bool*) – If true radians are used, else all angles are given in degrees

Returns True anomaly.

Return type numpy.ndarray or float

Uses:

- `_get_kepler_guess()`
- *`laguerre_solve_kepler()`*

`dpt_tools.`**`mean2true`**(*M*, *e*, *tol=1e-12*, *radians=True*)

Transforms mean anomaly to true anomaly.

Uses:

- *`mean2eccentric()`*
- *`eccentric2true()`*

Parameters

- **`M`** (*`float/numpy.ndarray`*) – Mean anomaly.
- **`e`** (*`float/numpy.ndarray`*) – Eccentricity of ellipse.
- **`tol`** (*`float`*) – Numerical tolerance for solving Keplers equation in units of radians.
- **`radians`** (*`bool`*) – If true radians are used, else all angles are given in degrees

Returns True anomaly.

Return type numpy.ndarray or float

`dpt_tools.`**`mjd2npdt`**(*mjd*)

Converts a modified Julian date to a numpy datetime64 value (UTC)

`dpt_tools.`**`mjd_to_j2000`**(*mjd_tt*)

Convert from Modified Julian Date to days past J2000.

Parameters **`mjd_tt`** (*`float/numpy.ndarray`*) – MJD in TT

Returns Days past J2000

Return type float/numpy.ndarray

`dpt_tools.`**`mjd_to_jd`**(*mjd*)

Convert Modified Julian Date (relative 0h Nov 17, 1858) to Julian Date (relative 12h Jan 1, 4713 BC)

`dpt_tools.`**`npdt2date`**(*dt*)

Converts a numpy datetime64 value to a date tuple

Parameters **`dt`** (*`numpy.datetime64`*) – Date and time (UTC) in numpy datetime64 format

Returns tuple (year, month, day, hours, minutes, seconds, microsecond) all except usec are integer

`dpt_tools.`**`npdt2mjd`**(*dt*)

Converts a numpy datetime64 value (UTC) to a modified Julian date

`dpt_tools.`**`orbit3D`**(*states*, *ax=None*)

Create a 3D plot of a set of states.

`dpt_tools.`**`orbital_period`**(*a*, *mu*)

Calculates the orbital period of an Keplerian orbit $v = 2\pi\sqrt{\frac{a^3}{\mu}}$.

Parameters

- **a** (*float/numpy.ndarray*) – Semi-major axis of ellipse.
- **mu** (*float*) – Standard gravitation parameter $\mu = G(m_1 + m_2)$ of the orbit.

Returns Orbital period.

`dpt_tools.`**`orbital_speed`**(*r*, *a*, *mu*)

Calculates the orbital speed at a given radius for an Keplerian orbit $v = \sqrt{\mu \left(\frac{2}{r} - \frac{1}{a}\right)}$.

Parameters

- **r** (*float/numpy.ndarray*) – Radius from the pericenter.
- **a** (*float/numpy.ndarray*) – Semi-major axis of ellipse.
- **mu** (*float*) – Standard gravitation parameter $\mu = G(m_1 + m_2)$ of the orbit.

Returns Orbital speed.

`dpt_tools.`**`orbits`**(*o*, ***options*)

This function creates several scatter plots of a set of orbital elements based on the different possible axis planar projections, calculates all possible permutations of plane intersections based on the number of columns

Parameters

- **o** (*numpy.ndarray*) – Rows are distinct orbits and columns are orbital elements in the order a, e, i, omega, Omega
- **options** – dictionary containing all the optional settings

Currently the options fields are:

marker [char] the marker type

size [int] the size of the marker

title [string] the title of the plot

title_font_size [int] the title font size

axis_labels [list of strings] labels for each column

tick_font_size [int] the axis tick font size

window [tuple/list] the size of the plot window in pixels (assuming dpi = 80)

save [string] will not display figure and will instead save it to this path

show [bool] if False will do draw() instead of show() allowing script to continue

tight_rect [list of 4 floats] configuration for the tight_layout function

Example:

```
import dpt_tools as dpt
import numpy as np
np.random.seed(19680221)

orbs = np.matrix([
    11  + 3  *np.random.randn(1000),
    0.5 + 0.2*np.random.randn(1000),
    60  + 10 *np.random.randn(1000),
    120 + 5  *np.random.randn(1000),
    33  + 2  *np.random.randn(1000),
]).T
```

(continues on next page)

(continued from previous page)

```
dpt.orbits(orbs,
    title = "My orbital element distribution",
    size = 10,
)
```

`dpt_tools.`**`plot_orbit_convention`**(*get_orbit*, *res=100*)

Plots the orbit convention used by arbitrary function/program

Parameters **`get_orbit`** (*`function`*) – A function pointer that takes an Kepler state as input and returns a state vector.

The arguments to `get_orbit` should be:

- *numpy.ndarray* of Kepler elements.
- Use degrees for angles.
- Kepler elements are a, e, i, ω, Ω, ν
- See *`kep2cart()`*

Example:

```
import dpt_tools as dpt
import space_object as so

def get_orbit(o):
    obj=so.space_object(
        a=o[0],
        e=o[1],
        i=o[2],
        raan=o[4],
        aop=o[3],
        mu0=o[5],
        C_D=2.3,
        A=1.0,
        m=1.0
    )
    return obj.get_state([0.0])

dpt.plot_orbit_convention(get_orbit)
```

`dpt_tools.`**`plot_ref_orbit`**(*get_orbit*, *res=100*, *orb_init=array([1.0e+07, 2.0e-01, 7.0e+01, 1.2e+02, 3.5e+01])*)

Plots a specific reference orbit.

Parameters **`get_orbit`** (*`function`*) – A function pointer that takes an Kepler state as input and returns a state vector.

The arguments to `get_orbit` should be:

- *numpy.ndarray* of Kepler elements.
- Use degrees for angles.
- Kepler elements are a, e, i, ω, Ω, ν
- See *`kep2cart()`*

Example:

```
import dpt_tools as dpt
import space_object as so

def get_orbit(o):
    obj=so.space_object(
        a=o[0],
        e=o[1],
        i=o[2],
        raan=o[4],
        aop=o[3],
        mu0=o[5],
        C_D=2.3,
        A=1.0,
        m=1.0
    )
    return obj.get_state([0.0])

dpt.plot_ref_orbit(get_orbit)
```

`dpt_tools.`**`posterior`**(*post*, *variables*, ***options*)

This function creates several scatter plots of a set of orbital elements based on the different possible axis planar projections, calculates all possible permutations of plane intersections based on the number of columns

Parameters

- **post** (*numpy.ndarray*) – Rows are distinct variable samples and columns are variables in the order of `variables`
- **variables** (*list*) – Name of variables, used for axis names.
- **options** – dictionary containing all the optional settings

Currently the options fields are:

bins [int] the number of bins

colormap [str] Name of colormap to use.

title [string] the title of the plot

title_font_size [int] the title font size

axis_labels [list of strings] labels for each column

tick_font_size [int] the axis tick font size

window [tuple/list] the size of the plot window in pixels (assuming dpi = 80)

save [string] will not display figure and will instead save it to this path

show [bool] if False will do draw() instead of show() allowing script to continue

tight_rect [list of 4 floats] configuration for the tight_layout function

`dpt_tools.`**`rot_mat_x`**(*theta*, *dtype=<type 'float'>*)

Generates the 3D transformation matrix for rotation around X-axis.

Parameters

- **theta** (*float*) – Angle to rotate.
- **dtype** (*numpy.dtype*) – The data-type of the output matrix.

Returns Rotation matrix

Return type (3,3) numpy.ndarray

`dpt_tools.`**`rot_mat_y`**(*theta*, *dtype=<type 'float'>*)
Generates the 3D transformation matrix for rotation around Y-axis.

Parameters

- **`theta`** (*`float`*) – Angle to rotate.
- **`dtype`** (*`numpy.dtype`*) – The data-type of the output matrix.

Returns Rotation matrix

Return type (3,3) numpy.ndarray

`dpt_tools.`**`rot_mat_z`**(*theta*, *dtype=<type 'float'>*)
Generates the 3D transformation matrix for rotation around Z-axis.

Parameters

- **`theta`** (*`float`*) – Angle to rotate.
- **`dtype`** (*`numpy.dtype`*) – The data-type of the output matrix.

Returns Rotation matrix

Return type (3,3) numpy.ndarray

`dpt_tools.`**`scatter`**(*x*, *y*, ***options*)
This function creates a scatter plot with lots of nice pre-configured settings unless they are overridden

Currently the options fields are:

marker [char] The marker type

size [int] The size of the marker

alpha [float] The transparency of the points.

title [string] The title of the plot

title_font_size [int] The title font size

xlabel [string] The label for the x axis

ylabel [string] The label for the y axis

tick_font_size [int] The axis tick font size

window [tuple/list] The size of the plot window in pixels (assuming dpi = 80)

save [string] Will not display figure and will instead save it to this path

show [bool] If False will do draw() instead of show() allowing script to continue

plot [tuple] A tuple with the `(fig, ax)` objects from matplotlib. Then no new figure and axis will be created.

Parameters

- **`x`** (*`numpy.ndarray`*) – x-axis data vector.
- **`y`** (*`numpy.ndarray`*) – y-axis data vector.
- **`options`** – dictionary containing all the optional settings.

Example:

```
import dpt_tools as dpt
import numpy as np
np.random.seed(19680221)

x = 10*np.random.randn(100)
y = 10*np.random.randn(100)

dpt.scatter(x, y, {
    "title": "My first plot",
    } )
```

`dpt_tools.`**`sec = numpy.timedelta64(1000000000,'ns')`**
numpy.datetime64: Interval of 1 second

`dpt_tools.`**`tai2utc`**(*ytime*)
TAI to UTC conversion using Leapseconds data.

`dpt_tools.`**`true2eccentric`**(*nu*, *e*, *radians=True*)
Calculates the eccentric anomaly from the true anomaly.

Parameters

- **nu** (*float/numpy.ndarray*) – True anomaly.
- **e** (*float/numpy.ndarray*) – Eccentricity of ellipse.
- **radians** (*bool*) – If true radians are used, else all angles are given in degrees

Returns Eccentric anomaly.

Return type numpy.ndarray or float

`dpt_tools.`**`true2mean`**(*nu*, *e*, *radians=True*)
Transforms true anomaly to mean anomaly.

Uses:

- *true2eccentric()*
- *eccentric2mean()*

Parameters

- **nu** (*float/numpy.ndarray*) – True anomaly.
- **e** (*float/numpy.ndarray*) – Eccentricity of ellipse.
- **radians** (*bool*) – If true radians are used, else all angles are given in degrees

Returns Mean anomaly.

Return type numpy.ndarray or float

`dpt_tools.`**`unix_to_jd`**(*unix*)
Convert Unix time to JD UT1

Constant is due to 0h Jan 1, 1970 = 2440587.5 JD

Parameters **unix** (*float/numpy.ndarray*) – Unix time in seconds.

Returns Julian Date UT1

Return type float/numpy.ndarray

`dpt_tools.`**`utc2tai`**(*ytime*)

UTC to TAI conversion using Leapseconds data.

`dpt_tools.`**`yearday_to_monthday`**(*year_day*, *leap*)

Convert a day of the year to a month-day pair. Only takes first order leap-year into account. The day of the year is actually counted so that it starts from 1, so that 1.0 corresponds to January 1 00:00.

Parameters

- **`year_day`** (*`float`*) – Day of the year.
- **`leap`** (*`bool`*) – Indicates of the year is a leap year or not.

Returns tuple of (month, day)

Example:

```
>>> yearday_to_monthday(1.1, False)
(1.0, 1.1)
>>> yearday_to_monthday(31.1, False)
(1.0, 31.1)
>>> yearday_to_monthday(32.1, False)
(2.0, 1.1)
```

7.3.5 TLE_tools

Collection of useful functions for handling TLE's.

Links:

- AIAA 2006-6753
- python-skyfield

`TLE_tools.`**`TEME_to_ITRF`**(*TEME*, *jd_ut1*, *xp*, *yp*)

Convert TEME position and velocity into standard ITRS coordinates. This converts a position and velocity vector in the idiosyncratic True Equator Mean Equinox (TEME) frame of reference used by the SGP4 theory into vectors into the more standard ITRS frame of reference.

Reference: AIAA 2006-6753 Appendix C.

Original work Copyright (c) 2013-2018 Brandon Rhodes under the MIT license Modified work Copyright (c) 2019 Daniel Kastinen

Since TEME uses the instantaneous North pole and mean direction of the Vernal equinox, a simple GMST and polar motion transformation will move to ITRS.

# TODO: There is some ambiguity about if this is ITRS00 or something else? I dont know.

Parameters

- **`TEME`** (*`numpy.ndarray`*) – 6-D state vector in TEME frame given in SI-units.
- **`jd_ut1`** (*`float`*) – UT1 Julian date.
- **`xp`** (*`float`*) – Polar motion constant for rotation around x axis
- **`yp`** (*`float`*) – Polar motion constant for rotation around y axis

Returns ITRF 6-D state vector given in SI-units.

