EXOSIMS Documentation

Release 3.2.0

SIOSlab

Jan 10, 2024

CONTENTS

The open source exoplanet imaging mission simulator.

CHAPTER

ONE

CREDITS

EXOSIMS was created by Dmitry Savransky and initially written by Christian Delacroix, Daniel Garrett, Gabriel Soto, Dean Keithly, Corey Spohn Dmitry Savransky, with contributions by Rhonda Morgan, Michael Turmon, Patrick Lowrance, Walker Dula and Neil Zimmerman. For a complete list of contributors, see: https://github.com/dsavransky/EXOSIMS#credits-and-acknowledgements

EXOSIMS development has been supported by NASA Grant Nos. NNX14AD99G (GSFC), NNX15AJ67G (WPS) and NNG16PJ24C (SIT).

EXOSIMS makes use of Astropy, a community-developed core Python package for Astronomy (Astropy Collaboration, 2013).

EXOSIMS optionally makes use of Forecaster (http://ascl.net/1701.007).

EXOSIMS optionally makes use of NASA's Navigation and Ancillary Information Facility's SPICE system components (https://naif.jpl.nasa.gov/naif/).

EXOSIMS optionally uses values from: Mamjek, E. "A Modern Mean Dwarf Stellar Color and Effective Temperature Sequence", http://www.pas.rochester.edu/~emamajek/EEM_dwarf_UBVIJHK_colors_Teff.txt

For further information, please see EXOSIMS's ASCL page (http://ascl.net/1706.010) and the following papers:

- http://adsabs.harvard.edu/abs/2016JATIS...2a1006S
- http://adsabs.harvard.edu/abs/2016SPIE.9911E..19D

CHAPTER

TWO

CONTENTS

2.1 Introduction

Building confidence in a mission concept's ability to achieve its science goals is always desirable. Unfortunately, accurately modeling the science yield of an exoplanet imaging mission can be almost as complicated as designing the mission itself. It is challenging to compare science simulation results and systematically test the effects of changing one aspect of the instrument or mission design.

EXOSIMS addresses this problem by generating ensembles of mission simulations for exoplanet direct imaging missions to estimate science yields—what is known as Monte Carlo Mission Simulation (MCMS). It is designed to allow systematic exploration of exoplanet imaging mission science yields. It consists of stand-alone modules written in Python which may be modified without requiring modifications to other portions of the code. This allows EXOSIMS to be easily used to investigate new designs for instruments, observatories, or overall mission designs independently.

2.1.1 Purpose and Scope

The *MCMS* approach, and many of the algorithms used by EXOSIMS are extensively described in the academic literature (in particular in [Savransky2010] and [Savransky2015], but see also the rest of the *References*). This documentation is intended to provide an overview of the software framework of EXOSIMS and details on its component parts. As the software is intended to be highly reconfigurable, operational aspects of the code are emphasized, with a focus on required input/output interfaces for all pieces of the framework. Users wishing to implement their own EXOSIMS modules should read **all** of this. Users who only wish to use existing implementations to run simulations (or generate intermediate products like synthetic planets or keepout maps, etc.) can skip to *Quick Start Guide* (but should probably also read the *references* to understand the code's operational principles).

2.1.2 Framework

EXOSIMS is composed of 14 distinct object classes (called 'modules'):

- `BackgroundSources` Encodes information about background (astrophysical confusion) sources
- `Completeness` Provides methods for computing single-visit and dynamic completeness
- `Observatory` Encodes information about the observatory spacecraft (and any external occulter spacecraft) and provides methods for orbital propagation and fuel use computation
- `OpticalSystem` Encodes everything about the science instrument(s), starlight suppression system(s), and provides methods for calculating integration times
- `PlanetPhysicalModel` Provides models of planet physical attributes (e.g., density and albedo models)
- `PlanetPopulation` Encodes distributions of planet physical and orbital parameters and provides methods for sampling them

- *PostProcessing* Encodes effects of image post-processing
- *SimulatedUniverse* Provides methods for generating synthetic universes composed of real stars and fake planets (or mixtures of real and fake planets) and encodes exosystem information
- *StarCatalog* Input catalog of potential target stars
- *SurveySimulation* Provides methods for scheduling and simulating full observing programs
- *SurveyEnsemble* Provides methods for running ensembles of survey simulations
- *TargetList* Provides methods for filtering input star catalogs into final target lists and stores all required star information for the final target list.
- *TimeKeeping* Encodes information about mission time
- *ZodiacalLight* Information about local and exo-zodiacal light

EXOSIMS provides a reference implementation (called 'prototypes') for each of these 14 modules, along with multiple additional implementations of most module types. Different implementations of the modules contain specific mission design parameters and physical descriptions of the universe, and will change according to the mission and planet population of interest. The prototype implementations (and especially their docstrings) provide the input/outptut specification (interface control) for the framework. Every module implementation **must** inherit a prototype module, and any method overloading a method defined in a prototype implementation **must** conform to the inputs/outputs of the prototype method.

In addition to the 14 modules, there is a top-level *MissionSim* class, used to instantiate objects of all 14 module types required to run a full simulation ensemble. The *MissionSim* works with an input specification file (see *Input Specification*) to set input parameters throughout all of the modules.

The overall framework of EXOSIMS is depicted in Fig. 2.1, which shows all of the component software modules in the order in which they are instantiated. Instantiating a *MissionSim* object will automatically instantiate 14 objects (accessible as attributes of the *MissionSim*).

Objects of all module classes can be instantiated independently, although most modules require the instantiation of other modules during their initialization—generating an object of any type will also generate objects of all downstream module classes, as depicted in Fig. 2.1. The upstream modules (including *TargetList*, *SimulatedUniverse*, *SurveySimulation*, and *SurveyEnsemble*) use attributes and methods from downstream modules and perform mission simulation tasks. Any module may perform any number or kind of calculations using any or all of the input parameters provided to the full framework. The specific implementations are only constrained by their input and output specification, as encoded by the prototypes. When creating new implementations, it can always be assumed that a module of a given type has access to all downstream module objects at runtime.

2.1.3 Terminology

The terminology used to describe the EXOSIMS software is loosely based upon object-oriented programing (OOP) terminology, and, in particular on the Python language and conventions. The term module can refer to the object class prototype representing the abstracted functionality of one piece of the software, an implementation of this object class which inherits the attributes and methods of the prototype, or an instance of this class. Input/output definitions of modules refer to the class prototype. Implemented modules refer to the inherited class definition. Passing modules (or their outputs) means the instantiation of the inherited object class being used in a given simulation. Relying on strict inheritance for all implemented module classes provides an automated error and consistency-checking mechanism. The outputs of a given object instance may be compared to the outputs of the prototype. It is trivial to pre-check whether a given module implementation will work within the larger framework, and this approach allows for flexibility and adaptability.

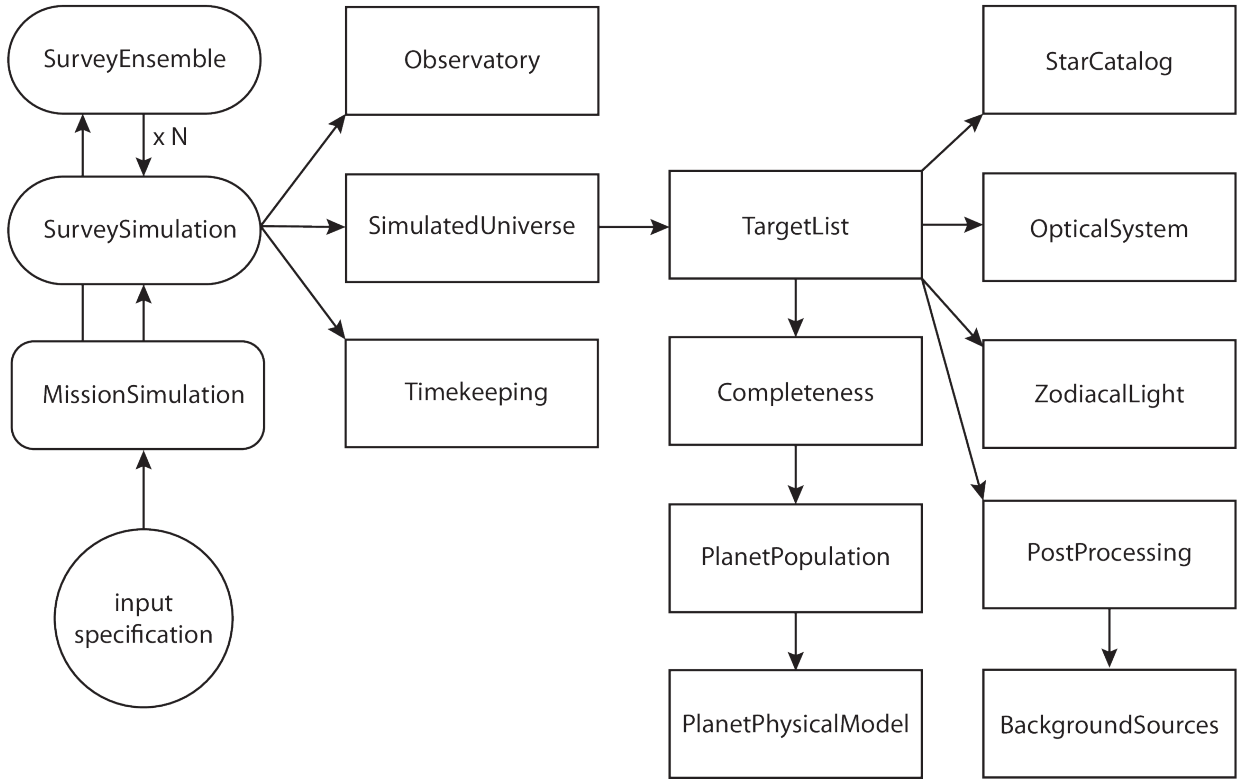


Fig. 2.1: Schematic depiction of the instantiation order of all EXOSIMS modules. The arrows indicate calls to the object constructor, and object references to each module are always passed up directly to the top calling module, so that a given module has access to any other module connected to it by a direct path of instantiations. For example, the *TargetList* module has access to both the *PostProcessing* and *BackgroundSources* modules, while the *Observatory* module does not have access to any other modules. The typical entry point to EXOSIMS is the construction of a *MissionSim* object, which causes the instantiation of the *SurveySimulation* module, which in turn instnatiates all the other modules. In the case of a parallelized *SurveyEnsemble* instnatiation, multiple, independent *SurveySimulation* modules are instantiated at the same time. At the end of construction, the *MissionSim* and *SurveySimulation* objects have direct access to all other modules as their attributes.

2.1.4 Directory Layout

The EXOSIMS repository directory structure is:

```
EXOSIMS/
|
├── EXOSIMS/
|   ├── BackgroundSources/
|   ├── Completeness/
|   ├── Observatory
|   ├── OpticalSystem
|   ├── PlanetPhysicalModel
|   ├── PlanetPopulation
|   ├── PostProcessing
|   ├── Prototypes
|   ├── Scripts
|   ├── SimulatedUniverse
|   ├── StarCatalog
|   ├── SurveyEnsemble
|   ├── SurveySimulation
|   ├── TargetList
|   ├── TimeKeeping
|   ├── ZodiacalLight
|   ├── run
|   └── util
|
├── tests/
|   ├── BackgroundSources/
|   ├── Completeness/
|   ├── Observatory
|   ├── OpticalSystem
|   ├── PlanetPhysicalModel
|   ├── PlanetPopulation
|   ├── PostProcessing
|   ├── Prototypes
|   ├── SimulatedUniverse
|   ├── StarCatalog
|   ├── SurveyEnsemble
|   ├── SurveySimulation
|   ├── TargetList
|   ├── TestModules
|   ├── TestSupport
|   ├── TimeKeeping
|   ├── ZodiacalLight
|   └── util
|
├── documentation/
```

The top-level EXOSIMS directory (the one containing all subfolders and the `setup.py` file) is referred to as the EXOSIMS root directory. The code resides in the `EXOSIMS` sub-folder, defining the top-level *EXOSIMS* package, which has 16 sub-packages. The *Prototypes* sub-package contains all of the prototype module implementations as sub-modules. Additional implementations of each module type are sub-modules to sub-packages of the same name as the module type (i.e., all Completeness implementations other than the prototype are submodules of *Completeness*, etc.). The 16th sub-package is *util* and contains various utilities used by the EXOSIMS modules as well as standalone analysis

and plotting tools. The EXOSIMS top-level directory also contains two additiona folders that are not sub-packages of the code. The `Scripts` directory contains sample and template *input specification files*. The `run` directory contains methods for parallel code execution (see *SurveyEnsemble* for more details).

The `tests` directory mirrors the layout of the EXOSIMS package, with additional folders for test support code. Further details are provided in *Testing*.

2.2 Installing and Configuring

2.2.1 Obtaining EXOSIMS

EXOSIMS is hosted on github at https://github.com/dsavransky/EXOSIMS. The master branch is a development branch and may be frequently changed, including ICD-breaking changes (however it should always pass all unit and end-to-end tests). For publishable/reproducible results, we strongly recommend using the latest tagged release from:

https://github.com/dsavransky/EXOSIMS/releases

2.2.2 Environment and Package Dependencies

EXOSIMS requires Python 3.7+ and a number of packages hosted on the PyPI. For a full list of current package dependencies, see the `requirements.txt` file in the top level of the repository.

2.2.3 Installation

If you wish to install EXOSIMS to try it out, or to run existing code, then just grab the current stable version from PyPI:

```
pip install EXOSIMS
```

If, however, you are planning on developing your own module implementations, then it is recommended that you install in developer mode. Clone (or download) a copy of the repository and in the top level folder (the one containing `setup.py` run:

```
pip install -e .
```

Note that this is the install mode used for all automated unit testing. Sere here for more details: https://pip.pypa.io/en/stable/reference/pip_install/#editable-installs

---

Note: Installing EXOSIMS via pip will automatically install all required and optional packages, except for cython. If cython and numpy are already present on the system, the cythonized components of EXOSIMS will be compiled automatically, otherwise they will be ignored. Note that this does not represent any loss of functionality, as these components are drop-in replacements for native python implementations. The only difference in execution will be an increase in total run time.

2.2.4 Cache Directory

EXOSIMS generates a large number of cached data products during run time. These are stored in the EXOSIMS cache, which can be controlled via environment variables or on a script-by-script basis.

On POSIX systems, the default cached directory is given by `/home/user/.EXOSIMS/cache`. On Windows systems, the default cache directory is typically like `C:/Users/User/.EXOSIMS/cache`. For details on how the home directory is determined, see method `get_home_dir` in `util/get_dirs.py`. If `cachedir` is specified in the input json script, the cache directory will be `cachedir` (this path may include environment variables and other expandable elements). Alternatively, the cache directory may be specified by setting environment variable `EXOSIMS_CACHE_DIR`.

The order of precedence for determining the cache directory is:

1. JSON input
2. Environment variable
3. Default path

2.2.5 Downloads Directory

The downloads directory is where files from outside EXOSIMS are stored (this includes SPK files used by jplephem, Forecaster fitting parameters, etc.). On POSIX systems, the downloads directory is given by `/home/user/.EXOSIMS/downloads`. On Windows systems, the downloads directory is typically like `C:/Users/User/.EXOSIMS/downloads`. Just like the *Cache Directory*, the downloads directory may be set via the JSON script (input `downloadsdir`) or via environment variable `EXOSIMS_DOWNLOADS_DIR`. The order of precedence in selecting the directory to use at runtime is the same as for the cache.

2.2.6 Compiling Cython Modules

> **Note:** Installing EXOSIMS via pip will automatically compile all of these components if cython *and* numpy are already installed on the system. You only need to preform this procedure if installing manually, or if you decide to add cython after initial installation. Note, however, that it is still preferable to just rerun the pip installation procedure after installing the cython package.

To speed up execution, some EXOSIMS components are implemented as both regular interpreted python and as statically compiled executables via Cython. The code is set to automatically use the compiled versions if they are available, and these (currently) must be manually compiled on each system where the code is installed. In all cases, compilation is done by executing a python setup script. The individual components with Cython implementations are listed below.

2.2.6.1 KeplerSTM

The `KeplerSTM` utility is responsible for orbital propagation in `EXOSIMS`. It has a Cython implementation: `CyKeplerSTM`, which wraps a pure C implementation of the propagation algorithms, called `KeplerSTM_C`. To compile the Cython implementation, navigate to `EXOSIMSROOT/EXOSIMS/util/KeplerSTM_C`. Execute:

```
> python CyKeplerSTM_setup.py build_ext --inplace
```

This will generate a `.c` file and compile to a `.so` file on MacOS/Linux or a `.pyd` file on Windows. The python `KeplerSTM` automatically loads the compiled module if it is present, and uses it by default if successfully loaded.

2.3 Quick Start Guide

This is intended as a very brief overview of the steps necessary to get EXOSIMS running. To start writing your own modules, please refer to the *Introduction* and the rest of this documentation. Before using this guide, read through the *Installing and Configuring* guide and follow all of the setup steps.

2.3.1 EXOSIMS Workflow

2.3.1.1 Creating a MissionSimulation

The default entry point to all EXOSIMS is via the instantiation of a *MissionSim* object, which is created via an *Input Specification* specifying the modules to be used and setting various input parameters. The instantiation of this object, in turn, causes the instantiation of all other module objects. Here is a quick example using a built-in sample script:

```
import EXOSIMS,EXOSIMS.MissionSim,os.path
scriptfile = os.path.join(EXOSIMS.__path__[0],'Scripts','sampleScript_coron.json')
sim = EXOSIMS.MissionSim.MissionSim(scriptfile)
```

The first time you run this code (or any new combination of modules/parameters), there will be a long series of calculations of various useful values. All of these are cached to disk, however (see: *Cache Directory*), which means that subsequent construction of any object using the same module and inputs will be significantly sped up.

Once instantiated, *MissionSim* object contains directly accessible instances of all modules (i.e., `sim.Observatory`) as well as a dictionary of the modules (`sim.modules`). At this point you can interrogate the `sim` object to see the results of the setup. In particular, you can check the size of your initial target list (`sim.TargetList.nStars`) as well as their initial (single-visit) *completeness* values (`sim.TargetList.int_comp`).

Warning: The sampleScript_coron.json script uses the prototype Completeness module, which does not actually do any completeness calculations, but simply returns the same value for all stars. To calculate completeness you must use one of the implementations such as *BrownCompleteness* or *GarrettCompleteness*.

You can also get a full list of all parameters (this includes the ones that weren't specified in your input script and were filled in by defaults) via the *genOutSpec()* method (called, in this case, as `sim.genOutSpec`), which returns a dictionary of all simulation parameters. Setting the `tofile` keyword to this method will also write this dictionary out to the specified path.

Note: The python JSON writer supports reading/writing values (such as infinity and nan) that are not in the JSON specification. This means that output script files may only be parseable by python, or another parser supporting these extensions to the specification.

2.3.1.2 Running a Simulation and Analyzing Results

The survey simulation is executed via the *run_sim()* method. When running with default settings (as in `sampleScript_coron.json`) the simulation details (observation numbers and detections/characterizations) will be printed as it is executed (this can be toggled off via the `verbose` script keyword). The full mission timeline is saved to the `DRM` variable in the `SurveySimulation` object, and can be accessed as:

```
sim.run_sim()
DRM = sim.SurveySimulation.DRM
```

The `DRM` is a list of dictionaries, each representing one observations, so that a mission simulation of 10 observations will produce a `DRM` of length 10. The dictionaries in `DRM` contain all of the details on each observation. You can look at a full list of dictionary keywords by executing `sim.SurveySimulation.DRM[0].keys()`. Of particular importance are:

- star_ind - The index of star observed. You can get information about the star by using this index with the TargetList object. For example `sim.TargetList.Name[sim.SurveySimulation.DRM[0]['star_ind']]` will return the name of the first star observed, and `sim.TargetList.coords[sim.SurveySimulation.DRM[0]['star_ind']]` will return its coordinates.
- plan_inds - The indices of all planets belonging to this star. You can get information about these planets by using this index with the SimulatedUniverse object. For example `sim.SimulatedUniverse.a[sim.SurveySimulation.DRM[0]['plan_inds']]` returns the semi-major axes of all planets in the system observed in the fist observation, and `sim.SimulatedUniverse.Rp[sim.SurveySimulation.DRM[0]['plan_inds']]` will return their radii.
- det_status - This encodes the outcome of the observation for each planet. 0 represents a missed detection, 1 represents a detection, and -1, -2 represent the planet being inside the IWA and outside the OWA, respectively. `sim.SurveySimulation.DRM[0]['plan_inds'][sim.SurveySimulation.DRM[0]['det_status'] == 1]` will return the indices of all planets found in the first observation.

To find the number of stars observed during my mission that have at least 1 planet detected, we could run:

```
len([DRM[x]['star_ind'] for x in range(len(DRM)) if 1 in DRM[x]['det_status']])
```

`MissionSim` also provides utilities for examining the `DRM`. The `DRM2array` method will return an array of all of the `DRM` entries for a specified key for the full `DRM`. So, running `sim.DRM2array('plan_inds')` will return and array of arrays of all the planet indices encountered (but not necessarily detected) during the full mission. `numpy.hstack(sim.DRM2array('plan_inds'))` will flatten this array into a 1D list of all planet indices encountered.

The `filter_status` method will filter a provided key with a given status code. `sim.filter_status('plan_inds',0)` will return all planet indices with missed detection throughout the full mission and `sim.filter_status('plan_inds',1)` will return the indices of all detected planets.

2.3.1.3 Running Additional Simulations

To run a new simulation using the same input scriptfile, simply reset the simulation and run it again. You can choose to generate new planets or to rewind the positions of the current set of planets to their initial states. Setting both of these keywords to `False` will result in running a simulation that starts with all planets in their final states from the previous simulation.

```
sim.reset_sim(genNewPlanets=True, rewindPlanets=True)
sim.run_sim()
```

You can also run an ensemble of N simulations, which produces a list of DRMs. From there, you can find e.g. the number of observations made during each survey.

```
sim.reset_sim()
N = 100
ens = sim.run_ensemble(N, genNewPlanets=True, rewindPlanets=True)
nb_obs = []
for i in range(N):
    DRM = ens[i]
    nb_obs.append(len(DRM))
```

The default ensemble will run in sequence. For more details on ensembles and parallelization see *SurveyEnsemble*.

2.3.2 Building Your Own Mission

This is a brief guide to iteratively building up a simulation script, with comments and sanity checks along the way. It touches on only a subset of all possible user settings for the base modules. A more complete list is available here: *EXOSIMS Prototype Inputs*.

2.3.2.1 Step 1

The only required components of the input specification are:

- The modules dictionary
- The science instruments list
- The starlight suppression systems list.

All other values will be filled in with defaults, although this will typically not produce a reasonable mission description, depending on the modules selected. We begin with an empty set of modules, which would load all of the prototypes, and a single instrument and starlight suppression system, which will define the default observing mode. In a directory of your choosing (preferably outside of the EXOSIMS repository), create a file called `test.json` with the following contents:

```
{
 "modules": {
 "PlanetPopulation": " ",
 "StarCatalog": " ",
 "OpticalSystem": " ",
 "ZodiacalLight": " ",
 "BackgroundSources": " ",
 "PlanetPhysicalModel": " ",
 "Observatory": " ",
 "TimeKeeping": " ",
 "PostProcessing": " ",
 "Completeness": " ",
 "TargetList": " ",
 "SimulatedUniverse": " ",
 "SurveySimulation": " ",
 "SurveyEnsemble": " "
 },
 "scienceInstruments": [
 { "name": "imager" }
 ],
 "starlightSuppressionSystems": [
```

(continues on next page)

(continued from previous page)

```
 {  "name": "coronagraph" }
 ]
}
```

You can create a `MissionSim` object with this script, but it won't be particularly useful, since there are no real stars in the prototype `StarCatalog`. We'll do it anyway to sanity check that the code is working. In a python interpreter running in the same directory as your test script run:

```
import EXOSIMS.MissionSim
sim = EXOSIMS.MissionSim.MissionSim('test.json')
```

You should see outputs showing the modules being loaded as the simulation object is instantiated, along the lines of

```
Imported SurveyEnsemble (prototype module) from EXOSIMS.Prototypes.SurveyEnsemble
Imported SurveySimulation (prototype module) from EXOSIMS.Prototypes.SurveySimulation
Imported SimulatedUniverse (prototype module) from EXOSIMS.Prototypes.SimulatedUniverse
Imported TargetList (prototype module) from EXOSIMS.Prototypes.TargetList
Imported StarCatalog (prototype module) from EXOSIMS.Prototypes.StarCatalog
Imported OpticalSystem (prototype module) from EXOSIMS.Prototypes.OpticalSystem
Imported ZodiacalLight (prototype module) from EXOSIMS.Prototypes.ZodiacalLight
Imported PostProcessing (prototype module) from EXOSIMS.Prototypes.PostProcessing
Imported BackgroundSources (prototype module) from EXOSIMS.Prototypes.BackgroundSources
Imported Completeness (prototype module) from EXOSIMS.Prototypes.Completeness
Imported PlanetPopulation (prototype module) from EXOSIMS.Prototypes.PlanetPopulation
Imported PlanetPhysicalModel (prototype module) from EXOSIMS.Prototypes.
↪PlanetPhysicalModel
Imported Observatory (prototype module) from EXOSIMS.Prototypes.Observatory
Imported TimeKeeping (prototype module) from EXOSIMS.Prototypes.TimeKeeping
Numpy random seed is: 491873991
```

Printing the contents of `sim.TargetList.nStars` and `sim.SimulatedUniverse.plan2star` will show that this simulation has one (fake) star with one simulated planet (`plan2star` is an array of indices mapping planet attributes to stars - in this case it is a single element array mapping to star 0). This planet is generated with properties that ensure that it is detectable with all of the default settings in the other modules.

2.3.2.2 Step 2

Now we must decide what kind of universe we will be modeling. Let's select the EXOCAT-1 input catalog (http://nexsci.caltech.edu/missions/EXEP/EXEPstarlist.html), provided by the `EXOCAT1 StarCatalog` implementation and only model Earth-twins in the habitable zone. We have two suitable `PlanetPopulation` implementations - `EarthTwinHabZone1` and `EarthTwinHabZone2`, but we would like to override the defaults and only consider eccentricities between 0 and 0.35 so we will use `EarthTwinHabZone2` (`EarthTwinHabZone1` does not allow for overriding orbital parameters). Our JSON script now becomes:

```
{
 "modules": {
 "PlanetPopulation": "EarthTwinHabZone2",
 "StarCatalog": "EXOCAT1",
 "OpticalSystem": " ",
 "ZodiacalLight": " ",
 "BackgroundSources": " ",
 "PlanetPhysicalModel": " ",
```

(continues on next page)

(continued from previous page)

```
 "Observatory": " ",
 "TimeKeeping": " ",
 "PostProcessing": " ",
 "Completeness": " ",
 "TargetList": " ",
 "SimulatedUniverse": " ",
 "SurveySimulation": " ",
 "SurveyEnsemble": " "
 },
 "scienceInstruments": [
 { "name": "imager" }
 ],
 "starlightSuppressionSystems": [
 {  "name": "coronagraph" }
 ],
 "erange": [0, 0.3]
}
```

We again build a `MissionSim` object called `sim` using this script and then verify that our `erange` has overwritten the default by looking at the contents of `sim.PlanetPopulation.erange` and by printing `sim.SimulatedUniverse.e.min(), sim.SimulatedUniverse.e.max()`. The former shows us the range used in sampling by the `PlanetPopulation` while the latter shows the range of values actually sampled when creating the simulated universe.

Another important thing to note is that the `EarthTwinHabZone2` populations set the `constrainOrbits` keyword to `True` by default. This flag forces all orbital radii to be within the semi-major axis range (so that $a(1+e) \leq a_{\max}$ and $a(1-e) \geq a_{\min}$). At the same time, the `EarthTwinHabZone` implementations also set the `scaleOrbits` flag to `True`, which causes the semi-major axes to be scaled by the square root of the stellar luminosities as they are generated in the `SimulatedUniverse`. To verify that these things are happening we can execute the following:

```
import numpy as np
Ls = sim.TargetList.L[sim.SimulatedUniverse.plan2star]
smas = sim.SimulatedUniverse.a/np.sqrt(Ls)
print(np.all((smas <= sim.PlanetPopulation.arange[1]) & (smas >= sim.PlanetPopulation.
↪arange[0])))
print(np.all((smas*(1+sim.SimulatedUniverse.e) <= sim.PlanetPopulation.arange[1]) &␣
↪(smas*(1-sim.SimulatedUniverse.e) >= sim.PlanetPopulation.arange[0])))
```

The `plan2star` attribute maps the simulated planets to their parent stars in the target list object, allowing us to extract the stellar luminosities. Both of the logical tests should evaluate to `True` (both the semi-major axes and extrema of the orbital radii should fall within the semi-major axis range with the default flags).

Another thing to test is that we are generating the proper number of planets. In this population, this is controlled by the `eta` parameter (also settable in the JSON script), which defaults to 0.1, meaning that we expect one planet per ten stars, on average. As these are generated probabilistically, we will not have an exact occurrence rate of 0.1 in any given simulation, but over many simulations, we should expect to average to this rate. We can explicitly test this by executing the following:

```
rate = 0
for j in range(100):
    rate += float(len(sim.SimulatedUniverse.plan2star))/sim.TargetList.nStars
    sim.reset_sim()

print(rate/100.0)
```

The rate should be very nearly 0.1 (with standard Poisson error).

At this point, we should have a large number of stars in our target list (verify by printing `sim.TargetList.nStars`) because the prototype Completeness isn't calculating the true completeness, and the default instrument settings will result in very low integration times for most stars, meaning that they won't be filtered out based on your integration time cutoff, encoded in `sim.OpticalSystem.intCutoff` with a default value of 50 days, and also settable as `intCutoff` in the JSON script. The filtering works by calculating the minimum necessary integration time (with no zodiacal light contribution) for a planet of `sim.OpticalSystem.dMag0` at a working angle of `sim.OpticalSystem.WA0` (both of these also settable in the JSON script as `dMag0` and `WA0`, respectively. The default `dMag0` is 15 (10^{-6} contrast), meaning that the vast majority of targets are retained.

2.3.2.3 Step 3

Now we can describe the actual instrument. We wish to model a 4 meter diameter, unobscured primary. Our coronagraph will have an inner working angle of 100 mas and an outer working angle of 1 arcsecond, with a constant contrast of 10^{-11}. We will assume a modest post-processing factor of 0.1 (meaning that we can reduce residual speckle noise by one order of magnitude via post-processing). The JSON script now looks like this:

```
{
 "modules": {
 "PlanetPopulation": "EarthTwinHabZone2",
 "StarCatalog": "EXOCAT1",
 "OpticalSystem": " ",
 "ZodiacalLight": " ",
 "BackgroundSources": " ",
 "PlanetPhysicalModel": " ",
 "Observatory": " ",
 "TimeKeeping": " ",
 "PostProcessing": " ",
 "Completeness": " ",
 "TargetList": " ",
 "SimulatedUniverse": " ",
 "SurveySimulation": " ",
 "SurveyEnsemble": " "
 },
 "scienceInstruments": [
 { "name": "imager" }
 ],
 "starlightSuppressionSystems": [
 {  "name": "coronagraph",
    "IWA": 0.1,
    "OWA": 1.0,
    "core_contrast": 1.0e-11
 }
 ],
 "erange": [0, 0.3],
 "pupilDiam": 4.0,
 "obscurFac": 0.0,
 "ppFact": 0.1
}
```

We again build a `MissionSim` object called `sim` using the updated script and check that our changes have been applied. Running:

```
sim.OpticalSystem.starlightSuppressionSystems[0]['core_contrast'](sim.OpticalSystem.
↪starlightSuppressionSystems[0]['lam'],sim.OpticalSystem.starlightSuppressionSystems[0][
↪'IWA'])
```

evaluates the contrast at the coronagraph central wavelength and inner working angle and should return our input constant contrast. Running:

```
sim.OpticalSystem.pupilDiam**2.*sim.OpticalSystem.shapeFac - sim.OpticalSystem.pupilArea
```

should return zero, verifying that the aperture is unobscured. `shapeFac` is another user-settable parameter, and is defined such that its product with the square of the aperture diameter gives the pupil area (it defaults to the value for circular apertures).

Looking at `sim.TargetList.nStars`, we see that our target list is now significantly smaller than it was before. This is directly a consequence of setting an inner and outer working angle for our coronagraph (the default values are zero to infinity). Due to the limited nature of the selected planet population, and finite IWA/OWA instantly filters out the majority of stars, for which the entire planet population would fall outside of this coronagraph's operating angular separation range.

2.3.2.4 Step 4

We will now replace the remaining prototype modules which don't perform the specific calculations and only return dummy values with full implementations. We will use:

- The Nemati `OpticalSystem` (integration time calculations are based on the equations found in [Nemati2014])
- The Brown `Completeness` (this is the Monte-Carlo version of the calculation, based on [Brown2005]; alternatively, we have `GarrettCompletness` which is a fully analytical implementation based on [Garrett2016])
- The Stark `ZodiacalLight` module (the local zodi is based on modeling from [Stark2014])
- The Forecaster `PlanetPhysicalModel` implementation (this uses Forecaster [Chen2016] to probabilistically calculate planet densities)

Our JSON script now looks as follows:

```
{
 "modules": {
 "PlanetPopulation": "EarthTwinHabZone2",
 "StarCatalog": "EXOCAT1",
 "OpticalSystem": "Nemati",
 "ZodiacalLight": "Stark",
 "BackgroundSources": " ",
 "PlanetPhysicalModel": "Forecaster",
 "Observatory": " ",
 "TimeKeeping": " ",
 "PostProcessing": " ",
 "Completeness": "BrownCompleteness",
 "TargetList": " ",
 "SimulatedUniverse": " ",
 "SurveySimulation": " ",
 "SurveyEnsemble": " "
 },
 "scienceInstruments": [
 { "name": "imager" }
```

(continues on next page)

(continued from previous page)

```
 ],
 "starlightSuppressionSystems": [
 {  "name": "coronagraph",
    "IWA": 0.1,
    "OWA": 1.0,
    "core_contrast": 1.0e-11
 }
 ],
 "erange": [0, 0.3],
 "pupilDiam": 4.0,
 "obscurFac": 0.0,
 "ppFact": 0.1
}
```

Building the `sim` object will now take considerably longer as the Monte Carlo completeness calculation executes (and the output will include status messages regarding this calculation). Note that this will only happen once per script, as the completeness is cached on disk. Looking at the new TargetList, we see that it has relatively few targets. This is due to the completeness filtering. This is controlled by two parameters: `minComp` and `dMagLim`. The former sets the cutoff below which targets are discarded, and the second sets the limiting Δmag of the dimmest planets of interest (the effective instrumental contrast floor used in the completeness calculation). The default values for these parameters (which can be confirmed either from the code, or by generating an outSpec dictionary, or by querying the parameters in the `sim.Completeness` object) are 0.1 and 25, respectively. Given that the population of Earth twins is typically dimmer than 25, these settings lead to relatively low completeness values.

If we wish to expand our initial target list, we can change `dMagLim` or `minComp` (or both). It is important to note that the `dMagLim` parameter value serves as the default for the `int_dMag` parameter in the `SurveySimulation` module, which (in the prototype implementation) sets the target planet magnitude used in determining integration times for each target. Increasing `dMagLim` without changing `dMagInt` will therefore cause integration times to grow, and may potentially waste a lot of mission time. We therefore allow for independent setting of these two parameters. However, once you select a `dMagInt` that is different from the `dMagLim`, you explicitly decouple the completeness from the execution of the survey (this is not a large consideration, as the two are always fundamentally different, but is important to remember when interpreting results).

2.3.2.5 Step 5

Finally, we will fill in a few more mission details. We will make this a five year mission with one year of integration time dedicated to planet finding. We also wish to only perform detections, and not spend any time on spectral characterizations. This is achieved by setting the SNR to zero in the characterization observing mode. Right now, there is only one observing mode that is automatically generated from the single instrument and starlight suppression system (stored in `sim.OpticalSystem.observingModes`), so we will have to define a dummy spectrometer instrument and two modes - one for detection and one for characterization. Our JSON script now looks like this:

```
{
 "modules": {
 "PlanetPopulation": "EarthTwinHabZone2",
 "StarCatalog": "EXOCAT1",
 "OpticalSystem": "Nemati",
 "ZodiacalLight": "Stark",
 "BackgroundSources": " ",
 "PlanetPhysicalModel": "Forecaster",
 "Observatory": " ",
 "TimeKeeping": " ",
```

(continues on next page)

(continued from previous page)

```
 "PostProcessing": " ",
 "Completeness": "BrownCompleteness",
 "TargetList": " ",
 "SimulatedUniverse": " ",
 "SurveySimulation": " ",
 "SurveyEnsemble": " "
 },
 "scienceInstruments": [
 { "name": "imager" },
 { "name": "spectrometer" }
 ],
 "starlightSuppressionSystems": [
 {  "name": "coronagraph",
    "IWA": 0.1,
    "OWA": 1.0,
    "core_contrast": 1.0e-11
 }
 ],
 "erange": [0, 0.3],
 "pupilDiam": 4.0,
 "obscurFac": 0.0,
 "ppFact": 0.1,
 "observingModes": [
    { "instName": "imager",
      "systName": "coronagraph",
      "detectionMode": true,
      "SNR": 5
    },
    { "instName": "spectrometer",
      "systName": "coronagraph",
      "SNR": 0
    }
 ],
 "minComp": 0.01,
 "dMagLim": 26,
 "missionLife": 5,
 "missionPortion": 0.2
}
```

After creating a new `sim` object with this script, we are now ready to run our simulation. We execute `sim.run_sim()` and the simulation progress is printed as it runs, terminating somewhere near 1826.25 days (the actual mission end time will depend on the specific observations scheduled).

Note: It is possible for the mission end time to be greater than the mission lifetime as observations are not interrupted if they extend past the end of the nominal mission life. However, no new observations will be scheduled after this point.

We can now use the same tools as described in *Running a Simulation and Analyzing Results* to analyze the results.

2.3.3 Creating Synthetic Universes

In some instances, you may wish to use EXOSIMS's synthetic universe generation capabilities without wanting to set up a full mission simulation (and all of the overhead that goes with it). You can do so by directly instantiating a `SimulatedUniverse` object. This requires only a subset of modules to be instantiated, namely:

1. TargetList
2. StarCatalog
3. PlanetPopulation
4. PlanetPhysicalModel
5. OpticalSystem
6. ZodiacalLight
7. BackgroundSources
8. PostProcessing
9. Completeness
10. SimulatedUniverse

While you probably don't care about several of these, they are needed to build the TargetList, and you can just specify their Prototype implementations. In particular, the prototype Completeness implementation returns values of 0.2 for every target, and so can be used to retain all targets regardless of their actual completeness values under your selected planet population. You can create a JSON script as in *Building Your Own Mission*, and then read it in like so:

```
import json
with open(scriptfile) as ff:
     specs = json.loads(ff.read())
```

or, alternatively, just define a specs dictionary in your python session. For example, if we wanted to build a Kepler-like simulated universe based on the EXOCAT-1 catalog, then a minimal specification would look like this:

```
specs = {"modules": {
      "PlanetPopulation": "KeplerLike2",
      "StarCatalog": "EXOCAT1",
      "OpticalSystem": "Nemati",
      "ZodiacalLight": "Stark",
      "BackgroundSources": " ",
      "PlanetPhysicalModel": "FortneyMarleyCahoyMix1",
      "PostProcessing": " ",
      "Completeness": " ",
      "TargetList": " ",
      "SimulatedUniverse": "KeplerLikeUniverse" },
      "scienceInstruments": [{ "name": "imager"}],
      "starlightSuppressionSystems": [{ "name": "coronagraph"}],
      "explainFiltering": True}
```

The `explainFiltering` key will cause EXOSIMS to print out how the target list is being filtered based on the other modules. You can control this behavior by setting other inputs, as described in the documentation for individual modules. Once the specs dictionary is defined, you can instantiate your Simulated Universe as:

```
import EXOSIMS.SimulatedUniverse.KeplerLikeUniverse
SU = EXOSIMS.SimulatedUniverse.KeplerLikeUniverse.KeplerLikeUniverse(**specs)
```

Warning: The instantiation of this object will modify the `specs` dictionary in such a way that you will not be able to instantiate another instance from it. If you wish to preserve its form, make a copy (not assignment) of `specs` prior to running the above code.

You can now interact with the SU object as usual. All of the planet properties are stored as numpy arrays as documented in the SimulatedUniverse docstrings and the ICD.

2.3.4 Generating Keepout Map Data

This is a set of instructions to generating the keepout map for a single star system. We use the following json `spec` input to instantiate the mission simulation object.

```
{
  "koAngles_SolarPanel":[56.0,124.0],
  "missionLife": 3,
  "checkKeepoutEnd": true,
  "pupilDiam": 2.37,
  "scienceInstruments": [
    { "name": "imager"
    }
  ],
  "starlightSuppressionSystems": [
    { "name": "HLC-565",
      "koAngles_Sun":[45.0,180.0],
      "koAngles_Earth":[45.0,180.0],
      "koAngles_Moon":[45.0,180.0],
      "koAngles_Small":[1.0,180.0]
    }
  ],
  "observingModes": [
    { "instName": "imager",
      "systName": "HLC-565",
      "detectionMode": true,
      "SNR": 5
    }
  ],
  "modules": {
    "PlanetPopulation": " ",
    "StarCatalog": "EXOCAT1",
    "OpticalSystem": " ",
    "ZodiacalLight": " ",
    "BackgroundSources": " ",
    "PlanetPhysicalModel": " ",
    "Observatory": "WFIRSTObservatoryL2",
    "TimeKeeping": " ",
    "PostProcessing": " ",
    "Completeness": "BrownCompleteness",
    "TargetList": " ",
    "SimulatedUniverse": " ",
    "SurveySimulation": " ",
    "SurveyEnsemble": " "
```

(continues on next page)

(continued from previous page)

```
  }
}
```

We will look at the star `starName='HIP 19855'`. We start by instantiating the sim object, finding the ind of the star, and setting up the times to evaluate keepout at. We then construct the set of keepout angles from the json script. The instrument specific keepout angles are defined in the suppression system. We then iterate over each time step and calculate the keepout of each star stored in `kogood` as well as the body culprits in `culprit`. Finally, we parse out these culprits to determine boolean arrays indicating when each body or the solar panels are at fault.

```
sim = EXOSIMS.MissionSim.MissionSim(spec, nopar=True)#Create Mission Object To Extract␣
↪Some Plotting Limits
obs, TL, TK = sim.Observatory, sim.TargetList, sim.TimeKeeping
indWhereStarName = np.where(TL.Name == starName)[0]#Get Star Name Ind
koEvaltimes = Time(np.arange(TK.missionStart.value, TK.missionStart.value+TK.missionLife.
↪to('day').value,1),format='mjd')

#Construct koangles
systNames = np.unique([OS.observingModes[x]['syst']['name'] for x in np.arange(len(OS.
↪observingModes))])
koStr     = ["koAngles_Sun", "koAngles_Moon", "koAngles_Earth", "koAngles_Small"]
koangles  = np.zeros([len(systNames),4,2])
for x in np.argsort(systNames):
    rel_mode = list(filter(lambda mode: mode['syst']['name'] == systNames[x], OS.
↪observingModes))[0]
    koangles[x] = np.asarray([rel_mode['syst'][k] for k in koStr])

#Keepouts are calculated here
kogood = np.zeros([1,koEvaltimes.size])
culprit = np.zeros([1,koEvaltimes.size,12])
for t,date in enumerate(koEvaltimes):
    tmpkogood,r_body, r_targ, tmpculprit, koangleArray = obs.keepout(TL,␣
↪[indWhereStarName,indWhereStarName], date, koangles, True)
    kogood[0,t] = tmpkogood[0,0,0] #reassign to boolean array of overall visibility
    culprit[0,t,:] = tmpculprit[0,0,0,:] #reassign to boolean array describing␣
↪visibility of individual keepout perpetrators

#creating an array of visibility based on culprit
sunFault   = [bool(culprit[0,t,0]) for t in np.arange(len(koEvaltimes))]
earthFault = [bool(culprit[0,t,2]) for t in np.arange(len(koEvaltimes))]
moonFault  = [bool(culprit[0,t,1]) for t in np.arange(len(koEvaltimes))]
mercFault  = [bool(culprit[0,t,3]) for t in np.arange(len(koEvaltimes))]
venFault   = [bool(culprit[0,t,4]) for t in np.arange(len(koEvaltimes))]
marsFault  = [bool(culprit[0,t,5]) for t in np.arange(len(koEvaltimes))]
solarPanelFault  = [bool(culprit[0,t,11]) for t in np.arange(len(koEvaltimes))]
```

2.3.5 Calculating Integration Time Adjusted Completeness

This is a set of instructions to use EXOSIMS to calculate integration time adjusted completeness. Integration time adjusted completeness requires the `exodetbox` PYPI package to function [Keithly2021]. The only outspec specification to run with IAC that is requires is specifying `IntegrationTimeAdjustedCompleteness` for the completeness module. To calculate IAC, call comp_calc with the normal smin, smax, dMag parameters and additionally specify tmax, starMass, and IACbool=True. IAC requires an integration time (tmax in days) to adjust completeness by, the mass of the host star to adjust orbital periods, and the boolean indicator to calculate completeness as IAC (IACbool=True). When IACbool=false, subtypecompleteness module computation of completeness is used.

```
comp = sim1.Completeness.comp_calc(smin, smax, dMag, subpop=-2, tmax=0.,starMass=const.M_
↪sun, IACbool=True)
```

Note: Note that IAC relies upon the quasi-Lambert phase function [Agol2007]. This assumption is implicitly made when using IAC.