Return type numpy.ndarray

`TLE_tools.`**`TEME_to_TLE`**(*state*, *mjd0*, *kepler=False*, *tol=1e-06*, *tol_v=1e-07*)

Convert osculating orbital elements in TEME to mean elements used in two line element sets (TLE's).

Parameters

- **`kep`** (*`numpy.ndarray`*) – Osculating State (position and velocity) vector in km and km/s, TEME frame. If `kepler = True` then state is osculating orbital elements, in km and radians. Orbital elements are semi major axis (km), orbital eccentricity, orbital inclination (radians), right ascension of ascending node (radians), argument of perigee (radians), mean anomaly (radians)
- **`kepler`** (*`bool`*) – Indicates if input state is kepler elements or cartesian.
- **`mjd0`** (*`float`*) – Modified Julian date for state, important for SDP4 iteration.
- **`tol`** (*`float`*) – Wanted precision in position of mean element conversion in km.
- **`tol_v`** (*`float`*) – Wanted precision in velocity mean element conversion in km/s.

Returns mean elements of: semi major axis (km), orbital eccentricity, orbital inclination (radians), right ascension of ascending node (radians), argument of perigee (radians), mean anomaly (radians)

Return type numpy.ndarray

`TLE_tools.`**`TEME_to_TLE_OPTIM`**(*state*, *mjd0*, *kepler=False*, *tol=1e-06*, *tol_v=1e-07*, *method=None*)

Convert osculating orbital elements in TEME to mean elements used in two line element sets (TLE's).

Parameters

- **`kep`** (*`numpy.ndarray`*) – Osculating State (position and velocity) vector in km and km/s, TEME frame. If `kepler = True` then state is osculating orbital elements, in km and radians. Orbital elements are semi major axis (km), orbital eccentricity, orbital inclination (radians), right ascension of ascending node (radians), argument of perigee (radians), mean anomaly (radians)
- **`kepler`** (*`bool`*) – Indicates if input state is kepler elements or cartesian.
- **`mjd0`** (*`float`*) – Modified Julian date for state, important for SDP4 iteration.
- **`tol`** (*`float`*) – Wanted precision in position of mean element conversion in km.
- **`tol_v`** (*`float`*) – Wanted precision in velocity mean element conversion in km/s.
- **`method`** (*`str`*) – Forces use of SGP4 or SDP4 depending on string 'n' or 'd', if None method is automatically chosen based on orbital period.

Returns mean elements of: semi major axis (km), orbital eccentricity, orbital inclination (radians), right ascension of ascending node (radians), argument of perigee (radians), mean anomaly (radians)

Return type numpy.ndarray

`TLE_tools.`**`TLE_propagation_TEME`**(*line1*, *line2*, *jd_ut1*, *wgs='72'*)

Convert Two-line element to TEME coordinates at a specific Julian date.

Parameters

- **`line1`** (*`str`*) – TLE line 1
- **`line2`** (*`str`*) – TLE line 2
- **`jd_ut1`** (*`float/numpy.ndarray`*) – Julian Date UT1 to propagate TLE to.
- **`wgs`** (*`str`*) – The used WGS standard, options are `'72'` or `'84'`.

Returns (6,len(jd_ut1)) numpy.ndarray of Cartesian states [SI units]

`TLE_tools.`**`TLE_to_TEME`**(*line1*, *line2*, *wgs='72'*)

Convert Two-line element to TEME coordinates and a Julian date epoch.

Here it is assumed that the TEME frame uses: The Cartesian coordinates produced by the SGP4/SDP4 model have their z axis aligned with the true (instantaneous) North pole and the x axis aligned with the mean direction of the vernal equinox (accounting for precession but not nutation). This actually makes sense since the observations are collected from a network of sensors fixed to the earth's surface (and referenced to the true equator) but the position of the earth in inertial space (relative to the vernal equinox) must be estimated.

Parameters

- **`line1`** (*`str`*) – TLE line 1
- **`line2`** (*`str`*) – TLE line 2
- **`wgs`** (*`str`*) – The used WGS standard, options are `'72'` or `'84'`.

Returns tuple of (6-D numpy.ndarray Cartesian state [SI units], epoch in Julian Date UT1)

`TLE_tools.`**`get_DUT`**(*jd_ut1*)

Get the Difference UT between UT1 and UTC, $DUT1 = UT1 - UTC$. This function interpolates between data given by IERS.

Parameters **`jd_ut1`** (*`float/numpy.ndarray`*) – Input Julian date in UT1.

Returns DUT

Return type numpy.ndarray

`TLE_tools.`**`get_IERS_EOP`**(*fname='/home/danielk/IRF/IRF_GITLAB/SORTSpp/data/eopc04_IAU2000.62-now'*)

Loads the IERS EOP data into memory.

Note: Column descriptions are hard-coded in the function and my change if standard IERS format is changed.

Parameters **`fname`** (*`str`*) – path to input IERS data file.

Returns tuple of (numpy.ndarray, list of column descriptions)

`TLE_tools.`**`get_Polar_Motion`**(*jd_ut1*)

Get the Polar motion coefficients x_p and y_p used in EOP. This function interpolates between data given by IERS.

Parameters **`jd_ut1`** (*`float/numpy.ndarray`*) – Input Julian date in UT1.

Returns x_p as column 0 and y_p as column 1

Return type numpy.ndarray

`TLE_tools.`**`theta_GMST1982`**(*jd_ut1*)

Return the angle of Greenwich Mean Standard Time 1982 given the JD. This angle defines the difference between the idiosyncratic True Equator Mean Equinox (TEME) frame of reference used by SGP4 and the more standard Pseudo Earth Fixed (PEF) frame of reference.

Reference: AIAA 2006-6753 Appendix C.

Original work Copyright (c) 2013-2018 Brandon Rhodes under the MIT license Modified work Copyright (c) 2019 Daniel Kastinen

Parameters **`jd_ut1`** (*`float`*) – UT1 Julian date.

Returns tuple of (Earth rotation [rad], Earth angular velocity [rad/day])

`TLE_tools.`**`tle_bstar`**(*line1*)

Extracts the BSTAR drag coefficient as a float from the first line of a TLE.

`TLE_tools.`**`tle_date`**(*line1*)

`TLE_tools.`**`tle_id`**(*line1*)
Extracts the Satellite number from the first line of a TLE.

`TLE_tools.`**`tle_jd`**(*line1*)

`TLE_tools.`**`tle_npdt`**(*line1*)

7.3.6 population_filter

Investigate what fraction of objects can be detected with a radar system.

At least two somewhat straightforward definitions can be made. We'll use D_{24h} due to it being very simple to evaluate numerically.

D_{24h}, Can the object be detected in 24 hours of observations. - Makes more sense in terms of objects that can be maintained in a catalog.

It an object is observed less than once a day, then it probably cannot be maintained in a catalog very well.

D_{∞}, Can the object be detected in infinite number of days. - This could be analytically determined, by using info on eccentricity, apogee, inclination,

and object size. Not sure why this would be more useful than D_{24h}

`population_filter.`**`filter_objects`**(*radar*, *m*, *ofname='det_filter.h5'*, *prop_time=24.0*)
Propagate for a number of hours, and determine if radar system can detect the object.

Parameters

- **`radar`** (`RadarSystem`) – The radar configuration used for the detectability filtering.
- **`m`** (`Population`) – Input population to filter.
- **`ofname`** (*`str`*) – Output file name. If `None` then the results are returned by the function instead of written to file.
- **`prop_time`** (*`float`*) – Time to propagate when filtering.

# TODO: Shouldent this function use the radar snr treshold to and not only enr treshhold of antennas?

`population_filter.`**`get_passes_simple`**(*o*, *radar*, *t0*, *t1*, *max_dpos=100000.0*, *debug=False*, *sanity_check=False*)
Follow object and find peak SNR. Assume that this occurs for minimum zenith angle of each TX.

Parameters

- **`o`** (`SpaceObject`) – The object in space to be followed.
- **`radar`** (`RadarSystem`) – The radar system used for tracking.
- **`t0`** (*`float`*) – Start time for tracking
- **`t1`** (*`float`*) – End time for tracking
- **`max_dpos`** (*`float`*) – Maximum separation in m between orbital evaluation points, used to calculate time-step size by approximating orbits as circles
- **`debug`** (*`bool`*) – Verbose output
- **`sanity_check`** (*`bool`*) – Even more verbose output

Returns Tuple of (Peak SNR for tracking, Number of receivers that can observe, time of best detection)

7.3.7 ray_trace

7.3.8 simulate_scan

Simulate discovery observations with user-defined scan pattern.

# TODO: Describe module usage

`simulate_scan.`**`get_detections`**(*obj*, *radar*, *t0*, *t1*, *max_dpos=10000.0*, *logger=None*, *pass_dt=None*)

Find all detections of a object by input radar between two times relative the object Epoch.

Parameters

- **`obj`** (`SpaceObject`) – Space object to find detections of.
- **`radar`** (`RadarSystem`) – Radar system that scans for the object.
- **`t0`** (*`float`*) – Start time for scan relative space object epoch.
- **`t1`** (*`float`*) – End time for scan relative space object epoch.
- **`max_dpos`** (*`float`*) – Maximum separation between evaluation points in meters for finding the pass interval.
- **`logger`** (*`Logger`*) – Logger object for logging the execution of the function.
- **`pass_dt`** (*`float`*) – The time step used when evaluating pass. Default is the scan-minimum dwell time but can be forces to a setting by this variable.

Returns Detections data structure in form of a list of dictionaries, see description below.

Return type list

Return data:

List of same length as radar system TX antennas. Each entry in the list is a dictionary with the following items:

- t0: List of pass start times. Length is equal the number of detection but unique times are equal to the number of passes..
- t1: List of pass end times, i.e. when the space object passes below the FOV. Same list configuration as “t0”
- snr: List of lists of SNR’s for each TX-RX pair for each detection. I.e. the top list length is equal the number of detections and the elements are lists of length equal to the number of TX-RX pairs.
- tm: List of times corresponding to each detection, same length as “snr” item.
- range: Same structure as the “snr” item but with ranges between the space object and each RX antenna. Unit is meters.
- range_rate: Same structure as the “range” item but with range-rates, i.e. line of sight velocity. Unit is meters per second.
- tx_gain: List of gains from the TX antenna for the detection, length of list is equal the number of detections.
- rx_gain: List of lists in the same structure as the “snr” item but with receiver gains instead of signal to noise ratios.
- on_axis_angle: List of angles between the space object and the pointing direction for each detection, length of list is equal to the number of detections.

`simulate_scan.`**`plot_scan_for_object`**(*obj*, *radar*, *t0*, *t1*, *plot_full_scan=False*)

`simulate_scan.`**`pp_det`**(*det_times*)

Function to pretty print detection times list returned by the `simulate_scan.get_iods()` function.

Parameters **`det_times`** (*`list`*) – List of dictionaries generated by get_iods.

7.3.9 simulate_tracking

Functions for simulating the tracking of an object in space.

Usage examples:

```
import radar_library as rl
from propagator_sgp4 import PropagatorSGP4

radar=rl.eiscat_3d(beam='gauss')

o=so.SpaceObject(
    a=7000,
    e=0.0,
    i=72,
    raan=0,
    aop=0,
    mu0=0,
    C_D=2.3,
    A=1.0,
    m=1.0,
    d=0.1,
    propagator = PropagatorSGP4,
    propagator_options = {
        'polar_motion': False,
    },
)

_t0 = time.time()
passes = get_passes(o, radar, 0, 18.*3600., max_dpos=1e3)
_t1=time.time()
print("wall clock time %1.2f"%(_t1-_t0))

print_passes(passes)

p_id = 1

t0, t1 = passes['t'][0][p_id]
t = n.linspace(t0, t1, num=1000)

scan_snr = get_scan_snr(t, o, radar)
track_snr = get_track_snr(t, o, radar)

fig = plt.figure(figsize=(15,15))
ax = fig.add_subplot(211)
ax = plot_snr(t, track_snr, radar, ax=ax)
ax.set_title('SNR when tracking')

ax = fig.add_subplot(212)
ax = plot_snr(t, scan_snr, radar, ax=ax)
ax.set_title('SNR when scanning')
```

(continues on next page)

(continued from previous page)

```
ts, angs = get_angles(passes, o, radar)
ax = plot_angles(ts, angs)

plt.show()
```

simulate_tracking.**find_linspace_num**(*t0*, *t1*, *a*, *e*, *max_dpos=1000.0*)

Find the number of linearly spaced temporal positions which are sufficient to achieve a maximum spatial separation. Assume elliptic orbit and use the velocity at periapsis, does not take perturbation patterns into account.

Parameters

- **t0** (*float*) – Start time in seconds
- **t1** (*float*) – End time in seconds
- **a** (*float*) – Semi-major axis in meters
- **max_dpos** (*float*) – Maximum separation between evaluation points in meters.

Returns Number of points needed

Return type int

simulate_tracking.**find_pass_interval**(*t*, *o*, *radar*)

Find a pass inside the FOV of a radar given a series of times for a space object.

Parameters

- **t** (*numpy.ndarray*) – Linear vector of times to use as a base to find pass in seconds relative space object epoch.
- **o** (SpaceObject) – Space object to find pass interval for.
- **radar** (RadarSystem) – Radar system that defines the FOV.

Returns Tuple of (passes, passes_id, idx_v, postx_v, posrx_v), description below.

Return data:

- passes: Three layers of lists where first layer is a list corresponding to every RX antenna of the radar system. Second layer is the a entry in the list for every pass. Last layer of lists is a list of two elements where the first is the time in seconds when object enters the FOV and second is the time in seconds when the object leaves the FOV.
- passes_id: Same structure as the passes data but with the time indices's instead of the actual times.
- idx_v: List of arrays of indices's of input time vector where the space object is inside the TX FOV, length of list is equal to the number of TX stations.
- postx_v: list of arrays containing the position of the space object relative the TX stations, length of list is equal to number of TX stations and the array is the length of the input time vector.
- posrx_v: list of arrays containing the position of the space object relative the RX stations, length of list is equal to number of RX stations and the array is the length of the input time vector.

simulate_tracking.**get_angles**(*passes*, *o*, *radar*, *dt=0.1*)

Takes the passes structure that is output from *simulate_tracking.get_passes()*, the space object and the radar system and calculates the zenith angle for all passes.

Parameters

- **passes** (*dict*) – Output from *simulate_tracking.get_passes()* that contains information about passes of an space object.

- **o** (SpaceObject) – Space object that made the passes.
- **radar** (RadarSystem) – Radar system that defines the FOV.
- **dt** (*float*) – Time step for angle evaluation.

Returns Tuple of list of lists of times and list of lists of angles corresponding to each pass.

`simulate_tracking.`**`get_passes`**(*o*, *radar*, *t0*, *t1*, *max_dpos=1000.0*, *logger=None*, *plot=False*, *t_samp=None*)

Follow object and determine possible maintenance track window. I.e. get all passes of the object inside the radar system FOV.

Parameters

- **o** (SpaceObject) – Space object to find passes for.
- **radar** (RadarSystem) – Radar system that defines the FOV.
- **t0** (*float*) – Start time for passes search in seconds relative space object epoch.
- **t1** (*float*) – Stop time for passes search in seconds relative space object epoch.
- **max_dpos** (*float*) – Maximum separation in km between orbital evaluation points.
- **logger** (*Logger*) – Logger object for logging the execution of the function.
- **t_samp** (*float*) – If not None, overrides the "max_dpos" variable and fixes a time-sampling.

Returns Dictionary containing information about all passes of the space object inside the radar system FOV.

Return type dict

Output dictionary:

- t: Three layers of lists where first layer is a list corresponding to every RX antenna of the radar system. Second layer is the a entry in the list for every pass. Last layer of lists is a list of two elements where the first is the time in seconds when object enters the FOV and second is the time in seconds when the object leaves the FOV. I.e. `pass_start_time = passes["t"][tx_index][pass_index][0]` and `pass_end_time = passes["t"][tx_index][pass_index][1]`.
- snr: This structure has the same format as the "t" item but with an extra layer of lists of receivers before the bottom. Then instead of the bottom layer of lists being start and stop times it records the peak SNR at the first item and the time of that peak SNR in the second item. I.e. `pass_peak_snr = passes["snr"][tx_index][pass_index][rx_index][0]` and `pass_peak_snr_time = passes["snr"][tx_index][pass_index][rx_index][1]`.

`simulate_tracking.`**`get_scan_snr`**(*t*, *o*, *radar*)

Takes a series of times, a space object and a radar system and calculates the SNR for that space object given the scan pattern of the radar over the given times.

Parameters

- **t** (*numpy.ndarray*) – Times in seconds relative space object epoch over witch SNR should be evaluated.
- **o** (SpaceObject) – Space object to be measured.
- **radar** (RadarSystem) – Radar system that performs the measurement.

Returns List of lists of numpy.ndarray's corresponding to TX antenna index, RX antenna index and SNR-array in that order of list depth.

`simulate_tracking.`**`get_track_snr`**`(`*t*, *o*, *radar*`)`

Takes a series of times, a space object and a radar system and calculates the SNR for that space object measured by that radar over the given times.

Parameters

- **`t`** (*`numpy.ndarray`*) – Times in seconds relative space object epoch over witch SNR should be evaluated.
- **`o`** (`SpaceObject`) – Space object to be measured.
- **`radar`** (`RadarSystem`) – Radar system that performs the measurement.

Returns List of lists of numpy.ndarray's corresponding to TX antenna index, RX antenna index and SNR-array in that order of list depth.

`simulate_tracking.`**`plot_angles`**`(`*ts*, *angs*, *ax=None*`)`

Plot the angles data returned by the *`simulate_tracking.get_angles()`* function.

Parameters

- **`ts`** (*`list`*) – List of times for each pass that the angles were evaluated over.
- **`angs`** (*`list`*) – List of angles for each pass.
- **`ax`** – matplotlib axis to plot the SNR's on. If not given, create new figure and axis.

Returns The matplotlib axis object

`simulate_tracking.`**`plot_snr`**`(`*t*, *all_snrs*, *radar*, *ax=None*`)`

Plots the SNR's structure (list of lists of numpy.ndarray's) returned by *`simulate_tracking.get_track_snr()`* and *`simulate_tracking.get_scan_snr()`*.

Parameters

- **`t`** (*`numpy.ndarray`*) – Times corresponding to the evaluated SNR's.
- **`all_snrs`** – List structure returned by *`simulate_tracking.get_track_snr()`* and *`simulate_tracking.get_scan_snr()`*.
- **`radar`** (`RadarSystem`) – Radar system that measured the SNR's.
- **`ax`** – matplotlib axis to plot the SNR's on. If not given, create new figure and axis.

Returns The matplotlib axis object

`simulate_tracking.`**`print_passes`**`(`*det_times*`)`

Function to pretty print detection times list returned by the *`simulate_tracking.get_passes()`* function.

Parameters **`det_times`** (*`list`*) – List of dictionaries generated by get_passes.

7.3.10 simulate_tracklet

Given scheduled observations of an object simulate the generated tracklet-data.

# TODO: Rewrite with new functionality # TODO: Do not re-do the entire "observation simulation" as there is already modules that to this better. Instead just take a time-series in and create the tracklet, let other code worry about if it is physically correct or not.

Simulate an EISCAT 3D tracking experiment using MASTER model objects

- Follow object from horizon to horizon
- Output estimated range and range-rate errors

`simulate_tracklet.`**`create_tracklet`**(*o*, *radar*, *t_obs*, *hdf5_out=True*, *ccsds_out=True*, *dname='./tracklets'*, *noise=False*, *dx=10.0*, *dv=10.0*, *dt=0.01*)

Simulate tracks of objects.

ionospheric limit is a lower limit on precision after ionospheric corrections

`simulate_tracklet.`**`iono_errfun = <scipy.interpolate.interpolate.interp1d object>`**

func: Model of the ionospheric error function. See module `debris`.

`simulate_tracklet.`**`write_ccsds`**(*o*, *meas*, *tx*, *rx*, *fname*, *idx=[]*)

Write tracklet in ccsds file format

`simulate_tracklet.`**`write_tracklets`**(*o*, *meas*, *radar*, *dname*, *hdf5_out=True*, *ccsds_out=True*, *dt=3600*)

Write a tracklet file.

# TODO: write as chunks of one pass per file

7.3.11 simulate_scaning_snr

Functions for single object propagation and SNR examination.

`simulate_scaning_snr.`**`simulate_full_scaning_snr_curve`**(*radar*, *o*, *det_times*, *tresh*, *rem_t*, *obs_par*, *groups*, *IPP_scale=1.0*, *plot=True*, *verbose=True*)

7.3.12 logging_setup

Sets up a logging framework that can be imported and used anywhere.

`logging_setup.`**`add_logging_level`**(*num*, *name*)

`logging_setup.`**`class_log_call`**(*form*)

`logging_setup.`**`construct_formatted_format`**(*form*, *args_len*, *kwargs*)

This takes a special formatted string, extracts the two possible keys, and based on what was passed as key-word arguments and what was passed as indexed arguemnts, chooses the correct format. If a option has a default value it will indicate this and not report a value.

Returns a correctly formatted string for the input arguemts of the function.

`logging_setup.`**`extract_format_keys`**(*form*)

This function looks for our special formatting of indicating index of argument and name of argument

Returns a list of tuples where each tuple is the key in index format and in named format

`logging_setup.`**`extract_format_strings`**(*form*)

Extracts the formatting string inside curly braces by returning the index positions

For example:

string = "{2|number_of_sheep} sheep {0|has} run away" form_v = extract_format_strings(string) print(form_v) for x,y in form_v:

print(string[x:y])

gives: 2|number_of_sheep 0|has

`logging_setup.`**`log_call`**(*form*, *logger*) `logging_setup.`**`logg_time_record`**(*exec_t*, *logger*)

Saves time record to log at info level

`logging_setup.`**`record_time_diff`**(*name*)

Records a time difference since last call

This function modifies a global variable 'exec_times' in this module! This is especcialy useful for timing contents of loops

example: .. code:python

record_time_diff('loop_start') for i in range(large_number):

function_one(*args) record_time_diff('function_one')

function_two(*args) record_time_diff('function_two')

`logging_setup.`**`setup_logging`**(*name='SORTS++'*, *root="*, *file_level=20*, *term_level=20*, *parallel=0*, *logfile=True*)

Returns a logger object to be used in simulations

Formats to output both to terminal and a log file

7.3.13 orbit_accuracy

Linearized error determination for orbital elements.

These error are calculated as a function of:

- Mean track length (track_length) seconds
- Number of tracklets: (n_tracklets), int
- Measurement spacing: (m_spacing), seconds

`orbit_accuracy.`**`create_measurements`**(*o*, *radar*, *t0=0*, *track_length=1000.0*, *n_tracklets=2*, *n_meas=10*, *debug=False*, *max_time=259200.0*)

`orbit_accuracy.`**`error_sweep`**(*o*, *r*, *n_meas=10*)

`orbit_accuracy.`**`error_sweep_n`**(*o*, *r*)

`orbit_accuracy.`**`error_sweep_n_meas`**(*o*, *r*)

`orbit_accuracy.`**`error_sweep_n_meas_constn`**(*o*, *r*)

`orbit_accuracy.`**`error_sweep_time`**(*o*, *r*)

`orbit_accuracy.`**`error_sweep_track_length`**(*o*, *r*) `orbit_accuracy.`**`kep_cov2cart_cov`**(*o*, *Sigma_kep*, *t0s=array([0., 146.93877551, 293.87755102, 440.81632653, 587.75510204, 734.69387755, 881.63265306, 1028.57142857, 1175.51020408, 1322.44897959, 1469.3877551, 1616.32653061, 1763.26530612, 1910.20408163, 2057.14285714, 2204.08163265, 2351.02040816, 2497.95918367, 2644.89795918, 2791.83673469, 2938.7755102, 3085.71428571, 3232.65306122, 3379.59183673, 3526.53061224, 3673.46938776, 3820.40816327, 3967.34693878, 4114.28571429, 4261.2244898, 4408.16326531, 4555.10204082, 4702.04081633, 4848.97959184, 4995.91836735, 5142.85714286, 5289.79591837, 5436.73469388, 5583.67346939, 5730.6122449, 5877.55102041, 6024.48979592, 6171.42857143, 6318.36734694, 6465.30612245, 6612.24489796, 6759.18367347, 6906.12244898, 7053.06122449, 7200.])*)

`orbit_accuracy.`**`linearized_errors`**(*o*, *radar*, *tracklets*, *plot=True*, *debug=False*, *t0s=array([0., 146.93877551, 293.87755102, 440.81632653, 587.75510204, 734.69387755, 881.63265306, 1028.57142857, 1175.51020408, 1322.44897959, 1469.3877551, 1616.32653061, 1763.26530612, 1910.20408163, 2057.14285714, 2204.08163265, 2351.02040816, 2497.95918367, 2644.89795918, 2791.83673469, 2938.7755102, 3085.71428571, 3232.65306122, 3379.59183673, 3526.53061224, 3673.46938776, 3820.40816327, 3967.34693878, 4114.28571429, 4261.2244898, 4408.16326531, 4555.10204082, 4702.04081633, 4848.97959184, 4995.91836735, 5142.85714286, 5289.79591837, 5436.73469388, 5583.67346939, 5730.6122449, 5877.55102041, 6024.48979592, 6171.42857143, 6318.36734694, 6465.30612245, 6612.24489796, 6759.18367347, 6906.12244898, 7053.06122449, 7200.])*, *time_vector=False*)

`orbit_accuracy.`**`plot_measurements`**(*o*, *r*, *tracklets*)

7.3.14 orbital_estimation

Estimates a space objects state vector from a set of ranges and range-rates.