2.4 Fundamental Concepts

This is a brief summary of fundamental physical concepts underlying the code, and how they are treated in the code. Many more details are available in the *References*.

2.4.1 Orbit Geometry

An exoplanet in `EXOSIMS` is defined via a set of scalar orbital and physical parameters. For each target star S, we define a reference frame $\mathcal{S} = (\hat{\mathbf{s}}_1, \hat{\mathbf{s}}_2, \hat{\mathbf{s}}_3)$, with $\hat{\mathbf{s}}_3$ pointing along the vector from the observer to the star ($\hat{\mathbf{r}}_{S/\text{observer}} \equiv -\hat{\mathbf{r}}_{\text{observer}/S}$) such that the plane of the sky (the plane orthogonal to the this vector lies in the $\hat{\mathbf{s}}_1 - \hat{\mathbf{s}}_2$ plane, as in Fig. 2.2. The $\mathcal{S}$ frame is fixed at the time of mission start, and does not evolve throughout the mission simulation, making $\mathcal{S}$ a true inertial frame. While the orientation of $\hat{\mathbf{s}}_3$) is arbitrary, we take it to be the same inertially fixed direction for all targets (by default equivalent to celestial north).

The planet's orbit is defined via Keplerian orbital elements, where a is the semi-major axis, e is the eccentricity, and the orbit's orientation in the $\mathcal{S}$ frame is given by 3-1-3 (Ω, I, ω) Euler angle set (the longitude of the ascending node, the inclination, and the argument of periapsis, respectively). By default, all of these quantities are considered to be constant (i.e., no orbital evolution due to perturbations or mutual gravitational effects in multi-planet systems), but the code may be extended to account for these effects, in which case they should be treated as the osculating values at epoch.

The planet's instantaneous location at time t is given by the true anomaly $\nu(t)$. The orbit (or osculating orbit, in cases where perturbations are allowed) is fully characterized by a simultaneous measurement of the orbital radius and velocity vectors. The orbital radius vector is given by:

$$\mathbf{r}_{P/S} = \begin{bmatrix} -\sin(\Omega)\sin(\theta)\cos(I) + \cos(\Omega)\cos(\theta) \\ \sin(\Omega)\cos(\theta) + \sin(\theta)\cos(I)\cos(\Omega) \\ \sin(I)\sin(\theta) \end{bmatrix}$$

where r is the orbital radius magnitude:

$$r \equiv \|\mathbf{r}_{P/S}\| = \frac{a(1-e^2)}{1 + e\cos\nu}$$

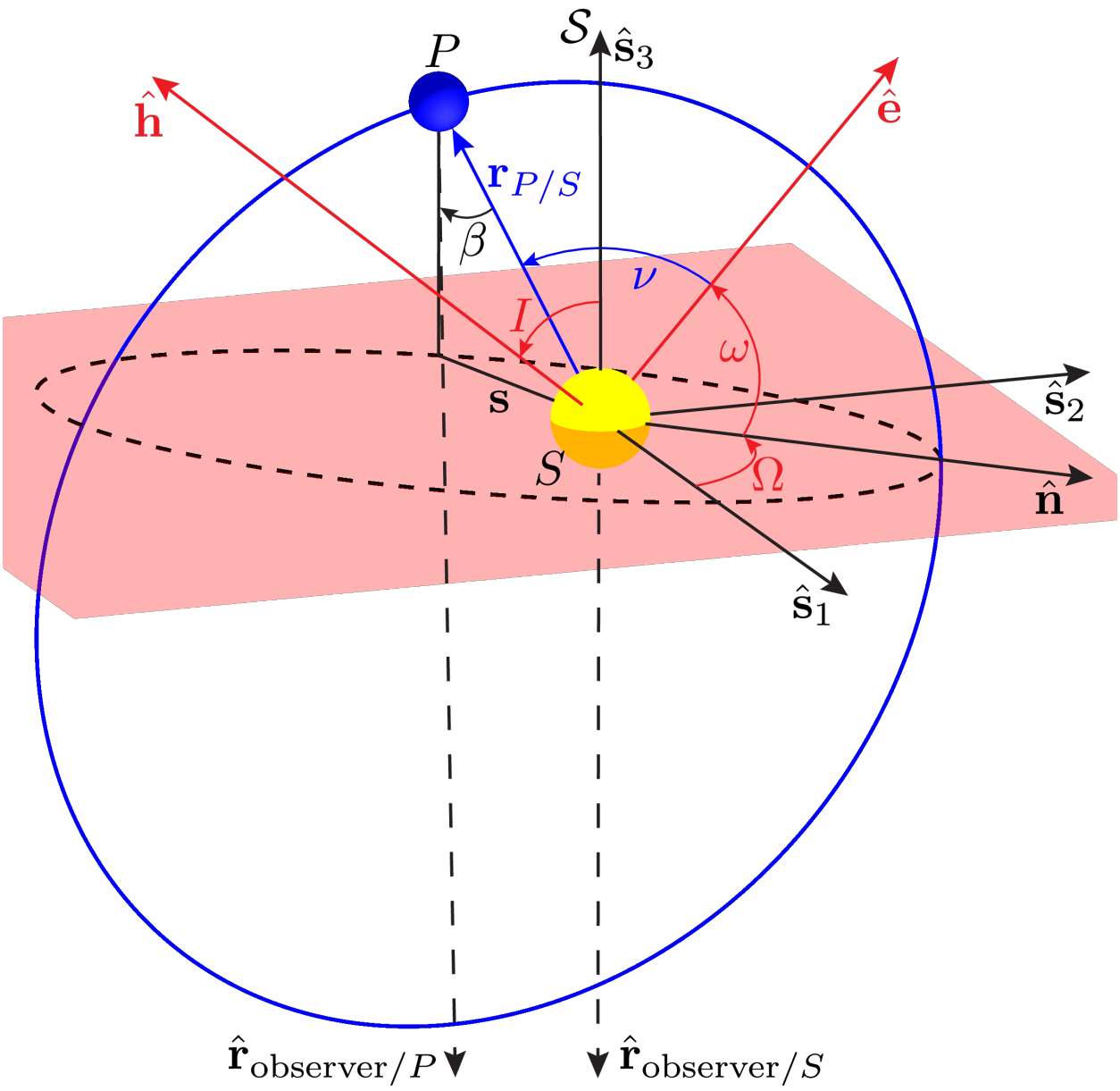


Fig. 2.2: Exoplanetary system orbit diagram.

and θ is the argument of latitude, $\theta \triangleq \nu + \omega$. The orbital velocity vector is given by:

$$\mathbf{v}_{P/S} = \sqrt{\frac{\mu}{a}}\sqrt{\frac{1}{1-e^2}}\begin{bmatrix} -e\sin(\Omega)\cos(I)\cos(\omega) - e\sin(\omega)\cos(\Omega) - \sin(\Omega)\cos(I)\cos(\theta) - \sin(\theta)\cos(\Omega) \\ -e\sin(\Omega)\sin(\omega) + e\cos(I)\cos(\Omega)\cos(\omega) - \sin(\Omega)\sin(\theta) + \cos(I)\cos(\Omega)\cos(\theta) \\ (e\cos(\omega) + \cos(\theta))\sin(I) \end{bmatrix}$$

where μ is the gravitational parameter: $\mu \triangleq G(m_S + m_P)$ for gravitational constant G and star and planet masses m_S and m_P, respectively. Internally, `EXOSIMS` stores the standard gravitational parameters of the stars and planets: $\mu_S = Gm_S$ and $\mu_P = Gm_P$, respectively. Each planet has only a 'true' mass, whereas for each target star, we generate a 'true' and 'estimated' mass, based on a fit to the star's luminosity, and the known error statistics of that fit.

An imaging detection measures the projection of the orbital radius onto the plane of the sky, which is known as the projected separation vector, $\mathbf{s} = \mathbf{r}_{P/O} - \mathbf{r}_{P/O} \cdot \hat{\mathbf{e}}_3$. The projected separation is the magnitude of this vector, and is given by:

$$s \triangleq \|\mathbf{s}\| = r\sqrt{1 - \sin^2(I)\sin^2(\theta)}$$

The calculation of this value from Keplerian orbital elements is provided in EXOSIMS by method *planet_star_separation()*. The angular separation associated with the projected separation can be calculated as:

$$\alpha = \tan^{-1}\left(\frac{s}{d}\right)$$

where d is the distance between the observer and the target star. In the small angle approximation (which applies in essentially all cases) this can be simplified to s/d. EXOSIMS typically does not make such small angle approximations (other than when explicitly noted, as in the case of the phase angle - see *below*), just in case you're trying to do something weird. And because we can.

2.4.2 Photometry

In general, spectral flux density in a given observing band can be approximated as:

$$f = \mathcal{F}_\prime 10^{-0.4m}$$

where $\mathcal{F}_\prime$ is the band-specific zero-magnitude spectral flux density (vegamag, by convention), and m is the band-specific apparent magnitude of the observed object. Multiplying f by the bandwidth ($\Delta\lambda$) of the observing band (see: *Observing Bands*) gives the approximate flux for the observation:

$$F = f\Delta\lambda$$

Further scaling by the effective collecting area (A), all other system throughput losses (τ), and detector quantum efficiency (QE) gives the count rate for the observation in counts/second (or electrons per second):

$$C = FA\tau\mathrm{QE}$$

EXOSIMS utilizes photon-wavelength units for spectral flux densities by default (akin to IRAF/synphot `photlam`; see: https://synphot.readthedocs.io/en/latest/synphot/units.html) but with arbitrary area and wavelength units. Spectral flux densities are thus typically encoded with default units of either photons $\mathrm{m}^{-2}\ \mathrm{s}^{-1}\ \mathrm{nm}^{-1}$ or `photlam`, which is photons $\mathrm{cm}^{-2}\ \mathrm{s}^{-1}\ \mathring{A}^{-1}$. As all quantities have associated units, unit conversion is automatic and occurs as needed in all calculations, and there is never an ambiguity in the units of a particular quantity. See *OpticalSystem* for further details.

2.4.2.1 Stellar Photometry

Following the *equation* above, a star's spectral flux density in a given observing band can be approximated as:

$$f = \mathcal{F}_\prime 10^{-0.4 m_S}$$

where m_S is the star's apparent magnitude in the observing band. If the observing band happens to match (or nearly match) a band where the apparent magnitude of a target star is already known, then both $\mathcal{F}_\prime$ and m_S can simply be looked up from cataloged values, which was the approach in EXOSIMS pre-2016. However, one of the major use cases of EXOSIMS is the analysis of observations in a variety of narrow (and possibly non-standard) bands, which requires better modeling to achieve sufficient fidelity of results.

Between software versions 1.0 and 2.0, EXOSIMS solely utilized the empirical relationships from [Traub2016] to evaluate stellar fluxes in arbitrary bands. The equations in that work (Sec. 2.2) are equivalent to:

$$\mathcal{F}_\prime = 10^{4.01-\left(\frac{\lambda_0}{1\mu\mathrm{m}}-0.55\right)/0.77}\ \text{photons}\ \mathrm{cm}^{-2}\ \mathrm{s}^{-1}\ \mathrm{nm}^{-1}$$
$$m_S = V + b(B-V)\left(\frac{1\mu\mathrm{m}}{\lambda_0} - 1.818\right)$$

where λ_0 is the center of the observing bandpass (or average wavelength, or effective wavelength), V is the target's apparent Johnson-V band magnitude, $B-V$ is the target's B-V color, and scaling factor b is given by:

$$b = \begin{cases} 2.20 & \lambda < 0.55\,\mu\mathrm{m} \\ 1.54 & \text{else} \end{cases}$$

[Traub2016] states that this parametrization is limited to the range $0.4\,\mu\mathrm{m} < \lambda < 1.0\,\mu\mathrm{m}$ and that fluxes calculated in this way are accurate to within approximately 7% in this range. The equations are implemented in EXOSIMS in *`TraubStellarFluxDensity()`*.

Fig. 2.3 shows a comparison of the zero-magnitude spectral flux density computed from the [Traub2016] equation, compared to a calculation of Vega's spectral density in a 10% band for the full valid wavelength range of the [Traub2016] equations, using the spectrum of Vega shown in Fig. 2.4 (`synphot`'s default). The values agree to better than 7%, on average, confirming the statements made in the original paper.

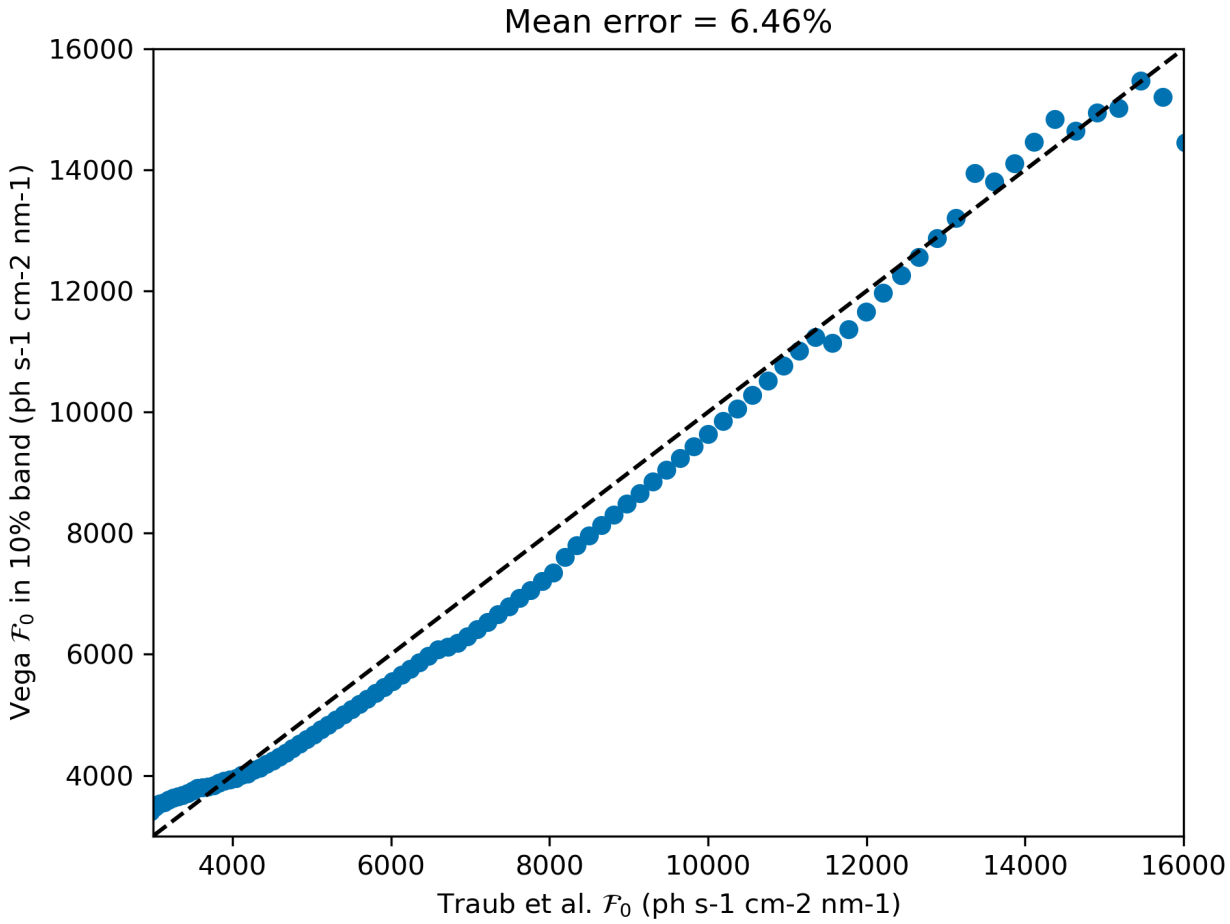


Fig. 2.3: $\mathcal{F}_0$ computed using the [Traub2016] equation compared with Vega's spectral flux density in a 10% band for $0.4\,\mu\mathrm{m} < \lambda < 1.0\,\mu\mathrm{m}$.

Template Spectra

To move beyond the wavelength restrictions of the [Traub2016] equations, starting circa version 2.0, `EXOSIMS` began augmenting these with flux calculations based on template spectra.

Starting with version 3.1, `EXOSIMS` now uses the `synphot` package (https://synphot.readthedocs.io/) for handling photometric calculations based on template spectra. This is a highly mature piece of software, with heritage tracing back to STSDAS SYNPHOT in IRAF and PYSYNPHOT in ASTROLIB. In order to accurately model the stellar flux in any arbitrary observing band for any spectral type, `EXOSIMS` makes use of two spectral catalogs:

1. The Pickles Atlas (specifically the UVKLIB spectra) - 131 flux calibrated stellar spectra covering all normal spectral types and luminosity classes at solar abundance.
2. The Bruzual-Persson-Gunn-Stryker Atlas (BPGS).

All Pickles spectra are normalized to 0 magnitude in vegamag in V band, while all BPGS spectra are normalized to a zero visual magnitude. `EXOSIMS` preferentially uses the Pickles spectra and only uses BPGS when the spectral type is stated.

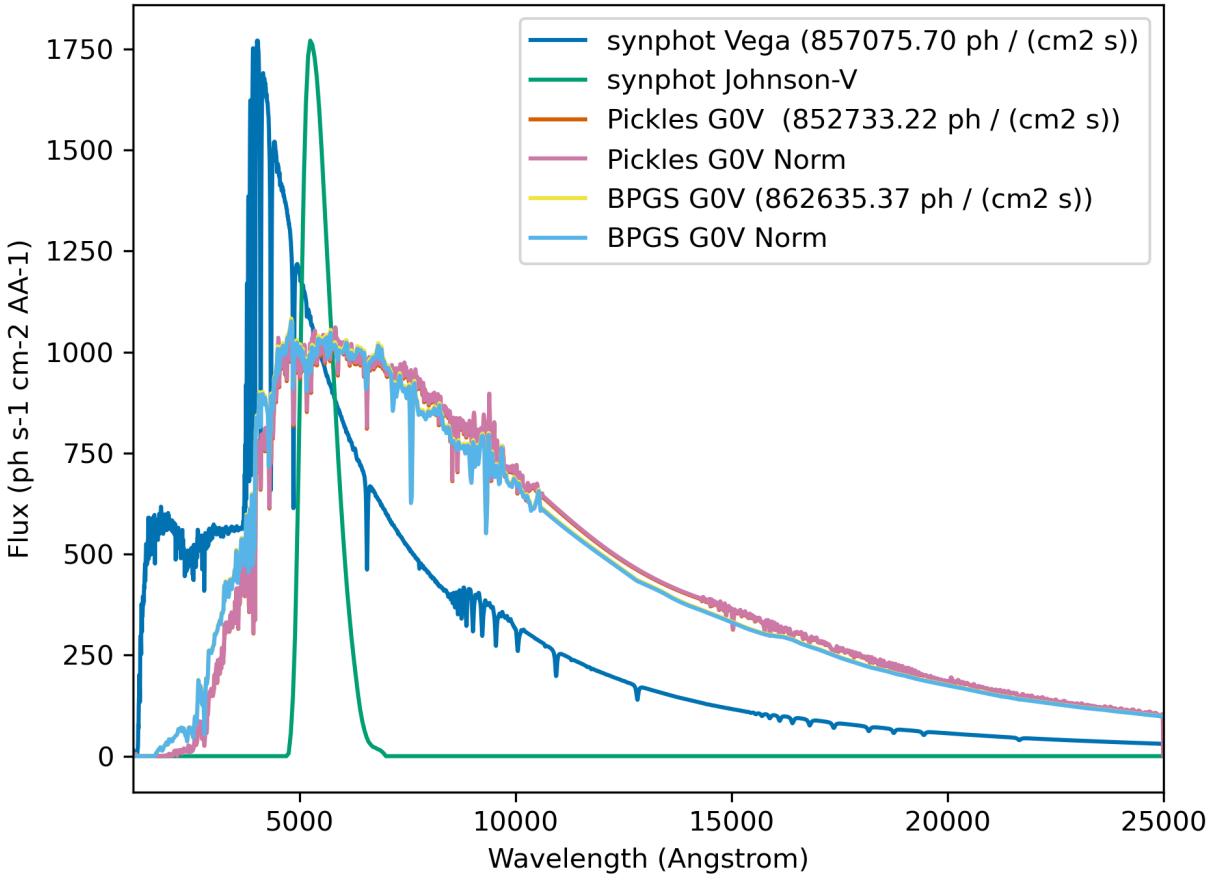


Fig. 2.4: G0V spectra from BPGS, also normalized to zero vegamag using synphot, along with synphot's vega spectrum and Johnson-V bandpass.

Fig. 2.4 shows two G0V spectra pulled from each of the two atlases, along with `synphot`'s default Vega spectrum and Johnson-V filter profile. The values in the legend represent the total integrated flux of each spectrum in the V-band filter. Re-normalizing to zero vegamag has minimal effect on both spectra, but does highlight the differences between

their normalizations and the Vega spectrum used preferentially by `synphot`. Fig. 2.5 shows the differences between the original spectra and their normalizations, as well as the difference between the two normalized spectra, which typically agree to within ~ 100 photons cm^{-2} s^{-1} $\AA^{-1}$.