`orbital_estimation.`**`estimate_state`**(*r_meas*, *rr_meas*, *p_rx*)

`orbital_estimation.`**`meas2pos`**(*m*, *p_rx*, *dt=0.01*)

`orbital_estimation.`**`meas2vel`**(*p*, *m*, *p_rx*, *dt=0.1*)

`orbital_estimation.`**`state_estimation`**(*tracklet_list*, *verbose=False*)

`orbital_estimation.`**`state_estimation_v2`**(*tracklet_folder*, *track_id=-1*, *verbose=False*)

7.3.15 plothelp

Functions for making plots quicker.

`plothelp.`**`draw_earth`**(*ax*)

`plothelp.`**`draw_earth_grid`**(*ax*, *num_lat=25*, *num_lon=50*, *alpha=0.1*, *res=100*, *color='black'*)

`plothelp.`**`draw_radar`**(*ax*, *lat*, *lon*, *name='radar'*, *color='black'*)

7.3.16 lgeom

Collection of simple geometric functions.

`lgeom.`**`dist`**(*a0*, *a1*, *b0*, *b1*, *clampAll=False*, *clampA0=False*, *clampA1=False*, *clampB0=False*, *clampB1=False*)

Given two lines defined by numpy.array pairs (a0,a1,b0,b1) Return distance, the two closest points, and their average

7.3.17 correlator

Correlate measurement time series with a population of objects to find the best match.

Currently only works for Mono-static measurements.

# TODO: Assume a uniform prior distribution over population index, posterior distribution is the probability of what object generated the data. Probability comes from measurement covariance.

`correlator.`**`correlate`**(*data*, *station*, *population*, *metric*, *n_closest=1*, *out_file=None*, *verbose=False*, *MPI_on=False*)

Given a mono-static measurement of ranges and rage-rates, a radar model and a population: correlate measurements with population.

Parameters

- **`data`** (*`dict`*) – Dictionary that contains measurement data. Contents are described below.
- **`station`** (`AntennaRX`) – Model of receiver station that performed the measurement.
- **`population`** (`Population`) – Population to correlate against.
- **`metric`** (*`function`*) – Metric used to correlate measurement and simulation of population.
- **`n_closest`** (*`int`*) – Number of closest matches to output.
- **`out_file`** (*`str`*) – If not `None`, save the output data to this path.
- **`MPI_on`** (*`bool`*) – If True use internal parallelization with MPI to calculate correlation. Turn to False to externally parallelize with MPI.

Measurement data:

The file must be a dictionary that contains three data-sets:

- 't': Times in unix-seconds
- 'r': Ranges in meters
- 'v': Range-rates in meters per second

They should all be numpy vectors of equal length.

`correlator.`**`plot_correlation`**(*dat*, *cdat*)

Plot the correlation between the measurement and simulated population object.

`correlator.`**`residual_distribution_metric`**(*t*, *r*, *v*, *r_ref*, *v_ref*)

Using the simulated and the measured ranges and rage-rates calculate a de-correlation metric.

Parameters

- **t** (*numpy.ndarray*) – Times in seconds corresponding to measurement and simulation data.
- **r** (*numpy.ndarray*) – Measured ranges in meters
- **v** (*numpy.ndarray*) – Measured rage-rates in meters per second
- **r_ref** (*numpy.ndarray*) – Simulated ranges in meters
- **v_ref** (*numpy.ndarray*) – Simulated rage-rates in meters per second

Returns Metric value, smaller values indicate better match.

Return type float

7.4 Instance libraries

radar_library	A collection of *radar_config.RadarSystem* instances, such as EISCAT 3D and EISCAT UHF.
population_library	Library of population instances.
antenna_library	A collection of functions that return common instances of the *BeamPattern* class.
radar_scan_library	A collection of *radar_scans.RadarScan* instances, such as fence scans or ionospheric grids.
scheduler_library	Collection of classes and functions related to constructing a radar system scheduler.
rewardf_library	

7.4.1 radar_library

A collection of *radar_config.RadarSystem* instances, such as EISCAT 3D and EISCAT UHF.

radar_library.**eiscat_3d**(*beam='interp'*, *stage=1*)

The EISCAT_3D system.

For more information see:

- EISCAT
- EISCAT 3D

Parameters

- **beam** (*str*) – Decides what initial antenna radiation-model to use.
- **stage** (*int*) – The stage of development of EISCAT 3D.

EISCAT 3D Stages:

- Stage 1: As of writing it is assumed to have all of the antennas in place but only transmitters on half of the antennas in a dense core ,i.e. TX will have 42 dB peak gain while RX still has 45 dB peak gain. 3 Sites will exist, one is a TX and RX, the other 2 RX sites.
- Stage 2: Both TX and RX sites will have 45 dB peak gain.
- Stage 3: (NOT IMPLEMENTED HERE) 2 additional RX sites will be added.

Beam options:

- gauss: Gaussian tapered beam model *antenna_library.planar_beam()*.
- interp: Interpolated array pattern.
- array: Ideal summation of all antennas in the array *antenna_library.e3d_array_beam_stage1()* and *antenna_library.e3d_array_beam()*.

# TODO: Geographical location measured with? Probably WGS84.

radar_library.**eiscat_3d_module**(*beam='gauss'*)

A single EISCAT 3D module with 100 antennas

Parameters **beam** (*str*) – Decides what initial antenna radiation-model to use.

Beam options:

- gauss: Gaussian tapered beam model *antenna_library.planar_beam()*.
- array: Ideal summation of all antennas in the array *antenna_library.e3d_array_beam_stage1()* and *antenna_library.e3d_array_beam()*.

Based on *radar_library.eiscat_3d()* but with modified beam pattern.

radar_library.**eiscat_svalbard**()

The steerable antenna of the ESR radar, default settings for the Space Debris radar mode.

radar_library.**eiscat_uhf**()

radar_library.**tromso_space_radar**(*freq=1200000000.0*)

7.4.2 population_library

Library of population instances.

population_library.**Microsat_R_debris**(*mjd*, *num*, *radii_range*, *mass_range*, *propagator*, *propagator_options*)

population_library.**Microsat_R_debris_TLE**(*mjd=None*)

population_library.**NESCv9_mini_moons**(*albedo*, *propagate_SNR=None*, *radar=None*, *truncate=None*)

population_library.**filtered_master_catalog_factor**(*radar*, *input_file='./master/celn_20090501_00.sim'*, *detectability_file=None*, *mjd0=54952.0*, *treshhold=0.01*, *min_inc=50*, *seed=65487945*, *prop_time=24.0*, *propagator=<class 'propagator_sgp4.PropagatorSGP4'>*, *propagator_options={}*)

Returns a random realization of the master population specified by the input file/population but filtered according detectability from a *radar_config.RadarSystem*.

Filter results are saved in the same folder as the `input_file` variable specifies the Master catalog file location.

Parameters

- **radar** (RadarSystem) – The radar configuration used for the detectability filtering.
- **input_file** (*str*) – Path to the input MASTER file. Is not used if `master_base` is given.

- **`detectability_file`** (*`str`*) – Path to the output-definition file so that a cached file can be used to load population instead of re-calculating every time.
- **`sort`** (*`bool`*) – If `True` sort according to diameters in ascending order.
- **`treshhold`** (*`float`*) – Diameter limit in meters below witch sampling objects are not included. Can be `None` to skip filtering.
- **`min_inc`** (*`float`*) – Inclination limit in degrees below witch sampling objects are not included. Can be `None` to skip filtering.
- **`seed`** (*`int`*) – Random number generator seed given to `numpy.random.seed` to allow for consisted generation of a random realization of the population. If seed is `None` a random seed from high-entropy data is used.
- **`prop_time`** (*`float`*) – Propagation time used to check if object is detectable.
- **`propagator`** (`PropagatorBase`) – Propagator class pointer used for *`space_object.SpaceObject`*.
- **`propagator_options`** (*`dict`*) – Propagator initialization keyword arguments.

Returns Filtered master population

Return type *population.Population*

`population_library.`**`master_catalog`**(*input_file='./master/celn_20090501_00.sim', mjd0=54952.0, sort=True, propagator=<class 'propagator_sgp4.PropagatorSGP4'>, propagator_options={}*)

Return the master catalog specified in the input file as a population instance. The catalog only contains the master sampling objects and not an actual realization of the population using the factor.

The format of the input master files is:

0. ID
1. Factor
2. Mass [kg]
3. Diameter [m]
4. m/A [kg/m2]
5. a [km]
6. e
7. i [deg]
8. RAAN [deg]
9. AoP [deg]
10. M [deg]

Parameters

- **`input_file`** (*`str`*) – Path to the input MASTER file.
- **`sort`** (*`bool`*) – If `True` sort according to diameters in descending order.
- **`mjd0`** (*`float`*) – The epoch of the catalog file in Modified Julian Days.
- **`propagator`** (`PropagatorBase`) – Propagator class pointer used for *`space_object.SpaceObject`*.
- **`propagator_options`** (*`dict`*) – Propagator initialization keyword arguments.

Returns Master catalog

Return type *population.Population*

`population_library.`**`master_catalog_factor`**(*input_file='./master/celn_20090501_00.sim', mjd0=54952.0, master_base=None, treshhold=0.01, seed=None, propagator=<class 'propagator_sgp4.PropagatorSGP4'>, propagator_options={}*)

Returns a random realization of the master population specified by the input file/population. In other words, each sampling object in the catalog is sampled a "factor" number of times with random mean anomalies to create the population.

Parameters

- **`input_file`** (*`str`*) – Path to the input MASTER file. Is not used if `master_base` is given.
- **`mjd0`** (*`float`*) – The epoch of the catalog file in Modified Julian Days. Is not used if `master_base` is given.
- **`master_base`** (`population.Population`) – A master catalog consisting only of sampling objects. This catalog will be modified and the pointer to it returned.
- **`sort`** (*`bool`*) – If `True` sort according to diameters in ascending order.
- **`treshhold`** (*`float`*) – Diameter limit in meters below witch sampling objects are not included. Can be `None` to skip filtering.
- **`seed`** (*`int`*) – Random number generator seed given to `numpy.random.seed` to allow for consisted generation of a random realization of the population. If seed is `None` a random seed from high-entropy data is used.
- **`propagator`** (`PropagatorBase`) – Propagator class pointer used for *`space_object.SpaceObject`*. Is not used if `master_base` is given.
- **`propagator_options`** (*`dict`*) – Propagator initialization keyword arguments. Is not used if `master_base` is given.

Returns Master population

Return type *population.Population*

`population_library.`**`propagate_population`**(*population, mjd*)

`population_library.`**`simulate_Microsat_R_debris`**(*num, max_dv, radii_range, mass_range, C_D_range, seed, propagator, propagator_options, mjd*)

`population_library.`**`simulate_Microsat_R_debris_v2`**(*num, max_dv, rho_range, mass_range, seed, propagator, propagator_options, mjd*)

`population_library.`**`tle_snapshot`**(*tle_file, sgp4_propagation=True, propagator=<class 'propagator_sgp4.PropagatorSGP4'>, propagator_options={}*)

Reads a TLE-snapshot file and converts the TLE's to orbits in a TEME frame and creates a population file. The BSTAR parameter is saved in column BSTAR (or `_objs[:,12]`). A snapshot generally contains several TLE's for the same object thus will this population also contain duplicate objects.

Numerical propagator assumptions: To propagate with a numerical propagator one needs to make assumptions.

- Density is :math:'5cdot 10^3 ;

rac{kg}{m^3}‘.

- Object is a sphere
- Drag coefficient is 2.3.

param str/list tle_file Path to the input TLE snapshot file. Or the TLE-set can be given directly as a list of two lines that can be unpacked in a loop, e.g. `[(tle1_l1, tle1_l2), (tle2_l1, tle2_l2)]`.

param bool sgp4_propagation If `False` then the population is specifically constructed to be propagated with *`propagator_orekit.PropagatorOrekit`* and assumptions are made on mass, density and shape of objects. Otherwise the *`space_object.SpaceObject`* is configured to use SGP4 propagation.

return TLE snapshot as a population with numerical propagator

rtype population.Population

7.4.3 antenna_library

A collection of functions that return common instances of the *`BeamPattern`* class.

Contains for example:

- Uniformly filled circular aperture of radius a
- Cassegrain antenna with radius a0 and subreflector radius a1
- Planar gaussian illuminated aperture (approximates a phased array)

Reference: https://www.cv.nrao.edu/course/astr534/2DApertures.html

`antenna_library.`**`airy`**(*k_in*, *beam*)

# TODO: Descriptive doc string.

a = radius f = frequency I_0 = gain at center

`antenna_library.`**`airy_beam`**(*az0*, *el0*, *I_0*, *f*, *a*)

# TODO: Description.

`antenna_library.`**`array`**(*k_in*, *beam*)

# TODO: Description.

`antenna_library.`**`array_beam`**(*az0*, *el0*, *I_0*, *f*, *antennas*)

# TODO: Description.

`antenna_library.`**`cassegrain`**(*k_in*, *beam*)

# TODO: Descriptive doc string.

A better model of the EISCAT UHF antenna

`antenna_library.`**`cassegrain_beam`**(*az0*, *el0*, *I_0*, *f*, *a0*, *a1*, *beam_name='Cassegrain'*)

# TODO: Description.

az and el of on-axis lat and lon of location I_0 gain on-axis a0 diameter of main reflector a1 diameter of the subreflector

`antenna_library.`**`e3d_array`**(*f*, *fname='data/e3d_array.txt'*)

# TODO: Description.

`antenna_library.`**`e3d_array_beam`**(*az0=0*, *el0=90.0*, *I_0=31622.776601683792*, *fname='data/e3d_array.txt'*)

# TODO: Description.

45dB peak according to e3d specs: Technical specification and requirements for antenna unit

`antenna_library.`**`e3d_array_beam_interp`**(*az0=0*, *el0=90.0*, *I_0=15848.93192461114*, *fname='data/inerp_e3d.npy'*, *res=400*)

`antenna_library.`**`e3d_array_beam_stage1`**(*az0=0*, *el0=90.0*, *I_0=15848.93192461114*, *fname='data/e3d_array.txt'*, *opt='dense'*)

# TODO: Description.

45dB-3dB=42dB peak according to e3d specs: Technical specification and requirements for antenna unit

`antenna_library.`**`e3d_array_beam_stage1_dense_interp`**(*az0=0*, *el0=90.0*, *I_0=15848.93192461114*, *fname='data/inerp_e3d_stage1_dense.npy'*, *res=400*)

`antenna_library.`**`e3d_array_stage1`**(*f*, *fname='data/e3d_array.txt'*, *opt='dense'*)

# TODO: Description.

`antenna_library.`**`e3d_module_beam`**(*az0=0*, *el0=90.0*, *I_0=158.48931924611142*)

# TODO: Description.

`antenna_library.`**`e3d_subarray`**(*f*)

# TODO: Description.

`antenna_library.`**`elliptic`**(*k_in*, *beam*)

# TODO: Description.

TDB: sqrt(u**2 + c**2 v**2)

http://www.iue.tuwien.ac.at/phd/minixhofer/node59.html https://en.wikipedia.org/wiki/Fraunhofer_diffraction_equation

x=n.linspace(-2,2,num=1024) xx,yy=n.meshgrid(x,x)

A=n.zeros([1024,1024]) A[xx**2.0/0.25**2 + yy**2.0/0.0625**2.0 < 1.0]=1.0

plt.pcolormesh(10.0*n.log10(n.fft.fftshift(n.abs(B)))) plt.colorbar() plt.axis("equal") plt.show()

Variable substitution

`antenna_library.`**`elliptic_airy`**(*k_in*, *beam*)

# TODO: Descriptive doc string.

a = radius f = frequency I_0 = gain at center

`antenna_library.`**`interpolated_beam`**(*k_in*, *beam*)

Assume that the interpolated grid at zenith is merely shifted to the pointing direction and scaled by the sine of the elevation angle.

`antenna_library.`**`planar`**(*k_in*, *beam*)

Gaussian tapered planar array

`antenna_library.`**`planar_beam`**(*az0*, *el0*, *I_0*, *f*, *a0*, *az1*, *el1*)

# TODO: Description.

`antenna_library.`**`plane_wave`**(*k*, *r*, *p*)

The complex plane wave function.

Parameters

- **k** (*numpy.ndarray*) – Wave-vector (wave propagation direction)

- **r** (*numpy.ndarray*) – Spatial location (Antenna position in space)
- **p** (*numpy.ndarray*) – Beam-forming direction (antenna array "pointing" direction)

antenna_library.**scaled_EISCAT_VHF_fixed**(*k_in*, *beam*)

antenna_library.**tsr_beam**(*el0*, *f*)

antenna_library.**uhf_beam**(*az0*, *el0*, *I_0*, *f*, *beam_name='UHF Measured beam'*)
# TODO: Description.

antenna_library.**uhf_meas**(*k_in*, *beam*)
Measured UHF beam pattern

7.4.4 radar_scan_library

A collection of *radar_scans.RadarScan* instances, such as fence scans or ionospheric grids.

radar_scan_library.**beampark_model**(*az*, *el*, *lat*, *lon*, *alt*, *name='Beampark'*)
A beampark model.

Parameters

- **az** (*float*) – Azimuth in degrees east of north.
- **el** (*float*) – Elevation in degrees above horizon.
- **lat** (*float*) – Geographical latitude of radar system in decimal degrees (North+).
- **lon** (*float*) – Geographical longitude of radar system in decimal degrees (East+).
- **alt** (*float*) – Geographical altitude above geoid surface of radar system in meter.

radar_scan_library.**calculate_fence_angles**(*min_el*, *angle_step*)
Calculate a vector of angles to be used in a fence scan.

radar_scan_library.**ew_fence_model**(*lat*, *lon*, *alt*, *min_el=30*, *angle_step=1.0*, *dwell_time=0.1*)

radar_scan_library.**flat_grid_model**(*lat*, *lon*, *alt*, *n_side=3*, *height=300000.0*, *side_len=100000.0*, *dwell_time=0.4*)

radar_scan_library.**n_const_pointing_model**(*az*, *el*, *lat*, *lon*, *alt*, *dwell_time=0.1*)
Model for a n-point beampark with fixed dwell-times.

Parameters

- **az** (*list*) – Azimuths in degrees east of north.
- **el** (*list*) – Elevations in degrees above horizon.
- **lat** (*float*) – Geographical latitude of radar system in decimal degrees (North+).
- **lon** (*float*) – Geographical longitude of radar system in decimal degrees (East+).
- **alt** (*float*) – Geographical altitude above geoid surface of radar system in meter.
- **dwell_time** (*float*) – Time spent at each azimuth-elevation pair of pointing direction.

radar_scan_library.**n_dyn_dwell_pointing_model**(*az*, *el*, *dwells*, *lat*, *lon*, *alt*)
Model for a n-point beampark with variable dwell-times.

Parameters

- **az** (*list*) – Azimuths in degrees east of north.
- **el** (*list*) – Elevations in degrees above horizon.

- **dwells** (*list*) – Times to spend at each azimuth-elevation pair of pointing direction.
- **lat** (*float*) – Geographical latitude of radar system in decimal degrees (North+).
- **lon** (*float*) – Geographical longitude of radar system in decimal degrees (East+).
- **alt** (*float*) – Geographical altitude above geoid surface of radar system in meter.

radar_scan_library.**ns_fence_model**(*lat*, *lon*, *alt*, *min_el=30*, *angle_step=1*, *dwell_time=0.1*)

radar_scan_library.**ns_fence_rng_model**(*lat*, *lon*, *alt*, *min_el=30*, *angle_step=1*, *dwell_time=0.1*)

radar_scan_library.**point_beampark**(*t*, *az*, *el*, ***kw*)
Pointing function for a single point beampark. AZ-EL coordinate system.

Parameters

- **t** (*float*) – Seconds past epoch.
- **az** (*float*) – Azimuth in degrees east of north.
- **el** (*float*) – Elevation in degrees above horizon.

radar_scan_library.**point_circle_fence**(*t*, *az*, *el*, *dwell_time*, ***kw*)

radar_scan_library.**point_cross_fence_scan**(*t*, *angles*, *dwell_time*, *state*, ***kw*)

radar_scan_library.**point_ew_fence_scan**(*t*, *angles*, *dwell_time*, ***kw*)
Pointing function for a east-west fence scan. NED coordinate system.

radar_scan_library.**point_n_beampark**(*t*, *az*, *el*, *dwell_time*, ***kw*)
Pointing function for a n-point beampark with fixed dwell-times. AZ-EL coordinate system.

Parameters

- **t** (*float*) – Seconds past epoch.
- **az** (*list*) – Azimuths in degrees east of north.
- **el** (*list*) – Elevations in degrees above horizon.
- **dwell_time** (*float*) – Time in seconds spent at each azimuth-elevation pair of pointing direction.

radar_scan_library.**point_n_dyn_beampark**(*t*, *az*, *el*, *scan_time*, *dwell_time*, ***kw*)
Pointing function for a n-point beampark with variable dwell-times. AZ-EL coordinate system.

Parameters

- **t** (*float*) – Seconds past epoch.
- **az** (*list*) – Azimuths in degrees east of north.
- **el** (*list*) – Elevations in degrees above horizon.
- **scan_time** (*float*) – Total scan time, sum of dwell times.
- **dwell_time** (*list*) – Times in seconds to spend at each azimuth-elevation pair of pointing direction.

radar_scan_library.**point_ns_fence_rng_scan**(*t*, *angles*, *dwell_time*, *state*, ***kw*)
Pointing function for a uniform radnom north-south fence scan using a fixed state at epoch, i.e reproducable sequence. NED coordinate system.

radar_scan_library.**point_ns_fence_scan**(*t*, *angles*, *dwell_time*, ***kw*)
Pointing function for a north-south fence scan. NED coordinate system.

radar_scan_library.**point_sph_rng_scan**(*t*, *dwell_time*, *min_el*, *state*, ***kw*)
Pointing function for a spherically uniform random scan with a minimum elevation and fixed state at epoch, i.e reproducable sequence. NED coordinate system.

radar_scan_library.**sph_rng_model**(*lat*, *lon*, *alt*, *min_el=30*, *dwell_time=0.1*)

7.4.5 scheduler_library

Collection of classes and functions related to constructing a radar system scheduler.

scheduler_library.**dynamic_scheduler**(*catalogue*, *radar*, *parameters*, *t0*, *t1*, ***kwargs*)
Dynamic scheduler

scheduler_library.**isolated_static_discovery_sceduler**(*sim*, *config*)

scheduler_library.**memory_static_discovery_sceduler**(*sim*, *config*)

scheduler_library.**scheduler_TEMPLATE**(*catalogue*, *radar*, *parameters*, *t0*, *t1*, ***kwargs*)

scheduler_library.**scheduling_movie**(*tracks*, *tracks_t*, *radar*, *population*, *root*, *time_slice=0.2*, *time_len=0.0005555555555555556*)

7.4.6 rewardf_library

rewardf_library.**rewardf_TEMPLATE**(*t*, *track*, *config*, ***kw*)

rewardf_library.**rewardf_exp_peak_SNR**(*t*, *track*, *config*, ***kw*)
Desc

rewardf_library.**rewardf_exp_peak_SNR_tracklet_len**(*t*, *track*, *config*, ***kw*)
Reward function that uses time from peak SNR and the number of accumulated data points as parameters to a normal and exponential distribution.

Config:

- sigma_t [float]: Desc
- lambda_N [float]: Desc

7.5 Integrator interfaces

propagator_sgp4	SGP4 interface with SORTS++.
propagator_orekit	Wrapper for the Orekit propagator into SORTS++ format.
propagator_kepler	Kepler propagation interface with SORTS++.
propagator_neptune	

7.5.1 propagator_sgp4

SGP4 interface with SORTS++.

Written in 2018 by Juha Vierinen based on code from Jan Siminski, ESA. Modified by Daniel Kastinen 2018/2019

propagator_sgp4.**MU_earth = 398600800000000.0**
float: Standard gravitational parameter of the Earth using the WGS72 convention.

`propagator_sgp4.`**`M_earth = 5.972370723755184e+24`**

float: Mass of the Earth using the WGS72 convention.

`class` `propagator_sgp4.`**`PropagatorSGP4`**(*polar_motion=False*, *polar_motion_model='80'*, *out_frame='ITRF'*)

Bases: `propagator_base.PropagatorBase`

Propagator class implementing the SGP4 propagator.

Variables

- **`polar_motion`** (*`bool`*) – Determines if polar motion should be used in calculating ITRF frame.
- **`polar_motion_model`** (*`str`*) – String identifying the polar motion model to be used. Options are '80' or '00'.
- **`out_frame`** (*`str`*) – String identifying the output frame. Options are 'ITRF' or 'TEME'.

The constructor creates a propagator instance with supplied options.