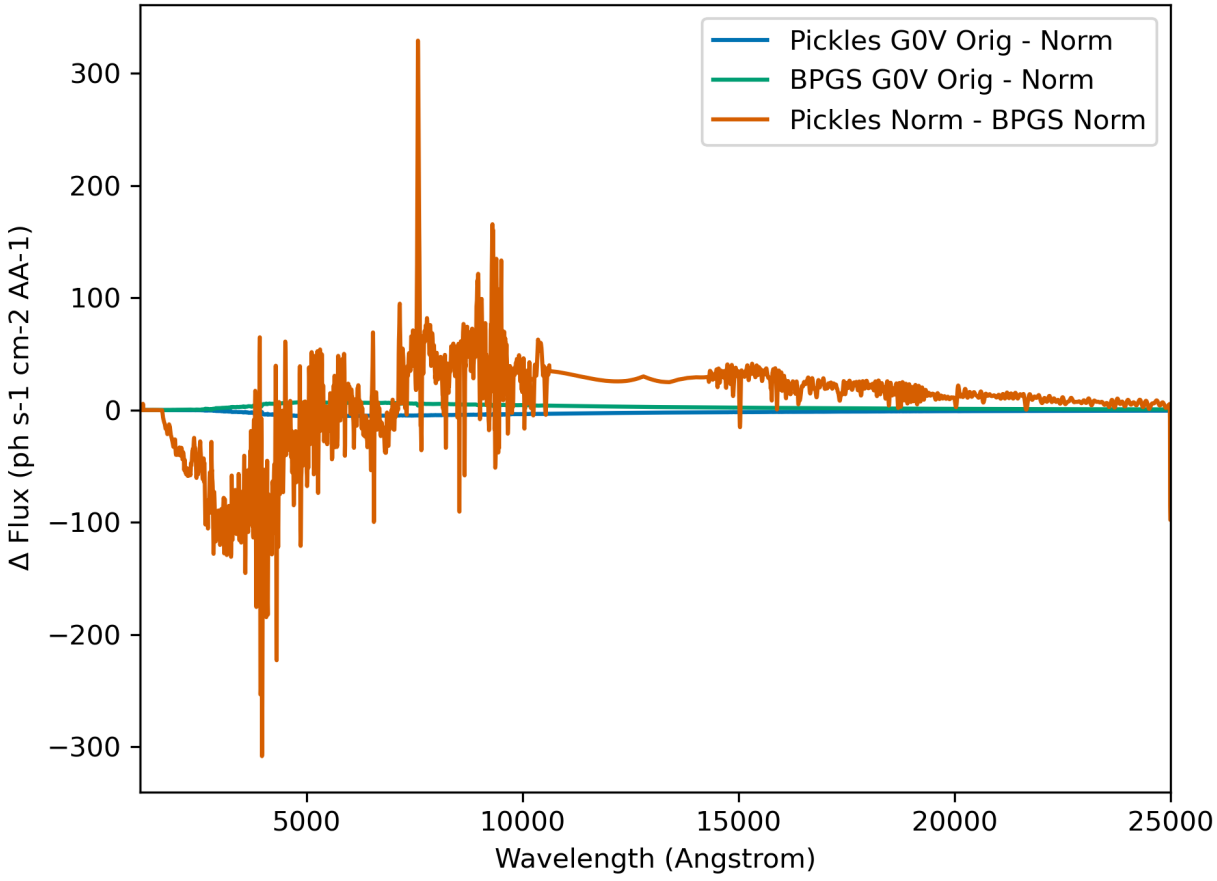


Fig. 2.5: Difference between original and re-normalized G0V spectra from Pickles and BPGS.

The basic procedure for evaluating the stellar flux based on template spectra for a given observing band is:

1. Match the closest available catalog spectrum to the target's spectral type. (At this point we can also optionally apply interstellar reddening, but do not, by default.)
2. Identify the closest (in wavelength) band (see Fig. 2.14) to the desired observing band, for which the original star catalog provided an apparent magnitude value.
3. Re-normalize the catalog spectrum to the target star's magnitude in the identified band.
4. Integrate the spectrum over the observing band to find the stellar flux for the observation.

Important: Computing stellar fluxes directly from template spectra bypasses the need to evaluate (or look up) the zero-magnitude flux in the obserivng band. However, if the zero-magnitude flux is needed for other calculations, it cannot be replaced with the stellar flux, and must be computed separately.

Fig. 2.6 shows a comparison of these two calculations (`synphot` vs. the [Traub2016] equations) for the subset of stars from the EXOCAT star catalog that have spectral types exactly matching entries in the Pickles Atlas. The fluxes are evaluated for a V-band-like observing band with a central wavelength of 549 nm and a Gaussian-equivalent FWHM of

81 nm. This equates to a bandwidth of 85.73 nm, which is the value used for scaling the Traub et al. spectral fluxes. Unsurprisingly (as the original [Traub2016] fits were geared towards V band observations) the two calculations have excellent agreement, differing by only about 1%, on average.

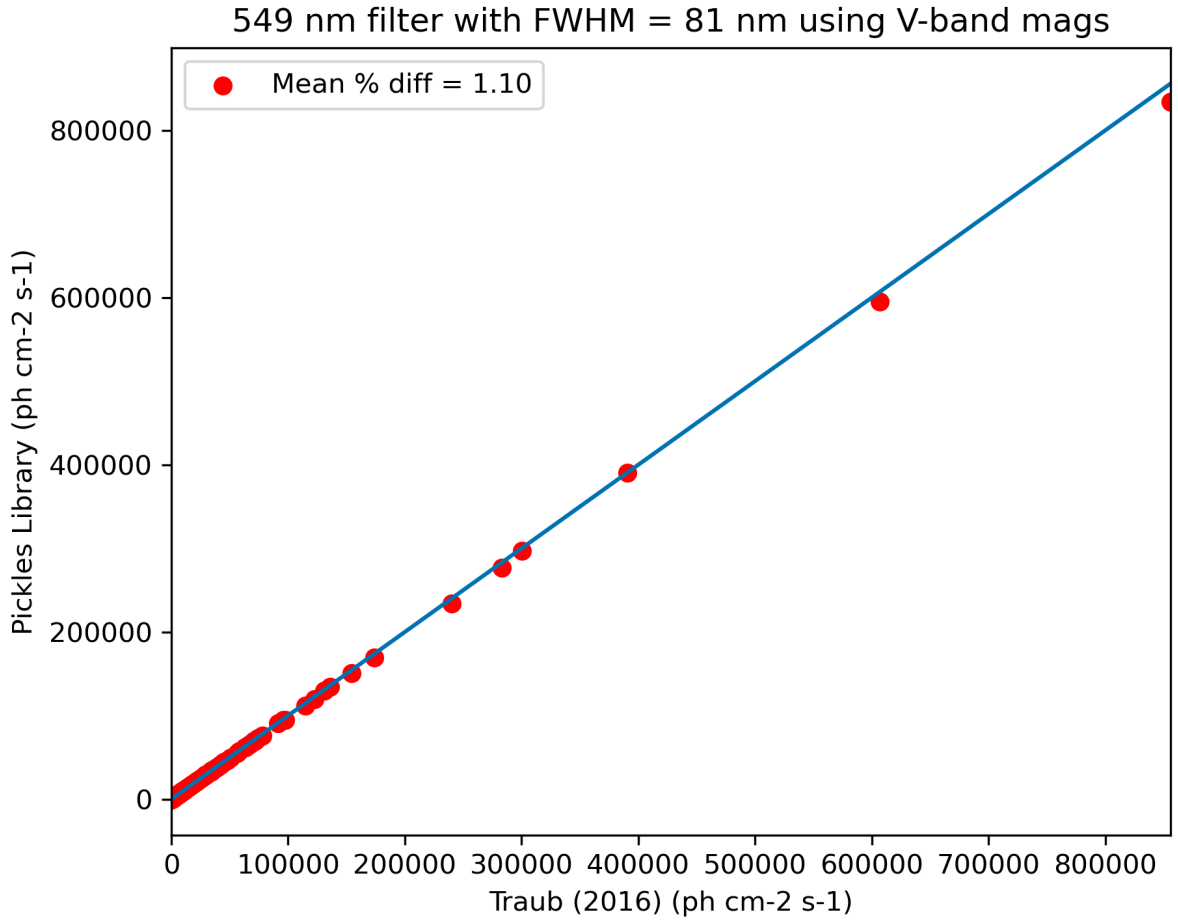


Fig. 2.6: `synphot` Stellar flux calculations using Pickles Atlas templates vs. the [Traub2016] parametric calculation. The points represent 1327 individual target stars and the reference line has slope 1.

We can repeat this experiment again, this time looking at B-band-like observations (439 nm filter with FWHM of 80 nm and equivalent bandwidth of 85 nm), with results shown in Fig. 2.7. IN this case, we perform the `synphot` calculations twice: first re-normalizing each target's template spectrum by its cataloged V-band magnitude (in the Johnson-V band) and next re-normalizing the template spectrum by the cataloged B-band magnitude (in the Johnson B-band). Once again, if we normalize in the appropriate band, the agreement between the template spectrum calculations and the [Traub2016] fits agree very well, with average deviations of only a few percent. Normalizing to V band magnitudes, however, produces averages of 10% error, indicating that use of the empirical relationship may be better in cases where cataloged band magnitudes (or colors) do not exist for a target star.

Finally, we can consider the case of an observing band strictly outside of the stated valid range of the [Traub2016] equations. We repeat the same calculations as in Fig. 2.7, but now using K-band magnitudes and a very narrow observing band (2195 nm filter with FWHM of 19 nm and equivalent bandwidth of 20 nm), with results shown in Fig. 2.8. In this case the `synphot` results diverge sharply from the [Traub2016] model, with average errors of hundreds of percent, depending on which normalization is used.

We can also look at the effects of bandwidth on the [Traub2016] empirical relationship. Fig. 2.9 shows the percent differences between the stellar fluxes computed via the [Traub2016] equations and with `synphot` template spectra as a function of fractional bandwidths for different spectral and luminosity classes. The central wavelength in all cases is

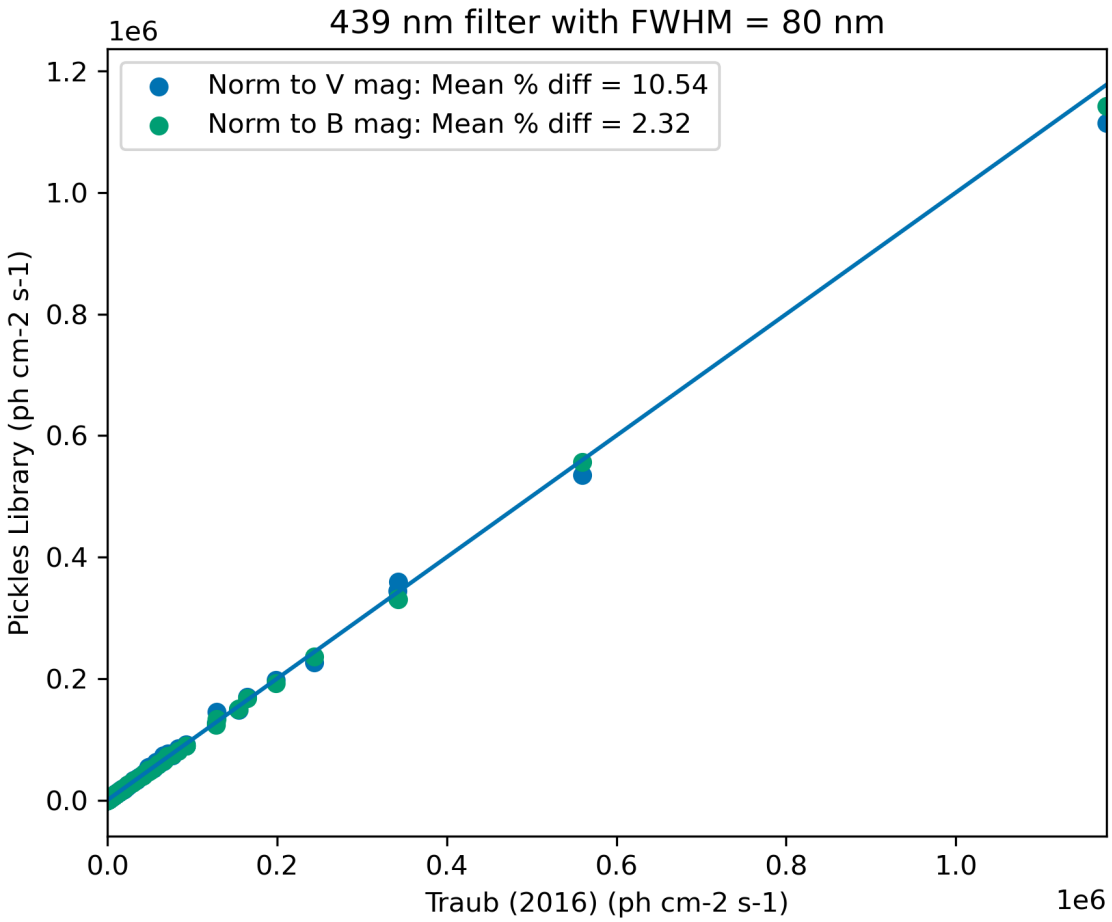


Fig. 2.7: `synphot` Stellar flux calculations using Pickles Atlas templates vs. the [Traub2016] parametric calculation. The points represent 1327 individual target stars and the reference line has slope 1. One set of points represent `synphot` calculations were template spectra were re-normalized by the cataloged target V-band magnitudes, while the other set represents re-normalization by the cataloged B-band magnitudes.

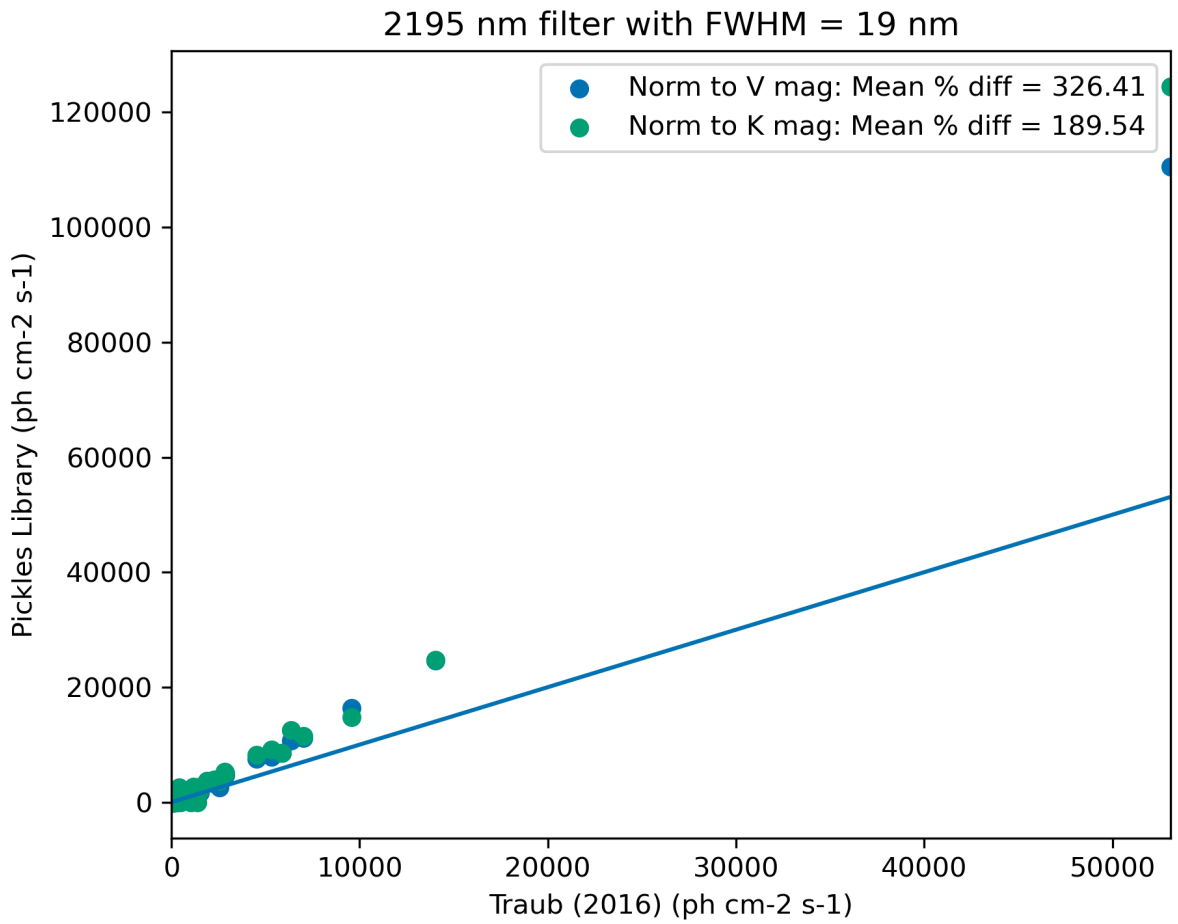


Fig. 2.8: `synphot` Stellar flux calculations using Pickles Atlas templates vs. the [Traub2016] parametric calculation. The points represent 1285 individual target stars and the reference line has slope 1. One set of points represent `synphot` calculations were template spectra were re-normalized by the cataloged target V-band magnitudes, while the other set represents re-normalization by the cataloged K-band magnitudes.

600 nm (the center of the stated valid range for the [Traub2016] equations). The black dashed line represents the limit of the valid range - past this point, the errors are nearly linear in fractional bandwidth for all spectral types considered here (but especially for all of the main sequence spectra. As the data set used to generate the [Traub2016] equations contained primarily dwarf star spectra, it is unsurprising that the equations do a much better job of modeling these spectra than those from other luminosity classes. However, even in the case of giant spectra, the differences between the two calculations (within the valid wavelength range of the equations) is typically well under 10%.

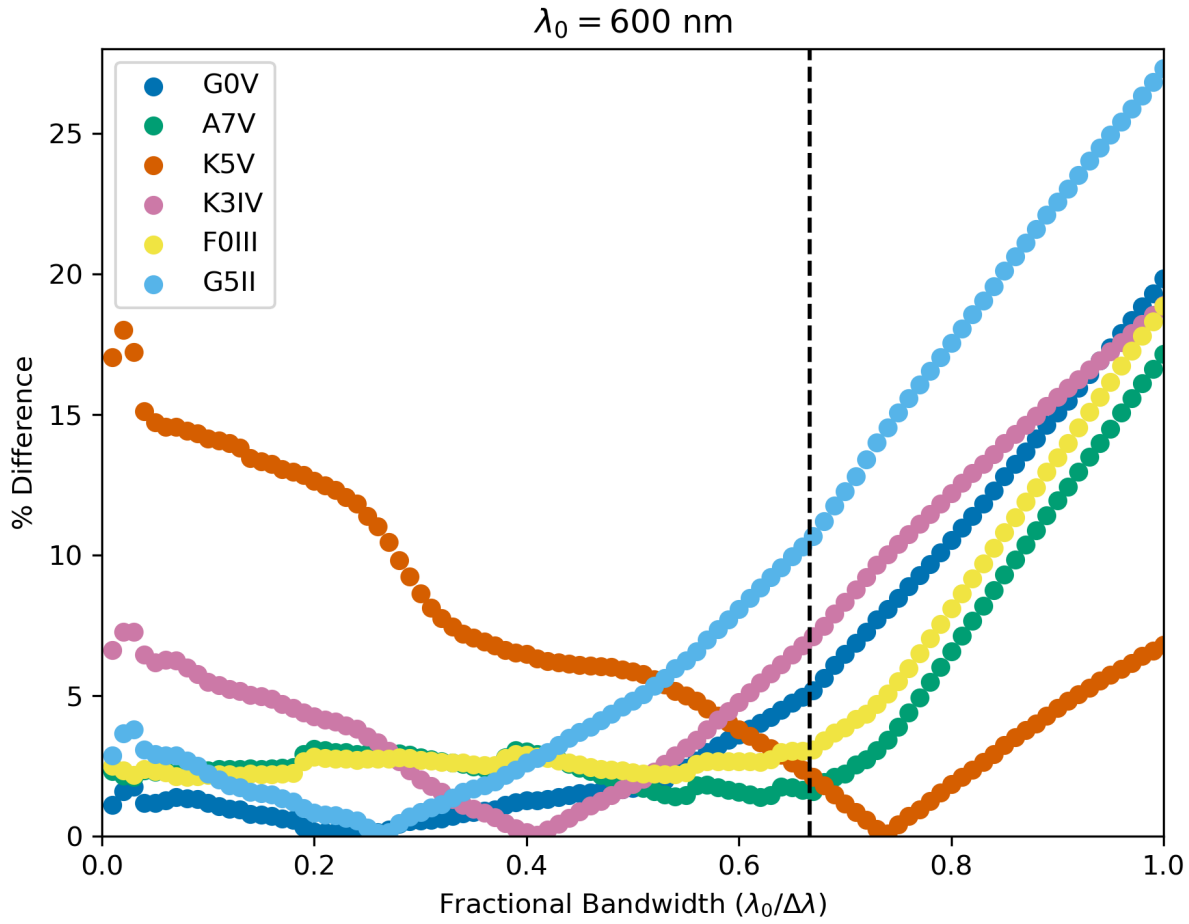


Fig. 2.9: Percent differences between `synphot` and [Traub2016] stellar fluxes as a function of fractional bandwidth with a central wavelength of 600 nm for three different spectral types. The dashed line represents the limit of the stated region of validity of the [Traub2016] equations.

The one exception is the K5V spectrum, which has a qualitatively different pattern of differences from the others. To check whether this is limited to this specific spectrum, we repeat the same calculations using template spectra spanning the whole main sequence. Fig. 2.10 repeats the calculations from Fig. 2.9, but only for main sequence spectral templates.

We can see that the [Traub2016] works best for F, G, and early K stars, holds reasonably well for most early-type stars, and starts to diverge significantly for late-type stars, and especially late K and M dwarfs.