Parameters

- **`polar_motion`** (*`bool`*) – Determines if polar motion should be used in calculating ITRF frame.
- **`polar_motion_model`** (*`str`*) – String identifying the polar motion model to be used. Options are '80' or '00'.
- **`out_frame`** (*`str`*) – String identifying the output frame. Options are 'ITRF' or 'TEME'.

`get_orbit`(*t*, *a*, *e*, *inc*, *raan*, *aop*, *mu0*, *mjd0*, ***kwargs*)

Implementation:

All state-vector units are in meters.

Keyword arguments contain only information needed for ballistic coefficient `B` used by SGP4. Either `B` or `C_D`, `A` and `m` must be supplied. They also contain a option to give angles in radians or degrees. By default input is assumed to be degrees.

Frame:

The input frame is ECI (TEME) for orbital elements and Cartesian. The output frame is as standard ECEF (ITRF). But can be set to TEME.

Parameters

- **`B`** (*`float`*) – Ballistic coefficient
- **`C_D`** (*`float`*) – Drag coefficient
- **`A`** (*`float`*) – Cross-sectional Area
- **`m`** (*`float`*) – Mass
- **`radians`** (*`bool`*) – If true, all angles are assumed to be in radians.

`get_orbit_TLE`(*t*, *line1*, *line2*)

Takes a TLE and propagates it forward in time directly using the SGP4 algorithm.

Parameters

- **`t`** (*`float/list/numpy.ndarray`*) – Time in seconds to propagate relative the initial state epoch.
- **`line1`** (*`str`*) – TLE line 1
- **`line2`** (*`str`*) – TLE line 2

get_orbit_cart (*t*, *x*, *y*, *z*, *vx*, *vy*, *vz*, *mjd0*, ***kwargs*)

Implementation:

All state-vector units are in meters.

Keyword arguments contain only information needed for ballistic coefficient `B` used by SGP4. Either `B` or `C_D`, `A` and `m` must be supplied. They also contain a option to give angles in radians or degrees. By default input is assumed to be degrees.

Frame:

The input frame is ECI (TEME) for orbital elements and Cartesian. The output frame is always ECEF.

Parameters

- **B** (*float*) – Ballistic coefficient
- **C_D** (*float*) – Drag coefficient
- **A** (*float*) – Cross-sectional Area
- **m** (*float*) – Mass
- **radians** (*bool*) – If true, all angles are assumed to be in radians.

class propagator_sgp4.**PropagatorTLE** (*polar_motion=False*, *polar_motion_model='80'*, *out_frame='ITRF'*)

Bases: *propagator_sgp4.PropagatorSGP4*

get_orbit (*t*, *a*, *e*, *inc*, *raan*, *aop*, *mu0*, *mjd0*, ***kwargs*)

Implementation:

Direct propagation of TLE. All input elements are ignored except for the two lines in `**kwargs`.

Frame:

The output frame is (ITRF) ECEF or (TEME) ECI.

Parameters

- **line1** (*str*) – TLE line 1
- **line2** (*str*) – TLE line 2

get_orbit_cart (*t*, *x*, *y*, *z*, *vx*, *vy*, *vz*, *mjd0*, ***kwargs*)

Implementation:

Direct propagation of TLE. All input elements are ignored except for the two lines in `**kwargs`.

Frame:

The output frame is (ITRF) ECEF or (TEME) ECI.

Parameters

- **line1** (*str*) – TLE line 1
- **line2** (*str*) – TLE line 2

class propagator_sgp4.**SGP4** (*mjd_epoch*, *mean_elements*, *B*)

The SGP4 class acts as a wrapper around the sgp4 module uploaded by Brandon Rhodes (http://pypi.python.org/pypi/sgp4/).

It converts orbital elements into the TLE-like 'satellite'-structure which is used by the module for the propagation.

Notes: The class can be directly used for propagation. Alternatively, a simple propagator function is provided below.

GM = 398600.8

MJD_0 = 2400000.5

Q0 = 120.0

RHO0 = 2.461e-08

R_EARTH = 6378.135

S0 = 78.0

WGS = EarthGravity(tumin=13.446839696959309, mu=398600.8, radiusearthkm=6378.135, xke=

position (*mjd*)
Inertial position at epoch mjd

Parameters **mjd** (*float*) – epoch where satellite should be propagated to

state (*mjd*)
Inertial position and velocity ([m], [m/s]) at epoch mjd

Parameters **mjd** (*float*) – epoch where satellite should be propagated to

velocity (*mjd*)
Inertial velocity at epoch mjd

Parameters **mjd** (*float*) – epoch where satellite should be propagated to

propagator_sgp4.**ecef2teme** (*t*, *p*, *v*, *mjd0=57084*, *xp=0.0*, *yp=0.0*, *model='80'*, *lod=0.0015563*)
Reverse operation, developed by Daniel Kastinen 2019

# TODO: Write proper docstring p, v and output are all in units of km and km/s, as for teme2ecef

propagator_sgp4.**gstime** (*jdut1*)
This function finds the greenwich sidereal time (iau-82).

References: Vallado 2007, 193, Eq 3-43

Author: David Vallado 719-573-2600 7 jun 2002 Adapted to Python, Daniel Kastinen 2018

Parameters **jdut1** (*float*) – Julian date of ut1 in days from 4713 bc

Returns Greenwich sidereal time in radians, 0 to 2π

Return type float

propagator_sgp4.**polarm** (*xp*, *yp*, *ttt*, *opt*)
This function calculates the transformation matrix that accounts for polar motion. both the 1980 and 2000 theories are handled. note that the rotation order is different between 1980 and 2000.

References: Vallado 2004, 207-209, 211, 223-224.

Author: David Vallado 719-573-2600 25 jun 2002. Adapted to Python, Daniel Kastinen 2018

Parameters

- **xp** (*float*) – x-axis polar motion coefficient in radians
- **yp** (*float*) – y-axis polar motion coefficient in radians
- **ttt** (*float*) – Julian centuries of tt (00 theory only)
- **opt** (*str*) – Model for polar motion to use, options are '01', '02', '80'.

Returns Transformation matrix for ECEF to PEF

Return type numpy.ndarray (3x3 matrix)

`propagator_sgp4.`**`sgp4_propagation`**(*mjd_epoch*, *mean_elements*, *B=0.0*, *dt=0.0*, *method=None*)

Lazy SGP4 propagation using SGP4 class

Create a satellite object from mean elements and propagate it

:param list/numpy.ndarray mean_elements : [a0,e0,i0,raan0,aop0,M0] :param float B: Ballistic coefficient (0.5*C_D*A/m) :param float dt: Time difference w.r.t. element epoch in seconds :param float mjd_epoch: Epoch of elements as Modified Julian Date (MJD) Can be ignored if the exact epoch is unimportant. :param str method: Forces use of SGP4 or SDP4 depending on string 'n' or 'd'

`propagator_sgp4.`**`teme2ecef`**(*t*, *p*, *v*, *mjd0=57084*, *xp=0.0*, *yp=0.0*, *model='80'*, *lod=0.0015563*)

This function trsnforms a vector from the true equator mean equniox frame (teme), to an earth fixed (ITRF) frame. the results take into account the effects of sidereal time, and polar motion.

References: Vallado 2007, 219-228.

Author: David Vallado 719-573-2600, 10 dec 2007. Adapted to Python, Daniel Kastinen 2018

Parameters

- **`t`** (*`numpy.ndarray`*) – numpy vector row of seconds relative to `mjd0`
- **`p`** (*`numpy.ndarray`*) – numpy matrix of TEME positions, Cartesian coordinates in km. Columns correspond to times in t, and rows to x, y and z coordinates respectively.
- **`v`** (*`numpy.ndarray`*) – numpy matrix of TEME velocities, Cartesian coordinates in km/s. Columns correspond to times in t, and rows to x, y and z coordinates respectively.
- **`mjd0`** (*`float`*) – Modified julian date epoch that t vector is relative to
- **`xp`** (*`float`*) – x-axis polar motion coefficient in radians
- **`yp`** (*`float`*) – y-axis polar motion coefficient in radians
- **`model`** (*`str`*) – The polar motion model used in transformation, options are '80' or '00', see David Vallado documentation for more info.
- **`lod`** (*`float`*) – Excess length of day in seconds

Returns State vector of position and velocity in km and km/s.

Return type numpy.ndarray (6-D vector)

Uses:

- *`propagator_sgp4.gstime()`*
- *`propagator_sgp4.polarm()`*

[recef,vecef,aecef] = teme2ecef (rteme,vteme,ateme,ttt,jdut1,lod,xp,yp);

7.5.2 propagator_orekit

Wrapper for the Orekit propagator into SORTS++ format.

Links:

- orekit
- orekit python
- orekit python guide
- Hipparchus

- orekit 9.3 api
- JCC

Example usage:

Simple propagation showing time difference due to loading of model data.

```python
from propagator_orekit import PropagatorOrekit
import time
import numpy as n
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

t0 = time.time()
p = PropagatorOrekit()
print('init time: {} sec'.format(time.time() - t0))

init_data = {
    'a': 9000,
    'e': 0.0,
    'inc': 90.0,
    'raan': 10,
    'aop': 10,
    'mu0': 40.0,
    'mjd0': 57125.7729,
    'C_D': 2.3,
    'C_R': 1.0,
    'm': 8000,
    'A': 1.0,
}
t = n.linspace(0,24*3600.0, num=1000, dtype=n.float)

t0 = time.time()
ecefs = p.get_orbit(t, **init_data)
print('get orbit time (first): {} sec'.format(time.time() - t0))


t0 = time.time()
ecefs = p.get_orbit(t, **init_data)
print('get orbit time (second): {} sec'.format(time.time() - t0))

fig = plt.figure(figsize=(15,15))
ax = fig.add_subplot(111, projection='3d')
ax.plot(ecefs[0,:], ecefs[1,:], ecefs[2,:],".",color="green")
plt.show()
```

Propagation using custom settings:

```python
from propagator_orekit import PropagatorOrekit
import numpy as n
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

init_data = {
    'a': 7500,
    'e': 0.1,
    'inc': 90.0,
    'raan': 10,
```

(continues on next page)

(continued from previous page)

```
    'aop': 10,
    'mu0': 40.0,
    'mjd0': 57125.7729,
    'C_D': 2.3,
    'C_R': 1.0,
    'm': 8000,
    'A': 1.0,
}
t = n.linspace(0,10*3600.0, num=10000, dtype=n.float)


p2 = PropagatorOrekit()
print(p2)
ecefs2 = p2.get_orbit(t, **init_data)

p1 = PropagatorOrekit(earth_gravity='Newtonian', radiation_pressure=False,␣
↪solarsystem_perturbers=[], drag_force=False)
print(p1)
ecefs1 = p1.get_orbit(t, **init_data)


dr = n.sqrt(n.sum((ecefs1[:3,:] - ecefs2[:3,:])**2, axis=0))
dv = n.sqrt(n.sum((ecefs1[3:,:] - ecefs2[3:,:])**2, axis=0))

r1 = n.sqrt(n.sum(ecefs1[:3,:]**2, axis=0))
r2 = n.sqrt(n.sum(ecefs1[:3,:]**2, axis=0))

fig = plt.figure(figsize=(15,15))
ax = fig.add_subplot(311)
ax.plot(t/3600.0, dr*1e-3)
ax.set_xlabel('Time [h]')
ax.set_ylabel('Position difference [km]')
ax.set_title('Propagation difference diameter simple vs advanced models')
ax = fig.add_subplot(312)
ax.plot(t/3600.0, dv*1e-3)
ax.set_xlabel('Time [h]')
ax.set_ylabel('Velocity difference [km/s]')
ax = fig.add_subplot(313)
ax.plot(t/3600.0, r1*1e-3, color="green",label='Simple model')
ax.plot(t/3600.0, r2*1e-3, color="red",label='Advanced model')
ax.set_xlabel('Time [h]')
ax.set_ylabel('Distance from Earth center [km]')
plt.legend()


fig = plt.figure(figsize=(15,15))
ax = fig.add_subplot(111, projection='3d')
plothelp.draw_earth_grid(ax)
ax.plot(ecefs1[0,:], ecefs1[1,:], ecefs1[2,:],"-",alpha=0.5,color="green",label=
↪'Simple model')
ax.plot(ecefs2[0,:], ecefs2[1,:], ecefs2[2,:],"-",alpha=0.5,color="red",label=
↪'Advanced model')
plt.legend()
plt.show()
```

Propagation using different coordinate systems:

```
from propagator_orekit import PropagatorOrekit
import numpy as n
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

p = PropagatorOrekit(in_frame='ITRF', out_frame='ITRF')

print(p)

init_data = {
    'a': R_e + 400.0,
    'e': 0.01,
    'inc': 90.0,
    'raan': 10,
    'aop': 10,
    'mu0': 40.0,
    'mjd0': 57125.7729,
    'C_D': 2.3,
    'C_R': 1.0,
    'm': 3.0,
    'A': n.pi*1.0**2,
}
t = n.linspace(0,3*3600.0, num=500, dtype=n.float)

ecefs1 = p.get_orbit(t, **init_data)

print(p)

p = PropagatorOrekit(in_frame='EME', out_frame='ITRF')

ecefs2 = p.get_orbit(t, **init_data)

fig = plt.figure(figsize=(15,15))
ax = fig.add_subplot(111, projection='3d')
plothelp.draw_earth_grid(ax)
ax.plot(ecefs1[0,:], ecefs1[1,:], ecefs1[2,:],".",color="green",label='Initial frame:␣
↪ITRF')
ax.plot(ecefs2[0,:], ecefs2[1,:], ecefs2[2,:],".",color="red",label='Initial frame:␣
↪EME')
plt.legend()
plt.show()
```

class propagator_orekit.**PropagatorOrekit**(*in_frame='EME', out_frame='ITRF', frame_tidal_effects=False, integrator='DormandPrince853', min_step=0.001, max_step=120.0, position_tolerance=10.0, earth_gravity='HolmesFeatherstone', gravity_order=(10, 10), solarsystem_perturbers=['Moon', 'Sun'], drag_force=True, atmosphere='DTM2000', radiation_pressure=True, solar_activity='Marshall', constants_source='WGS84', solar_activity_strength='WEAK'*)

Bases: *propagator_base.PropagatorBase*

Propagator class implementing the Orekit propagator.

Variables

- **solarsystem_perturbers** (*list*) – List of strings of names of objects in the solarsystem that should be used for third body perturbation calculations. All objects listed at CelestialBodyFactory are available.
- **in_frame** (*str*) – String identifying the input frame to be used. All frames listed at FramesFactory are available.
- **out_frame** (*str*) – String identifying the output frame to be used. All frames listed at FramesFactory are available.
- **drag_force** (*bool*) – Should drag force be included in propagation.
- **radiation_pressure** (*bool*) – Should radiation pressure force be included in propagation.
- **frame_tidal_effects** (*bool*) – Should coordinate frames include Tidal effects.
- **integrator** (*str*) – String representing the numerical integrator from the Hipparchus package to use. Any integrator listed at Hipparchus nonstiff ode is available.
- **minStep** (*float*) – Minimum time step allowed in the numerical orbit propagation given in seconds.
- **maxStep** (*float*) – Maximum time step allowed in the numerical orbit propagation given in seconds.
- **position_tolerance** (*float*) – Position tolerance in numerical orbit propagation errors given in meters.
- **earth_gravity** (*str*) – Gravitation model to use for calculating central acceleration force. Currently avalible options are *'HolmesFeatherstone'* and *'Newtonian'*. See gravity.
- **gravity_order** (*tuple*) – A tuple of two integers for describing the order of spherical harmonics used in the HolmesFeatherstoneAttractionModel model.
- **atmosphere** (*str*) – Atmosphere model used to calculate atmospheric drag. Currently available options are *'DTM2000'*. See atmospheres.
- **solar_activity** (*str*) – The model used for calculating solar activity and thereby the influx of solar radiation. Used in the atmospheric drag force model. Currently available options are *'Marshall'* for the MarshallSolarActivityFutureEstimation.
- **constants_source** (*str*) – Controls which source for Earth constants to use. Currently avalible options are *'WGS84'* and *'JPL-IAU'*. See constants.
- **mu** (*float*) – Standard gravitational constant for the Earth. Definition depend on the `propagator_orekit.PropagatorOrekit` constructor parameter `constants_source`
- **R_earth** (*float*) – Radius of the Earth in m. Definition depend on the `propagator_orekit.PropagatorOrekit` constructor parameter `constants_source`
- **f_earth** (*float*) – Flattening of the Earth (i.e. $\frac{a-b}{a}$). Definition depend on the `propagator_orekit.PropagatorOrekit` constructor parameter `constants_source`.
- **M_earth** (*float*) – Mass of the Earth in kg. Definition depend on the `propagator_orekit.PropagatorOrekit` constructor parameter `constants_source`
- **inputFrame** (*org.orekit.frames.Frame*) – The orekit frame instance for the input frame.

- **outputFrame** (*org.orekit.frames.Frame*) – The orekit frame instance for the output frame.
- **inertialFrame** (*org.orekit.frames.Frame*) – The orekit frame instance for the inertial frame. If inputFrame is pseudo innertial this is the same as inputFrame.
- **body** (*org.orekit.bodies.OneAxisEllipsoid*) – The model ellipsoid representing the Earth.
- **_forces** (*dict*) – Dictionary of forces to include in the numerical integration. Contains instances of children of `org.orekit.forces.AbstractForceModel`.
- **_tolerances** (*list*) – Contains the absolute and relative tolerances calculated by the tolerances function.
- **propagator** (*org.orekit.propagation.numerical.NumericalPropagator*) – The numerical propagator instance.
- **SolarStrengthLevel** (*org.orekit.forces.drag.atmosphere.data.MarshallSolarActivityFutureEstimation.StrengthLevel*) – The strength of the solar activity. Options are 'AVRAGE', 'STRONG', 'WEAK'.

The constructor creates a propagator instance with supplied options.

Parameters

- **solarsystem_perturbers** (*list*) – List of strings of names of objects in the solarsystem that should be used for third body perturbation calculations. All objects listed at CelestialBodyFactory are available.
- **in_frame** (*str*) – String identifying the input frame to be used. All frames listed at FramesFactory are available.
- **out_frame** (*str*) – String identifying the output frame to be used. All frames listed at FramesFactory are available.
- **drag_force** (*bool*) – Should drag force be included in propagation.
- **radiation_pressure** (*bool*) – Should radiation pressure force be included in propagation.
- **frame_tidal_effects** (*bool*) – Should coordinate frames include Tidal effects.
- **integrator** (*str*) – String representing the numerical integrator from the Hipparchus package to use. Any integrator listed at Hipparchus nonstiff ode is available.
- **min_step** (*float*) – Minimum time step allowed in the numerical orbit propagation given in seconds.
- **max_step** (*float*) – Maximum time step allowed in the numerical orbit propagation given in seconds.
- **position_tolerance** (*float*) – Position tolerance in numerical orbit propagation errors given in meters.
- **atmosphere** (*str*) – Atmosphere model used to calculate atmospheric drag. Currently available options are *'DTM2000'*. See atmospheres.
- **solar_activity** (*str*) – The model used for calculating solar activity and thereby the influx of solar radiation. Used in the atmospheric drag force model. Currently available options are *'Marshall'* for the MarshallSolarActivityFutureEstimation.
- **constants_source** (*str*) – Controls which source for Earth constants to use. Currently avalible options are *'WGS84'* and *'JPL-IAU'*. See constants.

- **earth_gravity** (*str*) – Gravitation model to use for calculating central acceleration force. Currently avalible options are *'HolmesFeatherstone'* and *'Newtonian'*. See gravity.
- **gravity_order** (*tuple*) – A tuple of two integers for describing the order of spherical harmonics used in the HolmesFeatherstoneAttractionModel model.
- **solar_activity_strength** (*str*) – The strength of the solar activity. Options are 'AVRAGE', 'STRONG', 'WEAK'.

class OrekitVariableStep

Bases: `PythonOrekitStepHandler`

Class for handling the steps.

handleStep (*interpolator*, *isLast*)

init (*s0*, *t*)

set_params (*t*, *start_date*, *states_pointer*, *outputFrame*)

get_orbit (*t*, *a*, *e*, *inc*, *raan*, *aop*, *mu0*, *mjd0*, ***kwargs*)

Implementation:

Units are in meters and degrees.

Keyword arguments are:

- float A: Area in m^2
- float C_D: Drag coefficient
- float C_R: Radiation pressure coefficient
- float m: Mass of object in kg

NOTE:

- If the eccentricity is below 1e-10 the eccentricity will be set to 1e-10 to prevent Keplerian Jacobian becoming singular.

The implementation first checks if the input frame is Pseudo inertial, if this is true this is used as the propagation frame. If not it is automatically converted to EME (ECI-J2000).

Since there are forces that are dependent on the space-craft parameters, if these parameter has been changed since the last iteration the numerical integrator is re-initialized at every call of this method. The forces that can be initialized without spacecraft parameters (e.g. Earth gravitational field) are done at propagator construction.

Uses:

- `propagator_base.PropagatorBase._make_numpy()`
- `propagator_orekit.PropagatorOrekit._construct_propagator()`
- `propagator_orekit.PropagatorOrekit._set_forces()`
- `dpt_tools.kep2cart()`
- `dpt_tools.cart2kep()`
- `dpt_tools.true2mean()`
- `dpt_tools.mean2true()`

See `propagator_base.PropagatorBase.get_orbit()`.

`get_orbit_cart` (*t*, *x*, *y*, *z*, *vx*, *vy*, *vz*, *mjd0*, ***kwargs*)

Implementation:

Converts Cartesian vector to Kepler elements and calls *`propagator_orekit.PropagatorOrekit.get_orbit()`*.

All units are in m and m/s.

Uses:

- *`dpt_tools.cart2kep()`*
- *`dpt_tools.true2mean()`*

See *`propagator_base.PropagatorBase.get_orbit_cart()`*.

`propagator_orekit.`**`frame_conversion`** (*state*, *mjd*, **args*, ***kwargs*)

`propagator_orekit.`**`iter_states`** (*fun*)

`propagator_orekit.`**`mjd2absdate`** (*mjd*)

Converts a Modified Julian Date value to an orekit AbsoluteDate

`propagator_orekit.`**`npdt2absdate`** (*dt*)

Converts a numpy datetime64 value to an orekit AbsoluteDate

7.5.3 propagator_kepler

Kepler propagation interface with SORTS++.

`class` `propagator_kepler.`**`PropagatorKepler`** (*in_frame='EME'*, *out_frame='ITRF'*, *frame_tidal_effects=False*)

Bases: *`propagator_base.PropagatorBase`*

Propagator class implementing a analytic Kepler propagator.

The constructor creates a propagator instance.

Variables

- **`in_frame`** (*`str`*) – String identifying the input frame to be used. All frames listed at FramesFactory are available.
- **`out_frame`** (*`str`*) – String identifying the output frame to be used. All frames listed at FramesFactory are available.
- **`frame_tidal_effects`** (*`bool`*) – Should coordinate frames include Tidal effects.

Parameters

- **`in_frame`** (*`str`*) – String identifying the input frame to be used. All frames listed at FramesFactory are available.
- **`out_frame`** (*`str`*) – String identifying the output frame to be used. All frames listed at FramesFactory are available.
- **`frame_tidal_effects`** (*`bool`*) – Should coordinate frames include Tidal effects.

`get_orbit` (*t*, *a*, *e*, *inc*, *raan*, *aop*, *mu0*, *mjd0*, ***kwargs*)

Implementation:

All state-vector units are in meters.

Keyword arguments contain only mass `m` in kg and is not required.

They also contain a option to give angles in radians or degrees. By default input is assumed to be degrees.

Frame:

The input frame is always the same as the output frame.

Parameters

- **m** (*float*) – Mass of the object in kg.
- **radians** (*bool*) – If true, all angles are assumed to be in radians.

get_orbit_cart (*t, x, y, z, vx, vy, vz, mjd0, **kwargs*)

Implementation:

All state-vector units are in meters.

Keyword arguments contain only mass `m` in kg and is not required.

They also contain a option to give angles in radians or degrees. By default input is assumed to be degrees.

Frame:

The input frame is always the same as the output frame.