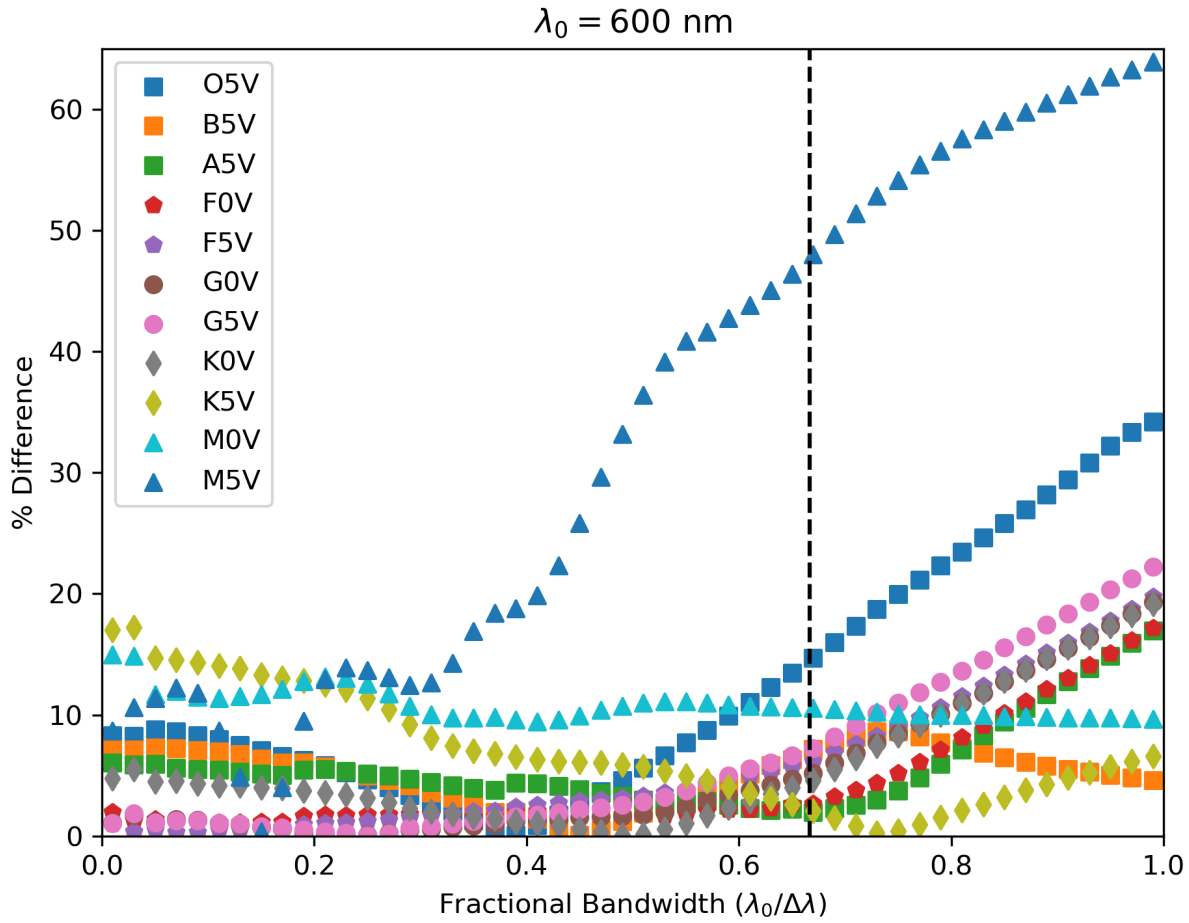


Fig. 2.10: Same as Fig. 2.9, except using only main sequence spectra.

Modeling Mid- to Far-IR Instruments

A fundamental limitation of the template spectra is that they extend only to approximately 2.5 μm. If we wish to model instruments operating beyond this wavelength, then we need to either replace our template spectra with ones covering longer wavelengths, or to rely on idealized black-body curves, parameterized by the stellar effective temperature. By default, `EXOSIMS` does the latter.

As black-body spectra are parametrized by stellar effective temperature, we need to either have these values tabulated in the original star catalog, or to compute them. Where catalog values are unavailable, `EXOSIMS` utilizes the empirical fit from [Ballesteros2012] (Eq. 14), which has the form:

$$T_{\mathrm{eff}} = 4600 \left(\frac{1}{0.92(B-V)+1.7} + \frac{1}{0.92(B-V)+0.6} \right) \mathrm{K}$$

To validate this relationship, we use our template spectra, compute their B-V colors, compute the effective temperatures, and then compare the resulting black-body spectra (normalized to the same V magnitude) to the original ones. In general, we find excellent agreement past 2 μm. Fig. 2.11 shows a sample of this comparison for various spectral and luminosity classes.

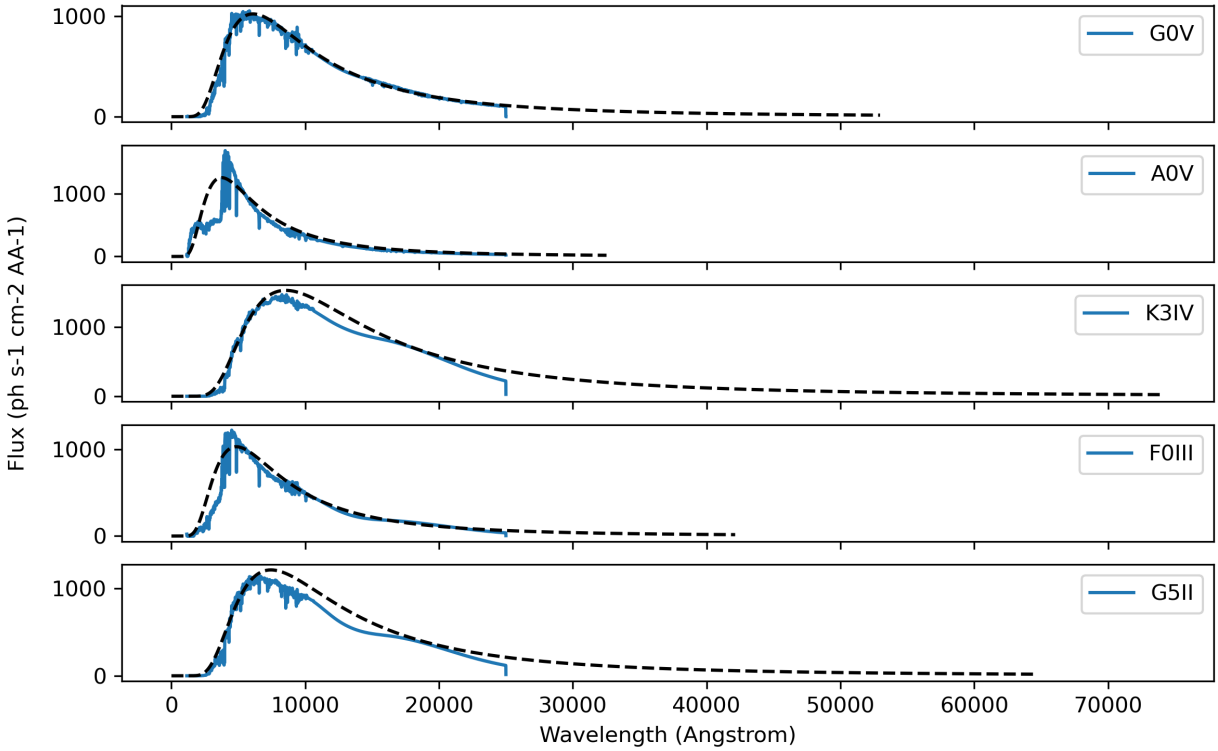


Fig. 2.11: Template spectra with black-body spectra (black dashed lines) for various spectral types, with stellar effective temperature computed using the [Ballesteros2012] fit.

2.4.2.2 Planet Photometry

The second quantity observed by direct imaging is the flux ratio between the planet and star: $\frac{F_P}{F_S}$. This is typically reported in astronomical magnitudes, as the difference in magnitude between star and planet:

$$\Delta\text{mag} \triangleq -2.5\log_{10}\left(\frac{F_P}{F_S}\right) = -2.5\log_{10}\left(p\Phi(\beta)\left(\frac{R_P}{r}\right)^2\right)$$

where p is the planet's geometric albedo, R_P is the planet's (equatorial) radius, and Φ is the planet's phase function (see: *Phase Functions*), which is parameterized by phase angle β. A planet's flux can therefore be calculated from the star's flux and an assumed Δmag as:

$$F_P = F_S 10^{-0.4\Delta\text{mag}}$$

The phase angle is the illuminant-object-observer angle, and therefore the angle between the planet-star vector ($\mathbf{r}_{S/P} \equiv -\mathbf{r}_{P/S}$) and the planet-observer vector $\mathbf{r}_{\text{observer}/P}$, which is given by:

$$\mathbf{r}_{\text{observer}/P} = \mathbf{r}_{\text{observer}/S} - \mathbf{r}_{P/S} = -d\hat{\mathbf{s}}_3 - \mathbf{r}_{P/S}$$

Thus, the phase angle can be evaluated as:

$$\cos\beta = \frac{-\mathbf{r}_{P/S}\cdot(-d\hat{\mathbf{s}}_3 - \mathbf{r}_{P/S})}{r\| - d\hat{\mathbf{s}}_3 - \mathbf{r}_{P/S}\|}$$

If we assume that $d \gg r$ (the observer-target distance is much larger than the orbital radius, a safe assumption for all cases), then the planet-observer and star-observer vectors become nearly parallel, and we can approximate $-d\hat{\mathbf{s}}_3 - \mathbf{r}_{P/S} \approx -d\hat{\mathbf{s}}_3$. In this case, the phase angle equation simplifies to:

$$\cos\beta \approx \frac{-\mathbf{r}_{P/S}\cdot -d\hat{\mathbf{s}}_3}{rd} = \frac{\mathbf{r}_{P/S}}{r}\cdot\hat{\mathbf{s}}_3$$

If we evaluate this expression in terms of the components of the orbital radius vector as a function of the Euler angles defined above, we find:

$$\cos\beta = \sin I\sin\theta$$

Important: `EXOSIMS` adpots the convention that the observer is *below* the planet of the sky, looking up (i.e., along the positive $\hat{\mathbf{s}}_3$ direction in Fig. 2.2). This is different from the convention used elsewhere, and especially the convention adopted by the Exoplanet Archive, where the observer is located *above* the planet of the sky, and looking down (i.e., along the negative $\hat{\mathbf{e}}_3$ axis). Switching conventions has no effect on the calculation of the projected separation, but does flip the sign of the phase angle, such that $\cos\beta = -\sin I\sin\theta$.

It is important to note that not every orbit admits the full range of possible phase angles. As θ always varies between 0 and 2π for every closed orbit, from the *equation*, we see that the phase angle is bounded by the value of the inclination, such that the maximum phase angle falls within the range $\left[\frac{\pi}{2} - I, \frac{\pi}{2} + I\right]$, as shown in Fig. 2.12. For a face-on orbit ($I = 0$), the only possible phase angle is $\frac{\pi}{2}$ (the observer is always at a right angle from the star-planet vector), while an edge-on orbit ($I = \frac{\pi}{2}$), admits the full range of phase angles, $\beta \in [0, \pi]$.

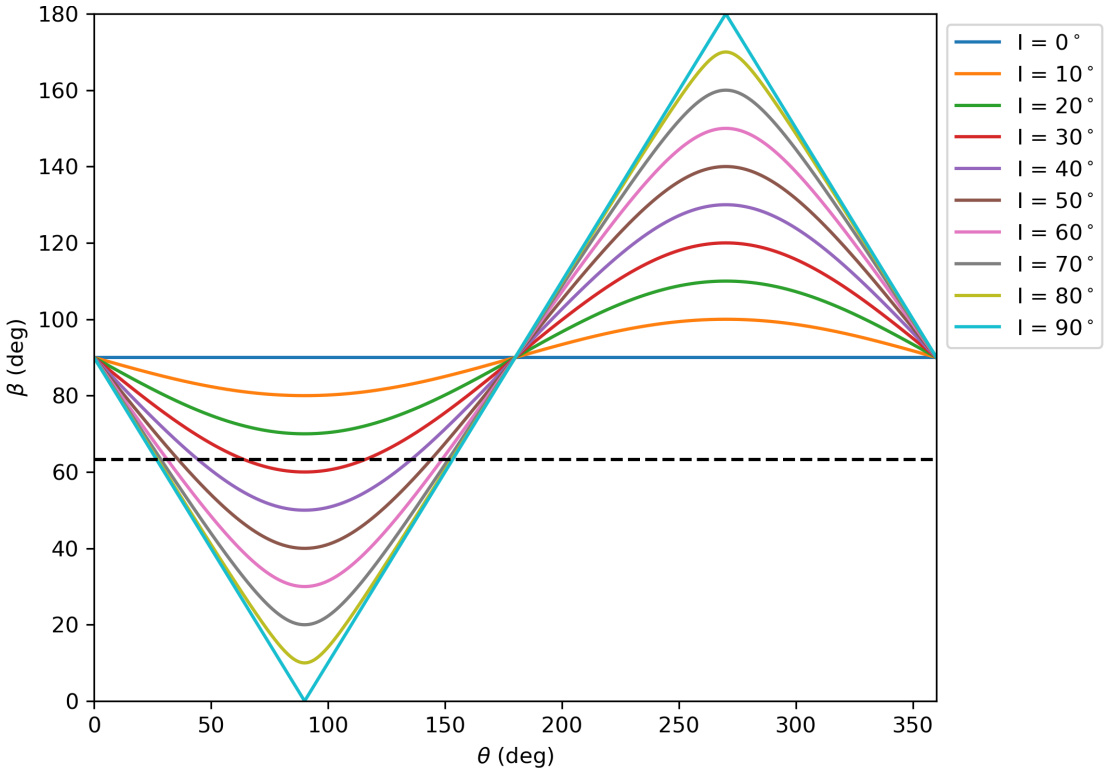


Fig. 2.12: The range of phase angles that can occur within a given orbit are strictly bounded by the orbit's inclination.

2.4.3 Observing Bands

`EXOSIMS` provides several ways to encode an observing band. If a specific filter profile is known (i.e., from measurements of an existing filter, or if use of a standard filter is assumed), then all flux calculations can be done utilizing this profile. Alternatively, if the filter profile is not known exactly, or if the filter definition is at a very early stage of development (i.e., you wish to evaluate a "10% band at 500 nm"), then the filter is internally described either as a box filter (characterized by bandwidth) or a Gaussian filter (characterized by its full-width at half max; FWHM). The bandwidth ($\Delta\lambda$) is defined as:

$$\Delta\lambda = \frac{1}{\max_\lambda T} \int_{-\infty}^{\infty} T \, \mathrm{d}\lambda$$

where T is the wavelength-dependent transmission of the filter (see [Rieke2008] for details). Internally, the variable `BW` is typically treated as the fractional bandwidth:

$$\mathrm{BW} = \frac{\Delta\lambda}{\lambda_0}$$

where λ_0 represents the mean (central) wavelength. A Gaussian of amplitude a has the functional form:

$$f(\lambda) = a \exp\left(-\frac{(\lambda - \lambda_0)^2}{2\sigma^2}\right)$$

where σ is the standard deviation. The full-width at half max of a Gaussian is given by:

$$\mathrm{FWHM} = 2\sqrt{2\ln(2)}\sigma$$

and the integral of the Gaussian is:

$$\int_{-\infty}^{\infty} a \exp\left(-\frac{(\lambda-\lambda_0)^2}{2\sigma^2}\right) \mathrm{d}\lambda = a\sqrt{2\pi}\sigma$$

meaning that we can relate the bandwidth and FWHM of a Gaussian filter as:

$$\Delta\lambda = a\sqrt{\frac{\pi}{\ln(2)}}\frac{\mathrm{FWHM}}{2}$$

Fig. 2.13 shows bandwidth-equivalent Gaussian and box filters corresponding to 10% bands at 500 nm and 1.5 μm (both with amplitude 1), overlaid on the G0V pickles template from Fig. 2.4. The fluxes computed for this spectrum using the two different filter definitions differ by much less than 1% in both cases (0.2% at 500 nm and 0.08% at 1.5 μm).

Fig. 2.14 shows the `synphot` default filter profiles for the standard Johnson-Cousins/Bessel bands, which are used in the template spectra re-normalization step.

2.4.4 Phase Functions

The phase function of a planet depends on the composition of its surface and atmosphere (including any potential clouds), and can be arbitrarily difficult to model. The simplest possible approximation to the phase function is given by the Lambert phase function, which describes a spherical, ideally isotropic, scattering body (none of which are good assumptions for planets. The Lambert phase function is given by (see [Sobolev1975] for a full derivation):

$$\pi\Phi_L(\beta) = \sin\beta + (\pi-\beta)\cos\beta$$

While not strictly correct for any physical planet, the Lambert phase function has the benefit of being very simple to evaluate. In particular, if assuming this phase function, we can strictly bound the Δmag. Following [Brown2004], the flux ratio (and therefore Δmag) extrema for any phase function can be found by solving for the zeros of the derivative of the flux ratio with respect to the phase angle:

$$\frac{\partial}{\partial\beta}\left(\frac{F_P}{F_S}\right) = \frac{2\Phi(\beta)\sin(\beta)\cos(\beta)}{s^2} + \frac{\sin^2(\beta)\frac{\mathrm{d}}{\mathrm{d}\beta}\Phi(\beta)}{s^2} = 0$$

where we have substituted $r = s/\sin(\beta)$ and assumed that both planet radius and geometric albedo are constants. This simplifies to:

$$2\Phi(\beta)\cos(\beta) + \sin(\beta)\frac{\mathrm{d}}{\mathrm{d}\beta}\Phi(\beta) = 0$$

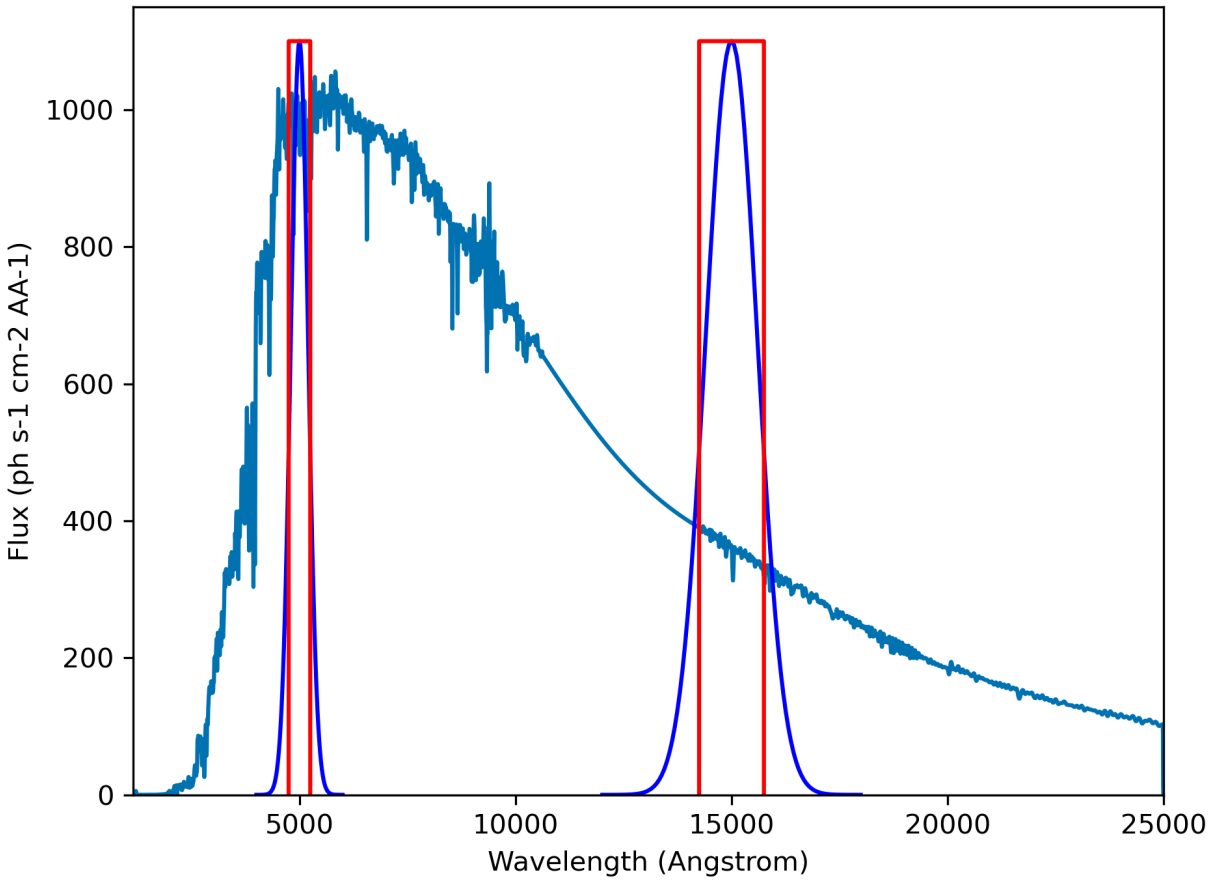


Fig. 2.13: Equivalent bandwidth Gaussian (blue) and box (red) filters for 10% bands centered at 500 nm and 1.5 μm overlaid on the G0V spetrum from Fig. 2.4. The bandpasses have amplitudes of 1 and are arbitrarily scaled for visualization purposes.

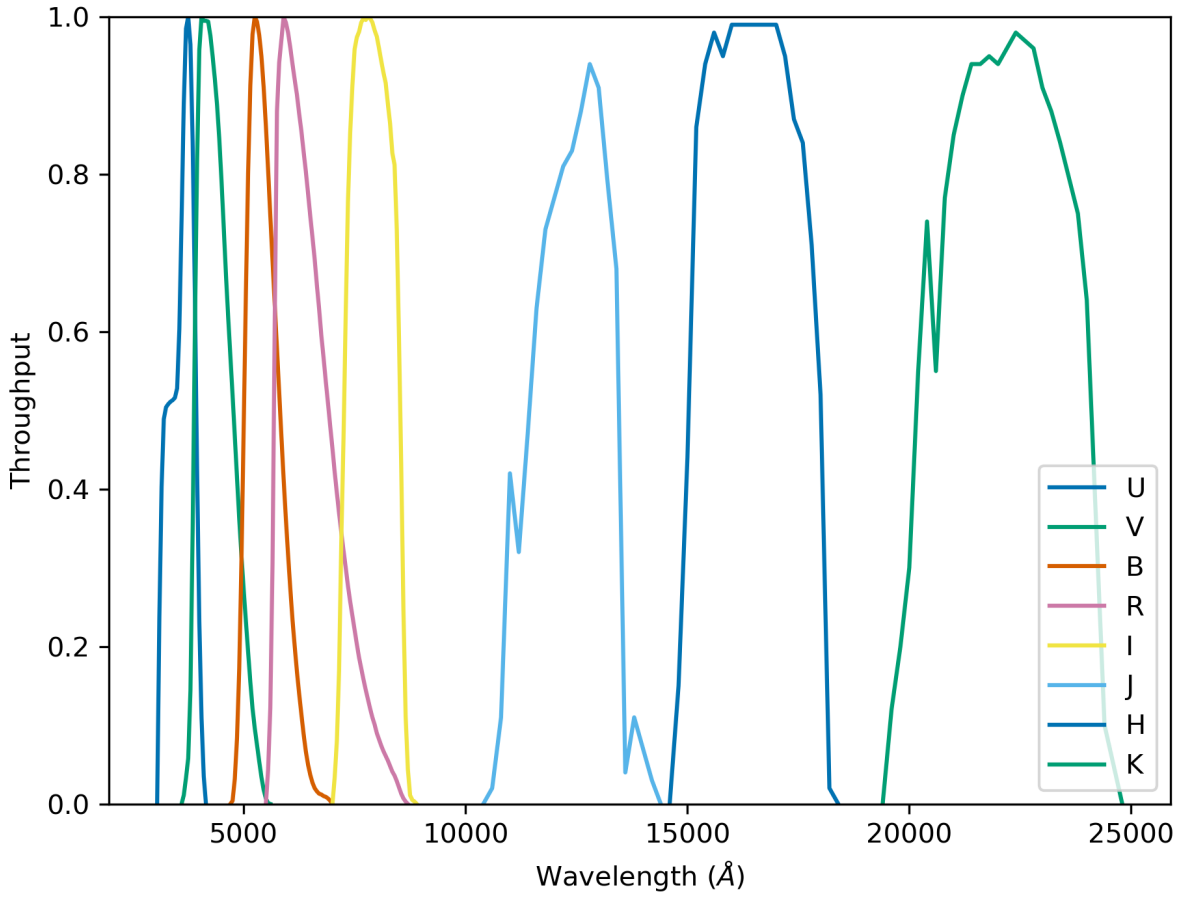


Fig. 2.14: `synphot` default band profiles for Johsnson-Cousins and Bessel bands. UVB are from [Apellaniz2006], RI are from [Bessell1983], and JHK are from [Bessell1988].

Substituting the Lambert phase function, we find the extrema-generating phase angle to be given by:

$$-3\beta\cos(2\beta) - \beta + 2\sin(2\beta) + 3\pi\cos(2\beta) + \pi = 0$$

which, as shown in Fig. 2.15, has a single non-trivial value at $\beta \approx 1.10472882$ rad (or 63.2963 degrees). This is the value shown by the black dashed line in Fig. 2.12.

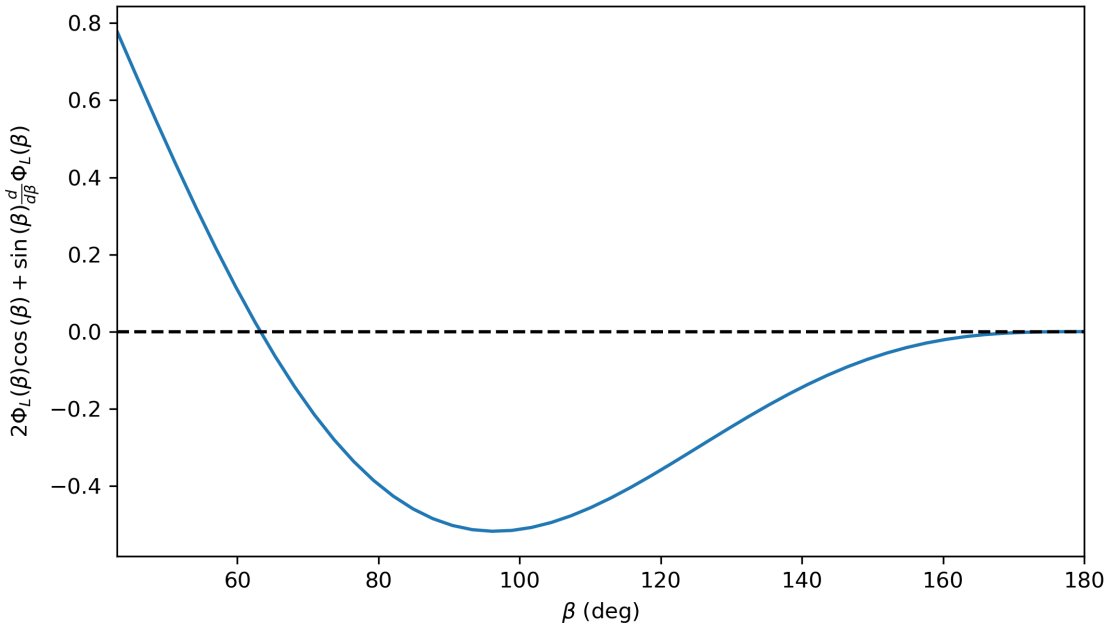


Fig. 2.15: The zeros of this function are the β values corresponding to flux ratio exterma.

A drawback of the Lambert phase function, however, is that it is not analytically invertible. An alternative, suggested in [Agol2007] is the quasi-Lambert function, which, while not physically motivated, approximates the Lambert phase function relatively well, and has the benefit of analytical invertibility:

$$\Phi_{QL}(\beta) = \cos^4\left(\frac{\beta}{2}\right)$$

For further discussion and other phase functions built into `EXOSIMS` see [Keithly2021]. All phase functions are provided by methods in *`phaseFunctions`*.

2.4.5 Completeness, Integration Time, and Δmag

Photometric and obscurational completeness, as defined originally in [Brown2005], is the probability of detecting a planet from some population (given that one exists), about a particular star, with a particular instrument, upon the first observation of that target (this is also known as the single-visit completeness). Completeness is evaluated as the double integral over the joint probability density function of projected separation and Δmag associated with the planet population:

$$c = \int_0^{\Delta\mathrm{mag}_{\max}(s,t_{\mathrm{int}})} \int_{s_{\min}}^{s_{\max}} f_{\bar{s},\overline{\Delta\mathrm{mag}}}(s,\Delta\mathrm{mag})\,\mathrm{d}s\,\mathrm{d}\Delta\mathrm{mag}.$$

The limits on the projected separation are given by the starlight suppression system's inner and outer working angles (*IWA* and *OWA*):

$$s_{\min} = \tan\left(\mathrm{IWA}\right) d \qquad s_{\max} = \tan\left(\mathrm{OWA}\right) d$$

In the small-angle approximation (essentially always appropriate for feasibly starlight suppression systems), these are just $s_{\min} = \mathrm{IWA}d$ and $s_{\max} = \mathrm{OWA}d$. For angles given in arcseconds and distances in parsecs, these evaluate to projected separations in AU.

The lower limit on $\Delta\mathrm{mag}$ technically depends on the assumed planet population, but as the density function will be uniformly zero below this limit, it can be taken to be zero for all separations, without loss of generality. The upper limit on $\Delta\mathrm{mag}$, however, is a function of the instrument *and* the integration time (t_{int}).

The integration time is typically calculated as the amount of time needed to reach a particular *SNR* with some optical system for a particular $\Delta\mathrm{mag}$. We can invert this relationship (either analytically or numerically, depending on the optical system model), to compute the largest possible $\Delta\mathrm{mag}$ that can be achieved by our instrument on a given star for a given integration time. Since the instrument's performance typically varies with angular separation, we end up with a different $\Delta\mathrm{mag}_{\max}$ for every angular separation even if using a single integration time.

Thus, single-visit completeness is directly a function of integration time. The relationship is not always invertible, as completeness is strictly bounded (by unity), meaning that completeness will saturate for some value of integration time. Completeness is also not guaranteed to saturate at unity, for two possible reasons:

1. The projected *IWA* and/or *OWA* for a given star may lie within the bounds of all possible orbit geometries for the selected planet population, such that the maximum obscurational completeness is less than 1.
2. The optical system model may include a noise floor, such that SNR stops increasing with additional integration time past some point. In this case, $\Delta\mathrm{mag}_{\max}$ will saturate at the noise floor integration time, leading to a maximum photometric completeness of less than 1.

All of this is illustrated in Fig. 2.16. The heatmap shows the joint PDF of the assumed planet population (in log scale) and the three black curves represent $\Delta\mathrm{mag}_{\max}(s)$ for three different integration times. All three of the curves have the same limits in s, set by the assumed instrument's inner and outer working angles, projected onto one particular target star. Even though the integration times are logarithmically spaced, we can see that the growth of $\Delta\mathrm{mag}_{\max}(s)$ is not linear on the logarithmic scale of the figure. In this case, this is due to the particular optical system model employed to generate this data. This model assumes that SNR increases as approximately $\sqrt{t_{\mathrm{int}}}$, and that there exists an absolute noise floor. In this specific case, the noise floor corresponds to an integration time of about 6 days, meaning that any integration time larger than this (including the displayed 10 day curve) will produce exactly the same $\Delta\mathrm{mag}_{\max}(s)$ curve and therefor the same completeness value.

All of this can get very complicated very quickly, and all of these calculations depend on having high-fidelity models of the instrument and the numerical machinery to invert the calculation of $\Delta\mathrm{mag}_{\max}$ as a function of integration time. It is typical (especially with instrument models that are not yet well-developed) to make the simplifying assumption (as in [Brown2005] and others) that $\Delta\mathrm{mag}$ is a constant value (sometimes called $\Delta\mathrm{mag}_0$ or $\Delta\mathrm{mag}_{\mathrm{lim}}$ in the literature) for all angular separations and for all targets. In this case, the calculation of completeness is greatly simplified. This simplification is made in `EXOSIMS` by default, but the full calculation is also available.

`EXOSIMS` actually keeps track of 3 sets of completeness, integration time, and $\Delta\mathrm{mag}$ values:

1. The integration time and completeness corresponding to user selected $\Delta\mathrm{mag}_{\mathrm{max}}$ at a particular angular separation from the target (controlled by inputs `int_dMag` and `int_WA` which can be target-specific or global. This is the default integration time and completeness used in mission scheduling (or as an initial guess for further optimization of integration time allocation between targets).
2. The $\Delta\mathrm{mag}_{\mathrm{max}}$ and completeness (stored as `saturation_dMag` and `saturation_comp`, respectively) associated with infinite integration times. These are the saturation values described above. In certain cases, the saturation $\Delta\mathrm{mag}_{\mathrm{max}}$ may be infinite, but the saturation completeness is always strictly bounded by 1. These values are useful in comparing mission simulation results to theoretically maximum yields.

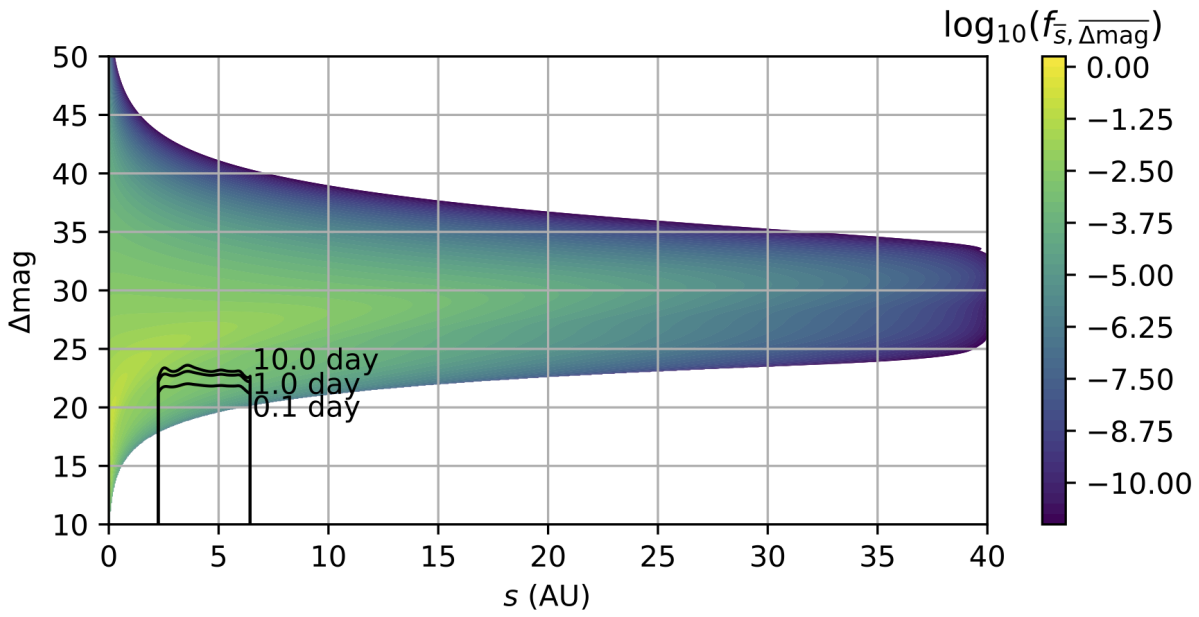


Fig. 2.16: Joint PDF of projected separation and $\Delta\mathrm{mag}$ with $\Delta\mathrm{mag}_{\mathrm{max}}$ curves for various integration times.

3. The $\Delta\mathrm{mag}_{\mathrm{max}}$ and completeness (stored as `intCutoff_dMag` and `intCutoff_comp`, respectively) associated with the maximum allowable integration time on any target by the mission rules (input variable `intCutoff`). In cases where the mission rules do not dictate a cutoff time, these values will be equivalent to the saturation values. These are used to filter out target stars where no detections are likely for a particular mission setup.

See *TargetList* for further details.

Note: Historically, `EXOSIMS` has used multiple different Δmag values. User-supplied values for determining default integration times were previously known as `dMagint` and `WAint`. A user-supplied `dMagLim` was used for evaluating single-visit completeness, and a user-supplied `dMag0` was utilized for computing minimum integratoin times for targets. As of version 3.0, `dMag0` was eliminated entirely, and `dMagLim` replaced by `intcutoff_dMag`.

2.4.6 Stellar Diameter

Because starlight suppression system performance may vary with stellar diameter, `EXOSIMS` needs to be able to track angular sizes for all targets. Where such information is unavailable from the star catalog, we use the polynomial relationship from [Boyajian2014] (specifically for V-band magnitudes and B-V colors), which has the form:

$$\log_{10}\left(\frac{\theta_{\mathrm{LD}}}{1\ \mathrm{mas}}\right) = \sum_{i=0}^{4} a_i(\mathrm{B\text{-}V})^i - 0.2m_V$$

where θ_{LD} is the angular diameter of the star, corrected for limb-darkening, defined as in [HanburyBrown1974], and a_i are the fit coefficients:

$$a_{0\ldots4} = 0.49612, 1.11136, -1.18694, 0.91974, -0.19526$$

We can validate this model in two ways. As a preliminary check, we look at some of the data used for the original model fit. [Boyajian2013], table 2, provides the measured angular diameters for 23 stars, 18 of which overlap with targets in the EXOCAT1 star catalog. For these 18 targets, we compare the model results (using the catalog's B-V and m_V values) to the measurements from the paper, with results in Fig. 2.17.

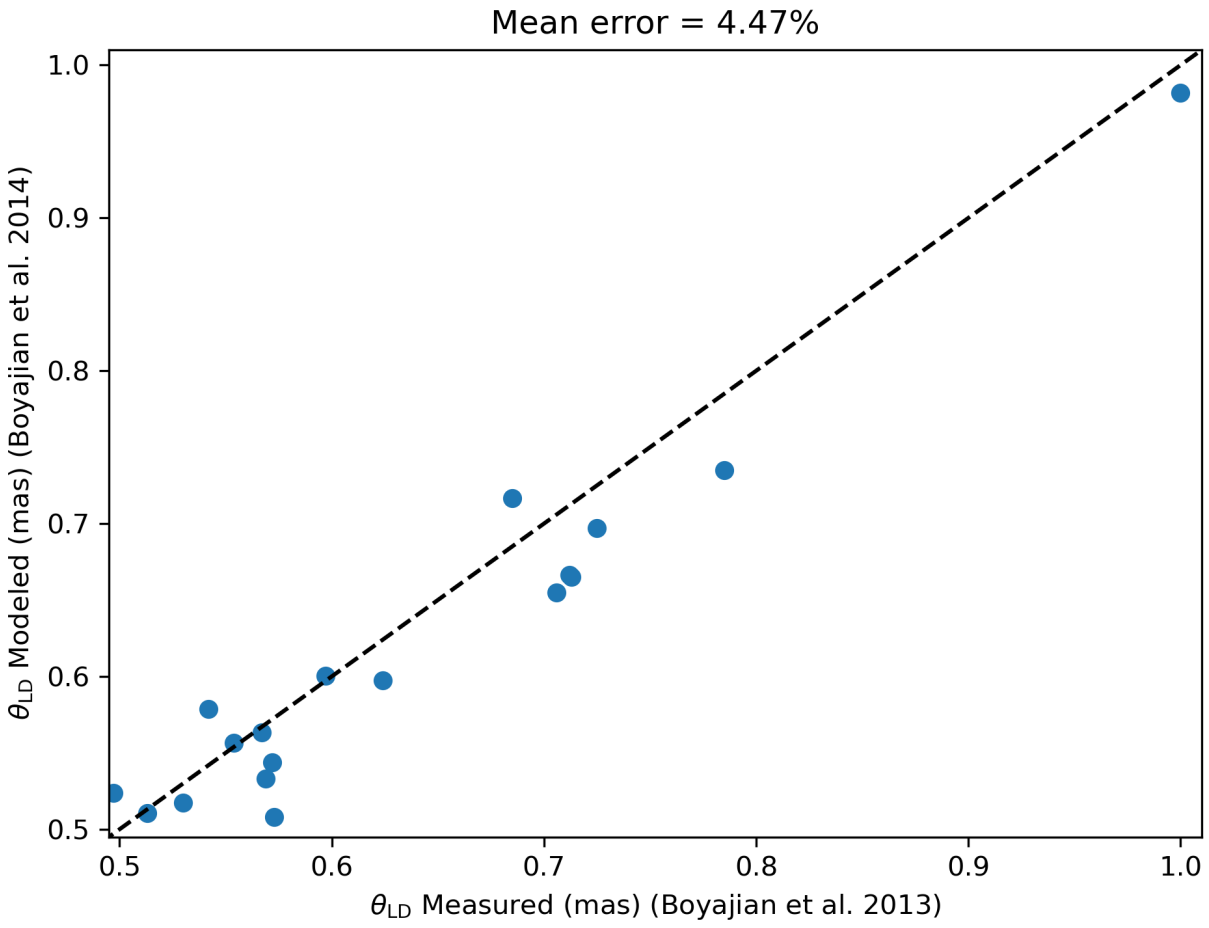


Fig. 2.17: Measured stellar diameters (from [Boyajian2013]) vs. modeled diameters (using model from [Boyajian2014]) for 18 EXOCAT1 targets. The dashed black line has slope=1 for reference.

The measurements and model have excellent agreement, with average errors below 5%. As an additional check, we consider the stellar radius predicted by the Stefan-Boltzmann law:

$$R_\star = \sqrt{\frac{L_\star}{4\pi\sigma_{SB}T_{\text{eff}}^4}}$$

where $L_\star$ is the star's bolometric luminosity as σ_{SB} is the Stefan-Boltzmann constant, with a value of 5.67 $\times 10^{-8}$ W m $^{-2}$ K $^{-4}$. Again relying on the [Ballesteros2012] model for effective temperature (see *Modeling Mid- to Far-IR Instruments*), we can compute the stellar radii (taking 1 solar luminosity to be 3.828 $\times 10^{26}$ W as per IAU 2015 Resolution B3) and then convert them to stellar angular diameters as:

$$\theta = 2\tan^{-1}\left(\frac{R_\star}{d}\right)$$

We compare this calculation with the [Boyajian2014] model for all targets in EXOCAT1, with the results shown in Fig. 2.18. The two calculations have excellent agreement, with mean errors of about 7.55%, despite the very large assumptions being made in the use of the Stefan-Boltzmann law.

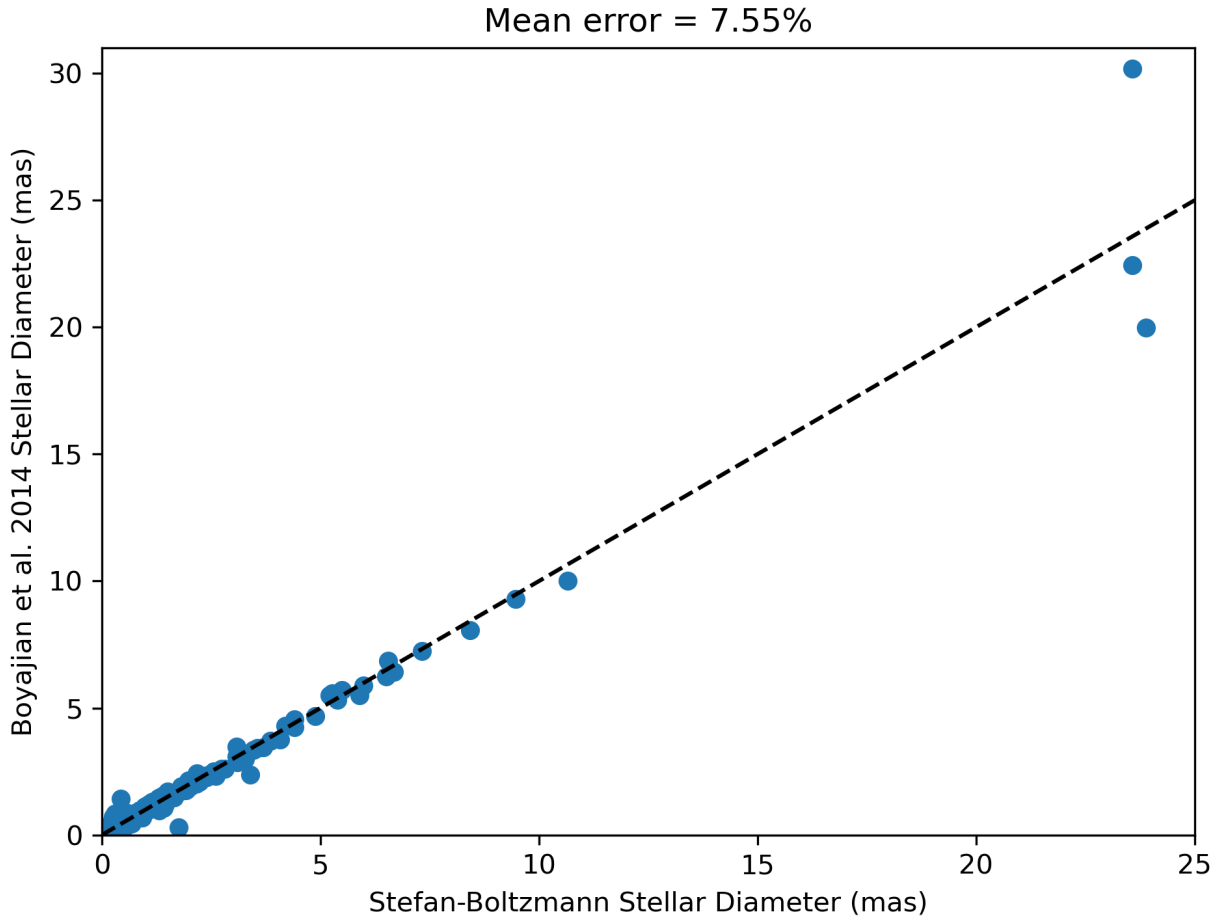


Fig. 2.18: Stellar diameters computed from Stefan-Boltzmann law vs. modeled as in [Boyajian2014] for 2193 stars. The dashed black line has slope=1 for reference.

2.4.7 Zodiacal and Exozodiacal Light

The local zodiacal light represents an important background noise source for all imaging observations, and one of the few that we have any control over when scheduling observations (as the light intensity depends on the orientation of the look vector with respect to the sun). Following [Stark2014] and [Stark2015], `EXOSIMS` uses tabulated data from [Leinert1998] (specifically Tables 17 and 19) to model the wavelength- and orientation-dependent variation in the zodiacal light. Fig. 2.19 shows the data from [Leinert1998] Table 17, linearly interpolated in solar ecliptic longitude ($\Delta\lambda_\odot$) and ecliptic latitude ($\beta_\odot$) corresponding to the target look vector (i.e., observatory to target line of sigh unit vector) and converted to units of photons $\mathrm{m}^{-2}\ \mathrm{s}^{-1}\ \mathrm{nm}^{-1}\mathrm{as}^{-2}$ (cf. [Leinert1998] Fig. 37 and [Keithly2020] Fig. 6). This represents the zodiacal light specific intensity at a wavelength of 500 nm.

Fig. 2.20 shows the data from [Leinert1998] Table 19, converted to units of photons $\mathrm{m}^{-2}\ \mathrm{s}^{-1}\ \mathrm{nm}^{-1}\mathrm{as}^{-2}$ along with a quadratic interpolant in log space (cf. [Leinert1998] Figs. 1 and 38 and [Keithly2020] Fig. 9). This represents the zodiacal light specific intensity at an ecliptic latitude of 0 and solar ecliptic longitude of 90° as a function of wavelength.

We define the interpolant from Fig. 2.19 as $I^V_{\mathrm{zodi},\mathbf{r}}(\Delta\lambda_\odot, \beta_\odot)$ and the interpolant from Fig. 2.20 as $I_{\mathrm{zodi},\lambda}$. Together, they allow us to compute the specific intensity of the local zodiacal light for a given observation as:

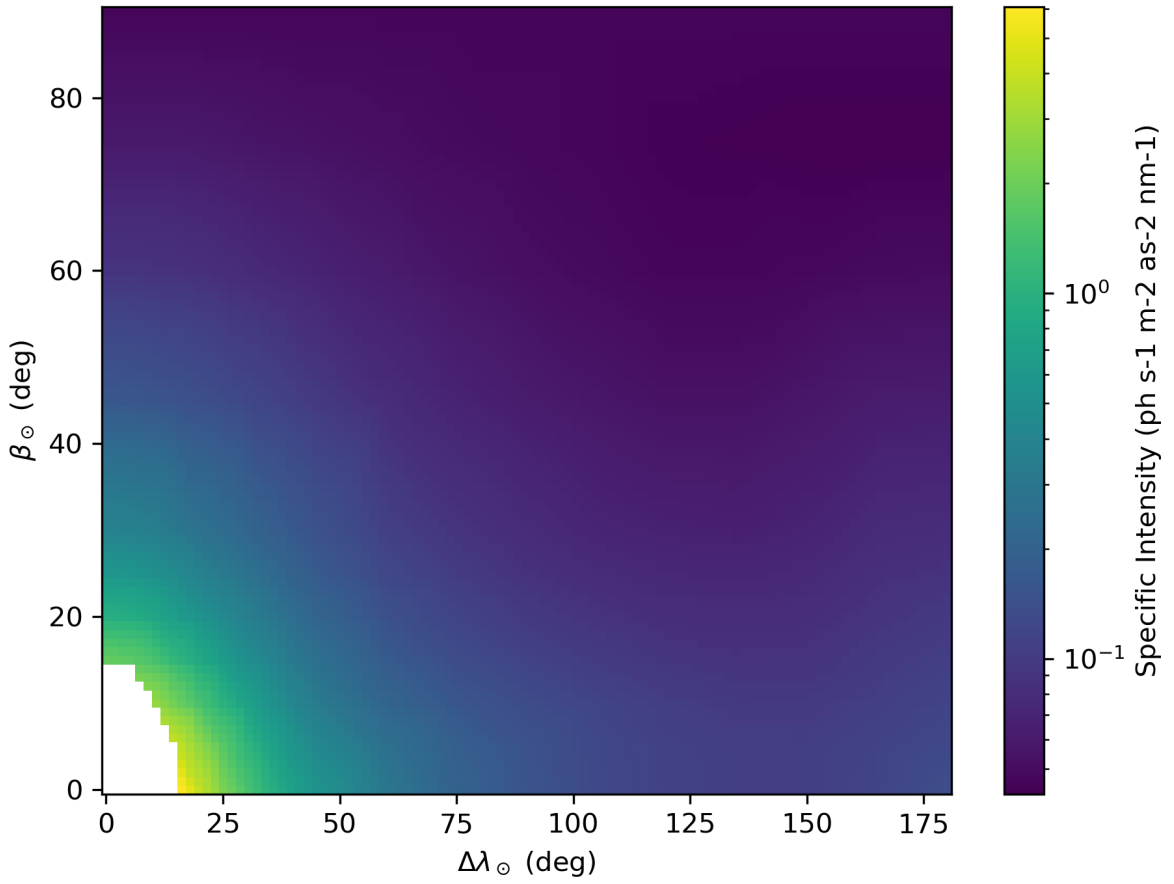


Fig. 2.19: Variation in Zodiacal light specific intensity with look vector orientation. Data from [Leinert1998], Table 17.

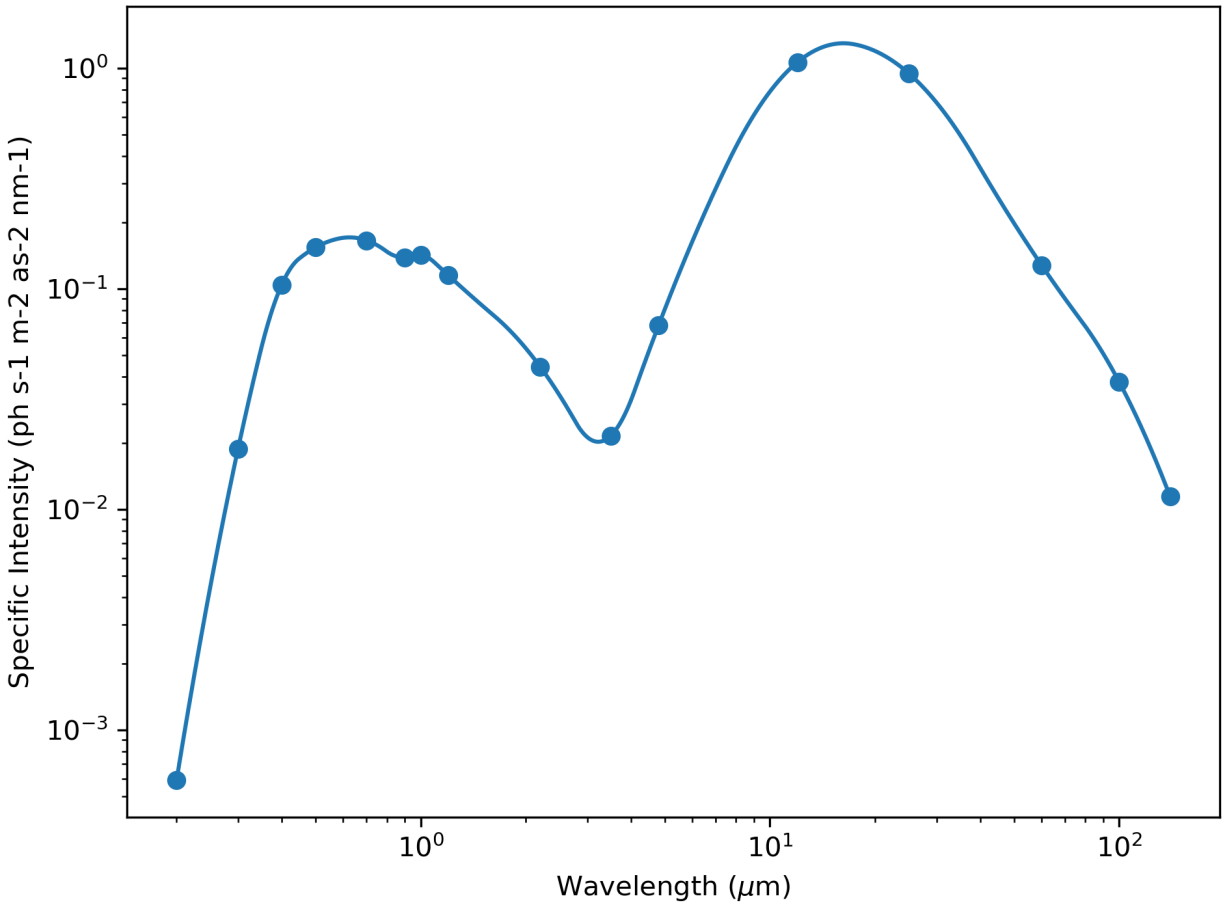


Fig. 2.20: Variation in Zodiacal light specific intensity with wavelength. Data from [Leinert1998], Table 19.

$$I_{\text{zodi}}(\Delta\lambda_\odot, \beta_\odot, \lambda_0) = I_{\text{zodi},\mathbf{r}}(\Delta\lambda_\odot, \beta_\odot)\frac{I_{\text{zodi},\lambda}(\lambda_0)}{I_{\text{zodi},\lambda}(500\text{ nm})}$$

Exozodiacal light is treated much in the same way as the local zodiacal light, save that we allow for a variable number of exozodi, encoded by n_{zodi}, which is defined as in Appendix C of [Stark2014]. The exozodiacal light specific intensity in V-band is evaluated as in equation (C4) of that work:

$$I^V_{\text{exozodi}} = n_{\text{zodi}}\mathcal{F}_{0,V}10^{-0.4(M_V - M_{V,\odot}+x)}\left(\frac{1\text{ AU}}{r}\right)^2$$

where $\mathcal{F}_{0,V}$ is the V-band zero-magnitude flux density, M_V and $M_{V,\odot}$ are the absolute magnitudes of the target star and the sun, respectively, x is the nominal specific brightness of the disk at 1 AU (22 mag arcsec^{-2}), and r is the magnitude of the planet's orbital radius vector at the time of the observation (see: *Orbit Geometry*).

We use the same Table 19 interpolant from [Leinert1998] to find the value in our arbitrary observing band:

$$I_{\text{exozodi}}(r, \lambda_0) = I^V_{\text{exozodi}}(r)\frac{I_{\text{zodi},\lambda}(\lambda_0)}{I_{\text{zodi},\lambda}(500\text{ nm})}$$

Finally, we may also wish to account for the impact of the inclination of the target system on the exozodiacal light brightness. Here we have several options: We can use the empirical relationship of the local zodi's latitudinal variation from the TPF planner model by Don Lindler (2006), which was published as equation 16 of [Savransky2010]. This has the form:

$$f(\theta) = 2.440.0403\left(\frac{\theta}{1\text{ deg}}\right) + 0.0002693\left(\frac{\theta^2}{1\text{ deg}^2}\right)$$

where $\theta \triangleq |90° - I|$ and I is the orbital inclination of the target planet ($\theta \equiv |\beta_\odot|$ for local zodi variations). We compare this against a similar relationship derived in [Stark2014] (equation B4) by fitting to the [Leinert1998] Table 17 data (at $\Delta\lambda_\odot = 135°$, where the local zodi approaches its minimum values). This has the form:

$$f(\theta) = 1.02 - 0.566\sin\theta - 0.884\sin^2\theta + 0.853\sin^3\theta$$

It is important to note that the former fit is normalized at $\theta = 90°$ while the latter is normalized at $\theta = 0°$. Finally, we can compare these to direct interpolants of the [Leinert1998] Table 17 at $\Delta\lambda_\odot = 135°$ and $90°$.

Fig. 2.21 shows a comparison of the two models (with the Lindler model re-normalized at $\theta = 0$) and the two interpolants. All of these, except for the interpolant at $\Delta\lambda_\odot = 90°$ have quite good agreement, and so we choose the Table interpolant at $\Delta\lambda_\odot = 135°$ as our default.

Important: While intensity ratios are all unitless, they will have different values at various wavelengths if evaluated in power units vs. photon units. Therefore, the units of the intensities used to compute the ratios must match those of the intensity being scaled. By default, `EXOSIMS` operates in the original units of the data (power units in the case of the [Leinert1998] tables) and converts to photon units as a final step, when needed.

Scaling the specific intensity values by the optical system's field of view gives the spectral flux densities of the zodiacal and exozodiacal light. For more in-depth discussion, see [Keithly2020] and *ZodiacalLight*.

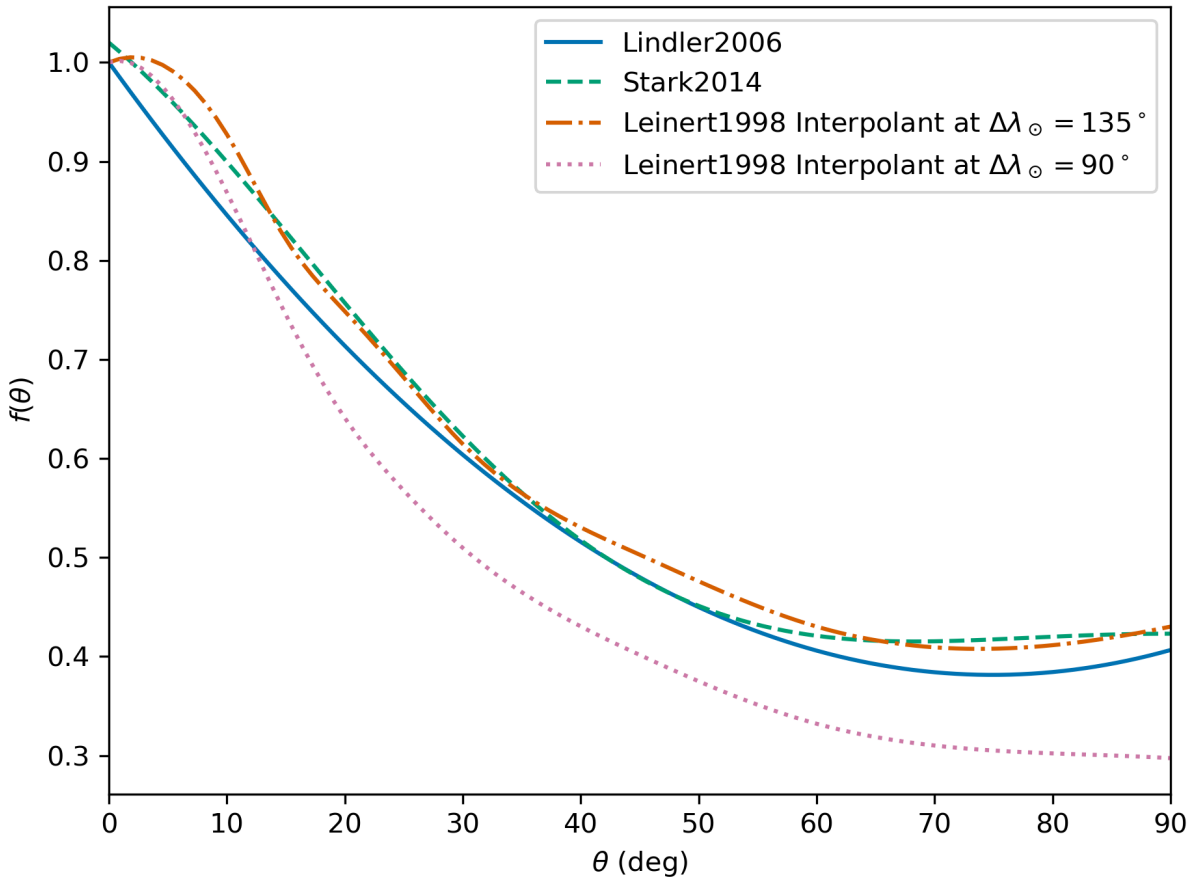


Fig. 2.21: Different models of variation in Zodiacal light with viewing angle.

2.5 Module Implementation

Fig. 2.22 and Fig. 2.23 show schematic representations of the three different aspects of a module, using the `StarCatalog` and `Observatory` modules as examples, respectively. Every module has a prototype that defines the module's standard attributes and methods, including their input/output structure. Prototype implementations also frequently implement common functionality that is reused by all or most implementations of that module type. The various implementations inherit the prototype and add/overload any attributes and methods required for their particular tasks, limited only by the preset input/output scheme for prototype methods. Finally, in the course of running a simulation, an object is generated for each module class selected for that simulation. The generated objects can be used interchangeably in the downstream code, regardless of what implementation they are instances of, due to the strict interface defined in the class prototypes. These objects are always called the generic module type throughout the code (implementation class names are used only when specifying which modules to select for a given simulation).

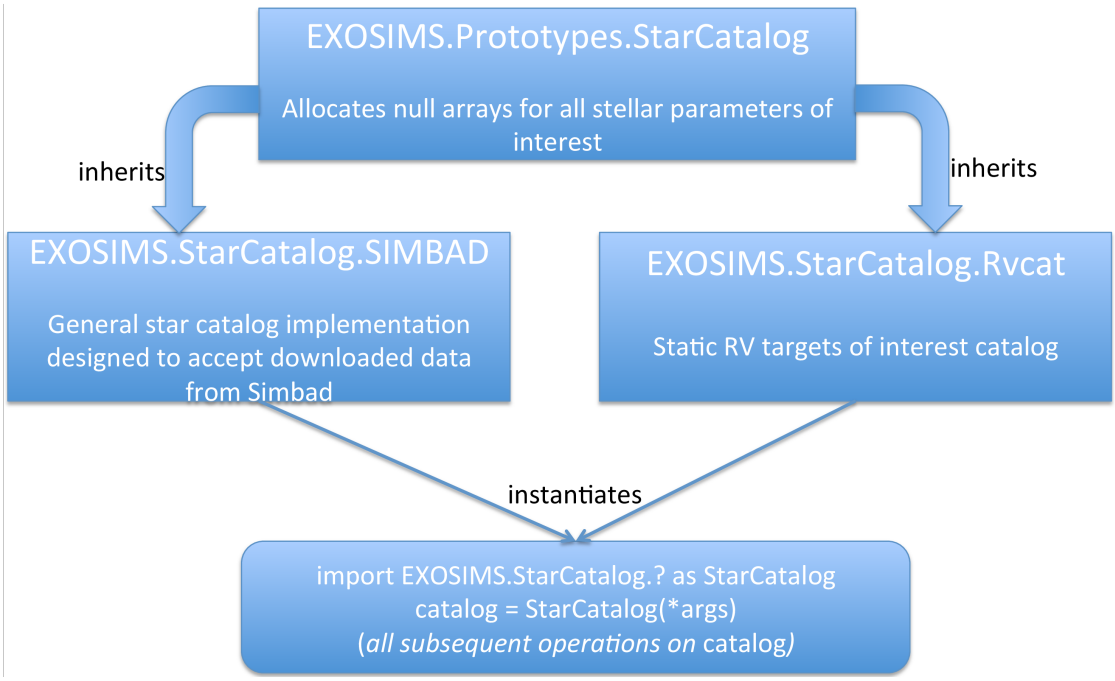


Fig. 2.22: Schematic of a sample set of implementation for the `StarCatalog` module. The prototype (top row) is immutable, specifies the input/output structure of the module along with all common functionality, and is inherited by all `StarCatalog` implementations (middle row). In this case, two different catalog classes are shown: one that reads in data from a SIMBAD catalog dump, and one which contains only information about a subset of known radial velocity targets. The object used at runtime during a simulation (bottom row) is an instance of one of these three classes, is always referred to as `StarCatalog` in all of the code, and can be used in exactly the same way in the rest of the code due to the common input/output scheme for all required methods.

For lower level (downstream) modules, the input specification is much more loosely defined than the output specifica-

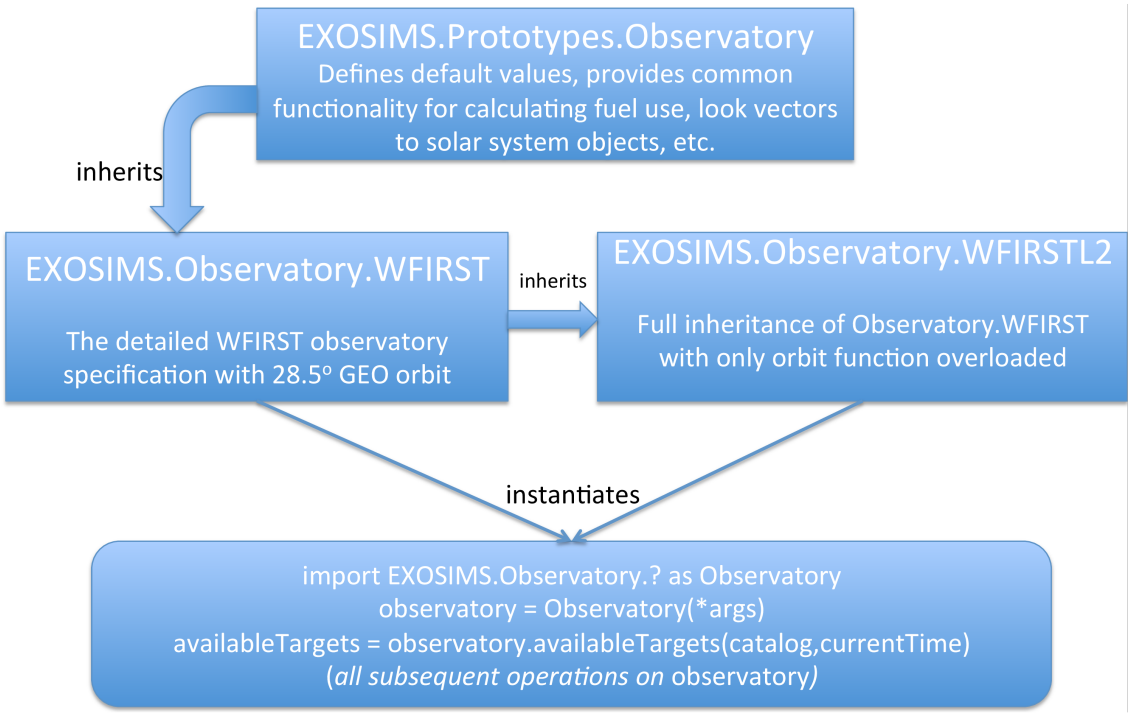


Fig. 2.23: Schematic of a sample set of implementations for the `Observatory` module. The prototype (top row) is immutable, specifies the input/output structure of the module along with all common functionality, and is inherited by all Observatory class implementations (middle row). In this case, two different observatory classes are shown that differ only in the definition of the observatory orbit. Therefore, the second implementation inherits the first (rather than directly inheriting the prototype) and overloads only the orbit method. The object used at runtime during a simulation (bottom row) is an instance of one of these classes, is always referred to as `Observatory` in all of the code, and can be used in exactly the same way in the rest of the code due to the common input/output scheme for all required methods.

tion, as different implementations may draw data from a wide variety of sources. For example, the `StarCatalog` may be implemented as reading values from a static file on disk, or may represent an active connection to a local or remote database. The output specification for these modules, however, as well as both the input and output for the upstream modules, is entirely fixed so as to allow for generic use of all module objects in the simulation.

2.5.1 Module Inheritance and Initialization

The only requirement on any implemented module is that it inherits the appropriate prototype (either directly or by inheriting another module implementation that inherits the prototype). It is similarly expected (but not required) that the prototype `__init__` will be called from the `__init__` of the newly implemented class (if the class overloads the `__init__` method). Here is an example of the beginning of an `OpticalSystem` module implementation:

```
from EXOSIMS.Prototypes.OpticalSystem import OpticalSystem

class ExampleOpticalSystem(OpticalSystem):

    def __init__(self, **specs):

        OpticalSystem.__init__(self, **specs)

        ...
```

Important: The filename **must** match the class name for all modules.

Important: If overloading the prototype `__init__`, the implemented module's `__init__` method **must** have a keyword argument dictionary input (the `**specs` argument in the example, above). This must be the *last* argument to the method. See here for an explanation of the syntax, and see *Input Specification* for further discussion on this input. Note that the name of the input is arbitrary, but is always `**specs` in the EXOSIMS prototypes.

2.5.2 Module Type

It is always possible to check whether a module is an instance of a given prototype, for example:

```
isinstance(obj,EXOSIMS.Prototypes.Observatory.Observatory)
```

However, it can be tedious to look up all of a given object's base classes so, for convenience, every prototype will provide a private variable `_modtype`, which will always return the name of the prototype and should not be overwritten by any module code. Thus, if the above example evaluates as `True`, `obj._modtype` will be equal to `Observatory`.

2.5.3 Callable Attributes

Certain module attributes may be represented in a way that allows them to be parametrized by other values. For example, the instrument throughput and contrast are functions of both the wavelength and the angular separation, and so must be encodable as such in the `OpticalSystem`. To accommodate this, as well as simpler descriptions where these parameters may be treated as static values, these and other attributes are defined as 'callable'. This means that they must be set as objects that can be called in the normal Python fashion, i.e., `object(arg1,arg2,...)`.

These objects can be function definitions defined in the code, or imported from other modules. They can be lambda expressions defined inline in the code. Or they can be callable object instances, such as the various scipy interpolants. In cases where the description is just a single value, these attributes can be defined as dummy functions that always return the same value, for example:

```
def throughput(wavelength,angle):
     return 0.5
```

or, more simply:

```
throughput = lambda wavelength,angle: 0.5
```

Warning: It is important to remember that Python differentiates between how it treats class attributes and methods in inheritance. If a value is originally defined as an attribute (such as a lambda function), then it cannot be overloaded by a method in an inheriting class implementation. So, if a prototype contains a callable value as an attribute, it must be implemented as an attribute in all inheriting implementations that wish to change the value. For this reason, the majority of callable attributes in prototype modules are instead defined as methods to avoid potential overloading issues.

2.5.4 Units

All attributes/variables representing quantities with units are encoded using `astropy.units.quantity.Quantity` objects. Docstrings will often state the default unit used for quantities, but it is never necessary to assume a unit, other than for inputs (see *Input Specification*).

2.5.5 Coding Conventions

EXOSIMS *attempts* to follow standard Python coding conventions (PEP-8, etc.) and it is required that all new code be blackened. Descriptive variable and module names are strongly encouraged. Documentation of existing modules follows the Google docstring style, although the NumPy style is acceptable for new contributions. For more details, see *Docstrings*.

The existing codebase (as it was written by many different contributors) contains a wide variety of naming conventions and naming styles, including lots of CamelCase and mixedCase names. The project PI thinks these look pretty and is firmly unapologetic on this point.

2.6 Interface Specification

The docstrings for the prototypes (see *Framework*) are the interface control documentation (ICD) for `EXOSIMS`.

Warning: Module implementations overloading a prototype method may **not** modify the calling syntax to the method. Doing so will almost invariably cause the new module to not function properly within the broader framework and will almost certainly cause unit tests to fail for that implementation.

New implementations must adhere to the interface specification, and should seek to overload as few methods as possible to produce the desired results. Any change in the method declaration in any prototype is considered interface breaking and will result in a software version bump.

2.7 Input Specification

A simulation specification is a single dictionary, typically stored on disk in JSON-formatted (http://json.org/) file that encodes user-settable parameters and the names of specific modules to use in the simulation.

Both the *MissionSim* and prototype *SurveySimulation* `__init__` can accept a string input (keyword `scriptfile`, or as the first positional argument) describing the full path on disk to a JSON-formatted script file. Alternatively, a user can manually read in such a file (see *Quick Start Guide* for sample code) and then pass its contents to either of these (or any other EXOSIMS module `__init__`) as a keyword dictionary (see *Module Inheritance and Initialization*).

2.7.1 Module Specification

The input specification must contain a dictionary called `modules` specifying which module implementations to use. The order of the modules in the dictionary is arbitrary, but they must **all** be present.

The prototype module can be specified by setting the relevant dictionary entry to the module type (i.e., entry `"BackgroundSources"` is set to `"BackgroundSources"`) or to a string with a single space (i.e., `" "`).

Warning: Setting the module to an empty string will **not** work.

If the module implementation you wish to use is located in the appropriate subfolder in the EXOSIMS tree (and EXOSIMS was installed in editable/developer mode - see *Installing and Configuring*), then it can be specified by the module name (i.e., the name of the module class, which must match the filename). However, if you wish to use an implemented module outside of the EXOSIMS directory, then you need to specify it via its full path in the input specification.

Note: Full path specifications in the input specification may include environment variables.

Here is an example of a module specification, including all 3 types of allowable values (default Prototypes, modules within the EXOSIMS code tree, and modules elsewhere on disk):

```
{
    "modules": {
        "BackgroundSources": " ",
        "Completeness": "GarrettCompleteness",
```

(continues on next page)

(continued from previous page)

```
        "Observatory": "WFIRSTObservatoryL2",
        "OpticalSystem": "Nemati",
        "PlanetPhysicalModel": "Forecaster",
        "PlanetPopulation": "KeplerLike2",
        "PostProcessing": "PostProcessing",
        "SimulatedUniverse": "KeplerLikeUniverse",
        "StarCatalog": "EXOCAT1",
        "SurveyEnsemble": "SurveyEnsemble",
        "SurveySimulation": "SurveySimulation",
        "TargetList": "TargetList",
        "TimeKeeping": "TimeKeeping",
        "ZodiacalLight": "$HOME/myEXOSIMSmodules/myZodicalLightModule.py"
    },
}
```

2.7.2 Keyword Inputs

Essentially all inputs to all EXOSIMS prototype module `__init__` methods are named keywords (so that a user does not need to remember or constantly look up positional argument order, and so that default values can be assigned to all possible inputs). Python syntax is such that any named keywords present in the syntax declaration of a method, which are also present within an input keyword dictionary, will automatically have values copied from the keyword dictionary and assigned to the named keyword. As an example, consider this simplified stub of a class implementation:

```
class EXOSIMSmod1(object):

    def __init__(self, arg1 = 0, **specs):

        self.arg1 = arg1
```

We have a keyword argument (`arg1`) as well as our usual `**specs` keyword argument dictionary. We now try to instantiate three different objects of this class:

```
specs1 = {'arg1': 1}
mod1a = EXOSIMSmod1()
mod1b = EXOSIMSmod1(**specs1)
mod1c = EXOSIMSmod1(arg1=2,**specs1)
```

`mod1a.arg1` will be 0, as that is the default value for this keyword. `mod1b.arg1` will be 1, as that is the value in the `specs1` dictionary. The attempt to initialize `mod1c`, however, will raise a `TypeError` as there will be multiple available values for `arg1`, and Python does not make any choices as to which to use in such instances.

This behavior means that EXOSIMS allows for fairly great flexibility in mixing input specifications stored on disk with additional input values set at runtime. For example, it may be that a user wishes to evaluate the effects of changing a single parameter, say the initial spacecraft mass, which is encoded by input `scMass` to the `Observatory` module, while keeping all other inputs constant. We could create a detailed input specification file on disk (say called `specfile1.json`), which would include values for everything we cared about *except for* `scMass` and then generate multiple different `MissionSim` objects as:

```
sim = EXOSIMS.MissionSim.MissionSim('/path/to/specfile1.json', scMass=5000)
```

with different values for the `scMass` keyword.

2.7.3 Keywords and Nested Module Initializations

When EXOSIMS modules create objects of other modules as part of their initialization (see Fig. 2.1), the same input specification is passed to every `__init__` called. Therefore, in theory, every single input is available to every module constructed in a single object instantiation. However, the behavior of Python is such that when keyword dictionary values are copied to named keywords, the associated keys are popped from the dictionary, meaning that the keyword information will be gone from `**specs` within the very first `__init__` where that keyword explicitly appears. To illustrate: let's consider the addition of a second class to our previous example:

```
class EXOSIMSmod2(object):

    def __init__(self, arg1 = 2, **specs):

        self.arg1 = arg1
        self.mod1 = EXOSIMSmod1(**specs)
```

That is, our `EXOSIMSmod2` will have its own `arg1` attribute (with a different default value from `EXOSIMSmod1`), and will also have an attribute containing an object instance of `EXOSIMSmod1`, with the same keyword dictionary passed to both `__init__` methods. We again have two possible instantiations:

```
specs1 = {'arg1': 1}
mod2a = EXOSIMSmod2()
mod2b = EXOSIMSmod2(**specs1)
```

`mod2a.arg1` and `mod2a.mod1.arg1` will be 2 and 0, respectively, as these are the default values for each. However, `mod2b.arg1` and `mod2b.mod1.arg1` will be 1 and 0, respectively. The value of `arg1` in `specs1` will be applied in the `__init__` of the `EXOSIMSmod2` object, and removed from the dictionary, meaning that the `**specs` passed to `EXOSIMSmod1.__init__` will be empty, causing the resulting object to use its default value for `arg1`.

It's fairly straightforward to get around this issue. If we know that a keyword needs to be reused in a downstream `__init__`, we can just pass it explicitly. Consider a modification to our second class, and the same two instantiations:

```
class EXOSIMSmod3(object):

    def __init__(self, arg1 = 2, **specs):

        self.arg1 = arg1
        self.mod1 = EXOSIMSmod1(arg1=self.arg1, **specs)

specs1 = {'arg1': 1}
mod3a = EXOSIMSmod3()
mod3b = EXOSIMSmod3(**specs1)
```

In this case, the `arg1` and `mod1.arg1` attributes of any `EXOSIMSmod3` object will always be equal. For `mod3a` they will both be 2 (the default) and for `mod3b` they will both be 1 (the value from `specs1`.

It should also be noted that the popping of keyword values only occurs within the scope of the relevant `__init__` (or any other method the keyword dictionary is passed to). In all of these examples, `specs1` remains unmodified in whatever scope it was originally defined. Thus, if a module does not have a particular keyword in its own `__init__`, but instantiates two modules in series that both use the same keyword input, then both will get the value for this keyword (if present) in the `**specs` input. One final illustration:

```
class EXOSIMSmod4(EXOSIMSmod1):

    pass
```

(continues on next page)

(continued from previous page)

```
class EXOSIMSmod5(object):

    def __init__(self, **specs):

        self.mod1 = EXOSIMSmod1(**specs)
        self.mod4 = EXOSIMSmod4(**specs)


specs1 = {'arg1': 1}
mod5a = EXOSIMSmod5()
mod5b = EXOSIMSmod5(**specs1)
```

`EXOSIMSmod4` is an exact copy of `EXOSIMSmod1` (since the `__init__` is directly inherited). Objects of `EXOSIMSmod5` will not have their own *arg1* attributes, but will have two attributes storing object instances of `EXOSIMSmod1` and `EXOSIMSmod4`. The attributes `mod1.arg1` and `mod4.arg1` will always be identical (as both have the same default values. In the case of `mod5a`, both will be the default (0) and for `mod5b` both will have the value from `specs1` (1).

The upshot is that EXOSIMS modules built at the same level can have access to the same entries in the input specification. An example of this is the `missionStart` input (representing the absolute time at the beginning of the mission). This is nominally intended for use by the *TimeKeeping* module, but can also be used by any *Observatory* implementations that need to know absolute times at initialization (before they could potentially have access to a `TimeKeeping` object) for orbital calculations.

2.7.4 Output Specification

Every EXOSIMS module contains a private dictionary attribute called `_outspec`. This dictionary includes a key for **every** user-settable parameter for that module implementation, along with their values once the object has been instantiated and initialized. This means that a given module object's `_outspec` will contain a mixture of user-set values from the input specification, along with default values from the class's `__init__` declaration.

Taken together, all of the 14 module `_outspec` dictionaries will define a complete specification for a given simulation. The `MissionSim`/`SurveySimulation` method *genOutSpec()* provides functionality for assembling a complete specification dictionary out of the 14 independent `_outspec` dictionaries, and (optionally) writing the complete specification to disk in JSON format. This JSON file can then be used as an input specification to another simulation, which will have the exact same parameters as the original simulation. `genOutSpec` also adds code version information to the output, and, in the case where EXOSIMS is installed in developer mode from a git repository (see *Installing and Configuring*), the commit hash.

Warning: If there are code changes in the modules being used that have not been checked in at the time when the output specification is generated, then this will not be captured by the versioning information and may lead to irreproducible results.

Important: Any new user inputs added to the `__init__` of a new module implementation **must** also be added to that implementation's `_outspec` dictionary attribute.

2.7.5 Input/Output Checking

By default, at the end of instantiation of a *MissionSim* object, a check is performed on both the input and output specifications (this can be toggled off via the boolean input keyword `inputCheck`). The check consists of compiling a list of all arguments to all `__init__` methods of every module (including the *MissionSim*), as well as all of the `__init__` methods of any base classes those modules might inherit. This list is then compared against the original input specification, as well as the output specification generated by *genOutSpec()*. Warnings are raised in the event that the input specification includes keywords not appearing in the arguments list, or if the argument list contains entries that aren't in the output specification.

The argument list for all Prototype modules is given in *EXOSIMS Prototype Inputs*.

2.8 BackgroundSources

These modules provide information about background sources that may cause confusion (i.e., false positives) in exoplanet imaging.

2.9 Completeness

The completeness module is responsible for computing the photometric and obscurational *completeness* [Brown2005] of target stars. There are primarily two approaches to performing these computations:

1. Via Monte Carlo methods, as in the original [Brown2005] reference. This is implemented in *EXOSIMS.Completeness.BrownCompleteness*.
2. Semi-analytically, as in [Garrett2016]. This is implemented in *EXOSIMS.Completeness.GarrettCompleteness*.

Because the computations in either approach are so intensive, the completeness prototype actually does not perform any calculations at all, rather returning a constant value of completeness for all targets. This is useful in cases where you wish to instantiate a full survey simulation (or any other module requiring a `Completeness` object) but do not actually need the completeness values. Use of the prototype completeness in these cases will significantly speed things up. However, for running real survey simulations, another implementation must be used.

2.9.1 Initialization

Fig. 2.24 shows the initialization of the *Completeness* prototype.

If the *Input Specification* includes attribute `completeness_specs` (see *the next section*) then a *PlanetPopulation* and *PlanetPhysicalModel* will be generated based on the contents of that attribute. Otherwise, those modules will be generated from the standard modules list. Afterwards, two class methods will be called in succession:

1. *generate_cache_names()*: Generate filenames for any caching to be done by the completeness module
2. *completeness_setup()*: Perform any implementation-specific computations required by the completeness module.

Both of these methods have returns, and set class attributes only. This allows for simple overloading of the particular computations to be executed.

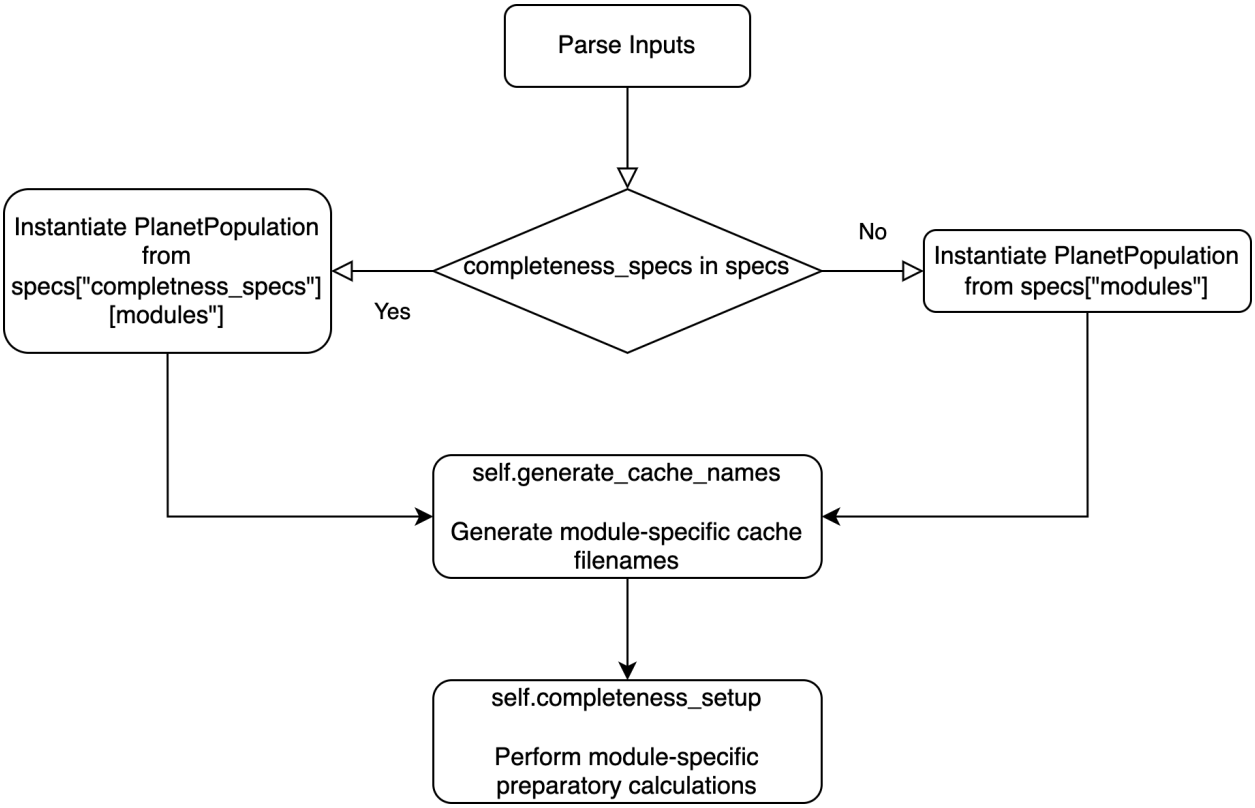


Fig. 2.24: Initialization of a Completeness module.

2.9.2 Different Planet Populations for Completeness

EXOSIMS allows for the calculation of completeness using a different planet population and/or physical model from the one used to populate the simulated universe used in the survey simulation. This functionality is intended to simulate the effects of our current lack of knowledge of the true planet population. The functionality is enabled by adding an optional `completeness_specs` dictionary to the *Input Specification*. This sub-dictionary must contain its own sub-dictionary called `modules`, containing keys for `PlanetPopulation` and (optionally) `PlanetPhyiscalModel`, as well as any inputs to be passed on instantiation of these modules.

Below is an example of (part of) the input specification utilizing this functionality:

```
{
  "completeness_specs":{
    "eta":1,
    "modules":{
      "PlanetPopulation": "JupiterTwin",
      "PlanetPhyiscalModel":" "
  },
...
  "modules": {
    "PlanetPopulation": "KeplerLike2",
    "StarCatalog": "EXOCAT1",
    "OpticalSystem": "Nemati",
    "ZodiacalLight": "Stark",
    "BackgroundSources": " ",
```

(continues on next page)

(continued from previous page)

```
    "PlanetPhysicalModel": "Forecaster",
    "Observatory": "WFIRSTObservatoryL2",
    "TimeKeeping": " ",
    "PostProcessing": " ",
    "Completeness": "BrownCompleteness",
    "TargetList": " ",
    "SimulatedUniverse": "KeplerLikeUniverse",
    "SurveySimulation": "SLSQPScheduler",
    "SurveyEnsemble": " "
  }
}
```

In this example, the target completeness will be evaluated using the *JupiterTwin* planet population with the prototype physical model. The simulated universe, on the other hand, will be populated based on the *KeplerLike2* planet population and the *Forecaster* planet physical model. Note also that the *JupiterTwin* population will be instantiated with input `eta=1`, whereas the *KeplerLike2* instance will not get this input (in order to set the `eta` input for the planet population listed in the main modules dictionary, it would have to be set elsewhere in the input specification, outside of the `completeness_specs` dictionary).

This specific example is illustrating a fairly common use case: optimizing the survey for a particular sub-population of planets (in this case Jupiter analogs), but running the survey simulation on a synthetic universe populated by all types of planets.

In the case where the `completeness_specs` dictionary is omitted, the default behavior is to use the same planet population and physica model for all modules that need them. The behavior in the presence of `completeness_specs` is shown schematically in Fig. 2.25,

2.10 Observatory

The observatory modules provide information about the spacecraft hosting the science instruments and methods for orbital propagation of these spacecraft and for fuel consumption calculations.

2.10.1 Starshades

This documentation describes starshade modelling capabilities in EXOSIMS and how to work with starshades. Starshades are an external occulter for exoplanet direct imaging that enable high contrast imaging. There are two implementations at the moment, using:

- the `Observatory` prototype with methods in the `SurveySimulation` prototype.
- the `SotoStarshade` module with methods in the `SurveySimulation` prototype.

The `SotoStarshade` module inherits the following hierarchy: `Observatory -> ObservatoryL2Halo`. Starshade fuel costs are estimated at different levels of fidelity depending on the chosen implementation. The two main flight modes that require modeling of fuel costs are the station-keeping and slewing of a starshade.

During station-keeping, the starshade is flying in formation with the space telescope during an observation. It must use fuel to keep up with the telescope's motion and line of sight (LOS) to the target star. An additional constraint is that the lateral position of the starshade relative to the LOS cannot exceed 1 meter. This ensures that the desired high contrast imaging is achieved