Parameters

- **m** (*float*) – Mass of the object in kg.
- **radians** (*bool*) – If true, all angles are assumed to be in radians.

output_to_input_frame (*states, t, mjd0*)

7.5.4 propagator_neptune

7.6 Tests

7.6.1 unit-tests

List of tests

```
tests/test_TLE_tools.py::TestTLE::test_TEME_to_ITRF_circle_polar
tests/test_TLE_tools.py::TestTLE::test_TEME_to_ITRF_equatorial_circle
tests/test_TLE_tools.py::TestTLE::test_TEME_to_TLE
tests/test_TLE_tools.py::TestTLE::test_TEME_to_TLE_cases
tests/test_TLE_tools.py::TestTLE::test_TLE_to_TEME
tests/test_TLE_tools.py::TestTLE::test_get_DUT
tests/test_TLE_tools.py::TestTLE::test_get_IERS_EOP
tests/test_TLE_tools.py::TestTLE::test_get_Polar_Motion
tests/test_TLE_tools.py::TestTLE::test_theta_GMST1982
tests/test_TLE_tools.py::TestTLE::test_tle_bstar
tests/test_TLE_tools.py::TestTLE::test_tle_id
tests/test_TLE_tools.py::TestTLE::test_tle_jd
tests/test_TLE_tools.py::TestTLE::test_yearday_to_monthday
tests/test_antenna.py::TestAntennaRX::test_ecef
tests/test_antenna.py::TestAntennaRX::test_init
tests/test_antenna.py::TestAntennaRX::test_str
tests/test_antenna.py::TestAntennaTX::test_get_scan
tests/test_antenna.py::TestAntennaTX::test_init
tests/test_antenna.py::TestGainConv::test_inst_gain2full_gain
tests/test_antenna.py::TestGainConv::test_inst_gain2full_gain_types
```

(continues on next page)

(continued from previous page)

```
tests/test_antenna.py::TestBeamPattern::test_gain
tests/test_antenna.py::TestBeamPattern::test_on_axis
tests/test_coord.py::TestCoord::test_angle_deg
tests/test_coord.py::TestCoord::test_az_el_r2geodetic
tests/test_coord.py::TestCoord::test_azel_to_cart
tests/test_coord.py::TestCoord::test_cart_to_azel
tests/test_coord.py::TestCoord::test_ecef2geodetic
tests/test_coord.py::TestCoord::test_ecef2local
tests/test_coord.py::TestCoord::test_ecef_geo_inverse
tests/test_coord.py::TestCoord::test_geodetic2ecef
tests/test_coord.py::TestCoord::test_geodetic_to_az_el_r
tests/test_coord.py::TestCoord::test_ned2ecef
tests/test_dpt.py::TestKeplerSolver::test_kepler_guess
tests/test_dpt.py::TestKeplerSolver::test_laguerre_solve_kepler
tests/test_dpt.py::TestAnomalies::test_eccentric2mean
tests/test_dpt.py::TestAnomalies::test_eccentric2true_coencides
tests/test_dpt.py::TestAnomalies::test_eccentric2true_hand_calc
tests/test_dpt.py::TestAnomalies::test_eccentric_true_inverse
tests/test_dpt.py::TestAnomalies::test_mean2eccentric
tests/test_dpt.py::TestAnomalies::test_mean2true
tests/test_dpt.py::TestAnomalies::test_mean_eccentric_inverse_array
tests/test_dpt.py::TestAnomalies::test_mean_eccentric_inverse_float
tests/test_dpt.py::TestAnomalies::test_mean_true_inverse_array
tests/test_dpt.py::TestAnomalies::test_mean_true_inverse_float
tests/test_dpt.py::TestAnomalies::test_true2eccentric_coencides
tests/test_dpt.py::TestAnomalies::test_true2eccentric_hand_calc
tests/test_dpt.py::TestAnomalies::test_true2mean
tests/test_dpt.py::TestTimes::test_date_to_jd_float
tests/test_dpt.py::TestTimes::test_date_to_jd_int
tests/test_dpt.py::TestTimes::test_gmst
tests/test_dpt.py::TestTimes::test_gmst_numpy
tests/test_dpt.py::TestTimes::test_jd_to_date_float
tests/test_dpt.py::TestTimes::test_jd_to_date_floor
tests/test_dpt.py::TestTimes::test_jd_to_date_int
tests/test_dpt.py::TestTimes::test_yearday_to_monthday
tests/test_dpt.py::TestOrbits::test_elliptic_radius
tests/test_dpt.py::TestOrbits::test_period_hand_calc
tests/test_dpt.py::TestOrbits::test_period_numpy
tests/test_dpt.py::TestOrbits::test_speed_hand_calc
tests/test_dpt.py::TestOrbits::test_speed_numpy
tests/test_dpt.py::TestKepCart::test_cart2kep_circ
tests/test_dpt.py::TestKepCart::test_cart_kep_inverse
tests/test_dpt.py::TestKepCart::test_kep2cart_Omega_inc
tests/test_dpt.py::TestKepCart::test_kep2cart_circ
tests/test_dpt.py::TestKepCart::test_kep2cart_ecc
tests/test_dpt.py::TestKepCart::test_kep2cart_inc
tests/test_dpt.py::TestKepCart::test_kep2cart_omega
tests/test_dpt.py::TestKepCart::test_kep2cart_omega_inc
tests/test_population.py::TestPopulation::test_allocate
tests/test_population.py::TestPopulation::test_column_order
tests/test_population.py::TestPopulation::test_constructor
tests/test_population.py::TestPopulation::test_constructor_arguemnts
tests/test_population.py::TestPopulation::test_filter
tests/test_population.py::TestPopulation::test_generator
tests/test_population.py::TestPopulation::test_get_item
tests/test_population.py::TestPopulation::test_get_item_nan
tests/test_population.py::TestPopulation::test_iter
```

(continues on next page)

(continued from previous page)

```
tests/test_population.py::TestPopulation::test_set_item
tests/test_population.py::TestPopulation::test_shape
tests/test_population.py::TestPopulation::test_space_object
tests/test_population_library.py::TestPopulationLibrary::test_filtered_master
tests/test_population_library.py::TestPopulationLibrary::test_master
tests/test_population_library.py::TestPopulationLibrary::test_master_factor
tests/test_population_library.py::TestPopulationLibrary::test_master_factor_cnt
tests/test_population_library.py::TestPopulationLibrary::test_tle_snapshot
tests/test_propagator_base.py::TestBaseProp::test_base_prop_methods
tests/test_propagator_base.py::TestBaseProp::test_meta_raise_no_method
tests/test_propagator_base.py::TestBaseProp::test_meta_raise_wrong_method
tests/test_propagator_base.py::TestBaseProp::test_numpy_conv_float
tests/test_propagator_base.py::TestBaseProp::test_numpy_conv_list
tests/test_propagator_base.py::TestBaseProp::test_numpy_conv_numpy
tests/test_propagator_base.py::TestBaseProp::test_numpy_conv_raise
tests/test_propagator_orekit.py::TestPropagatorOrekit::test_circ_orbit
tests/test_propagator_orekit.py::TestPropagatorOrekit::test_frame_conversion
tests/test_propagator_orekit.py::TestPropagatorOrekit::test_get_orbit_cart
tests/test_propagator_orekit.py::TestPropagatorOrekit::test_get_orbit_kep
tests/test_propagator_orekit.py::TestPropagatorOrekit::test_kep_cart
tests/test_propagator_orekit.py::TestPropagatorOrekit::test_options_drag_off
tests/test_propagator_orekit.py::TestPropagatorOrekit::test_options_frames
tests/test_propagator_orekit.py::TestPropagatorOrekit::test_options_gravity_kep
tests/test_propagator_orekit.py::TestPropagatorOrekit::test_options_gravity_order
tests/test_propagator_orekit.py::TestPropagatorOrekit::test_options_integrator
tests/test_propagator_orekit.py::TestPropagatorOrekit::test_options_jpliau
tests/test_propagator_orekit.py::TestPropagatorOrekit::test_options_more_solarsystem
tests/test_propagator_orekit.py::TestPropagatorOrekit::test_options_rad_off
tests/test_propagator_orekit.py::TestPropagatorOrekit::test_options_tidal
tests/test_propagator_orekit.py::TestPropagatorOrekit::test_options_tolerance
tests/test_propagator_orekit.py::TestPropagatorOrekit::test_raise_bodies
tests/test_propagator_orekit.py::TestPropagatorOrekit::test_raise_frame
tests/test_propagator_orekit.py::TestPropagatorOrekit::test_raise_models
tests/test_propagator_orekit.py::TestPropagatorOrekit::test_raise_sc_params_missing
tests/test_propagator_sgp4.py::TestPropagatorSGP4::test_PropagatorSGP4_cart
tests/test_propagator_sgp4.py::TestPropagatorSGP4::test_PropagatorSGP4_cart_kep_cases
tests/test_propagator_sgp4.py::TestPropagatorSGP4::test_PropagatorSGP4_cart_kep_
↪inverse
tests/test_propagator_sgp4.py::TestPropagatorSGP4::test_PropagatorSGP4_cart_kep_
↪inverse_cases
tests/test_propagator_sgp4.py::TestPropagatorSGP4::test_PropagatorSGP4_cart_polar_
↪motion00
tests/test_propagator_sgp4.py::TestPropagatorSGP4::test_PropagatorSGP4_get_orbit
tests/test_propagator_sgp4.py::TestPropagatorSGP4::test_PropagatorSGP4_get_orbit_B
tests/test_propagator_sgp4.py::TestPropagatorSGP4::test_PropagatorSGP4_get_orbit_cart
tests/test_propagator_sgp4.py::TestPropagatorSGP4::test_PropagatorSGP4_polar_motion
tests/test_propagator_sgp4.py::TestPropagatorSGP4::test_PropagatorSGP4_polar_motion00
tests/test_propagator_sgp4.py::TestPropagatorSGP4::test_class_implementation
tests/test_propagator_sgp4.py::TestPropagatorSGP4::test_ecef_teme_inverse
tests/test_propagator_sgp4.py::TestPropagatorSGP4::test_propagator_sgp4_mjd_
↪invaraiance
tests/test_radar_config.py::TestRadarSystem::test_init
tests/test_space_object.py::TestSpaceObject::test_kep_cart_init
tests/test_space_object.py::TestSpaceObject::test_propagator_change_orekit
tests/test_space_object.py::TestSpaceObject::test_propagator_change_sgp4
tests/test_space_object.py::TestSpaceObject::test_propagator_options_orekit
tests/test_space_object.py::TestSpaceObject::test_propagator_options_sgp4
```

(continues on next page)

(continued from previous page)

```
tests/test_space_object.py::TestSpaceObject::test_return_sizes
tests/test_space_object.py::TestSpaceObject::test_update_elements_cart
tests/test_space_object.py::TestSpaceObject::test_update_elements_kep
tests/test_space_object.py::TestSpaceObject::test_update_error

no tests ran in 1.19 seconds
```

7.6.2 Visual-tests

Visual tests produce some amount of plots that are inspected to make a rough estimate of validation.

Test propagator

These test files will take a propagator and produce 3 different orbit plots:

1. Circular orbit in equatorial place
2. Elliptic orbit in equatorial plane
3. Elliptic polar orbit in ECEF over several periods

Most scientists that work with orbits daily will have a good grasp of how these scenarios should look visually and can make early detection of errors by inspecting the output of the below code.

```python
import sys
import os
sys.path.insert(0, os.path.abspath('.'))

import unittest
import numpy as n
import numpy.testing as nt

import propagator_base

class new_propagator(propagator_base.PropagatorBase):
    def get_orbit(self, t, a, e, inc, raan, aop, mu0, mjd0, **kwargs):
        pass
    def get_orbit_cart(self, t, x, y, z, vx, vy, vz, mjd0, **kwargs):
        pass

class TestBaseProp(unittest.TestCase):

    def test_base_prop_methods(self):

        prop = new_propagator()

        assert prop.get_orbit(0,
            0,0,0,
            0,0,0,
            0,
        ) is None

        assert prop.get_orbit_cart(0,
            0,0,0,
            0,0,0,
```

(continues on next page)

(continued from previous page)

```
            0,
        ) is None

    def test_meta_raise_no_method(self):

        class new_wrong_propagator(propagator_base.PropagatorBase):
            pass

        self.assertRaises(TypeError, new_wrong_propagator)

    def test_meta_raise_wrong_method(self):

        class new_wrong_propagator(propagator_base.PropagatorBase):
            def get_orbit(self):
                pass
            def get_orbit_cart(self, t, x, y, z, vx, vy, vz, mjd0, **kwargs):
                pass

        self.assertRaises(AssertionError, new_wrong_propagator)


        class new_wrong_propagator(propagator_base.PropagatorBase):
            def get_orbit(self, x, a, e, inc, raan, aop, mu0, mjd0, **kwargs):
                pass
            def get_orbit_cart(self, t, x, y, z, vx, vy, vz, mjd0, **kwargs):
                pass

        self.assertRaises(AssertionError, new_wrong_propagator)


    def test_numpy_conv_float(self):

        prop = new_propagator()

        x = 5.3
        x_conv = prop._make_numpy(x)

        assert isinstance(x_conv, n.ndarray)

        nt.assert_almost_equal(x_conv[0], x, decimal=9)

    def test_numpy_conv_list(self):

        prop = new_propagator()

        x = [5.3]
        x_conv = prop._make_numpy(x)

        assert isinstance(x_conv, n.ndarray)

        nt.assert_almost_equal(x_conv[0], x[0], decimal=9)

    def test_numpy_conv_numpy(self):

        prop = new_propagator()

        x = n.array([5.3], dtype=n.float)
```

(continues on next page)

(continued from previous page)

```
        x_conv = prop._make_numpy(x)

        assert isinstance(x_conv, n.ndarray)

        assert x is x_conv

        nt.assert_array_almost_equal(x_conv, x, decimal=9)

    def test_numpy_conv_raise(self):

        prop = new_propagator()

        x = '4'
        self.assertRaises(Exception, prop._make_numpy, x)



if __name__ == '__main__':
    unittest.main(verbosity=2)
```

7.6.3 Simulation-tests

test

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX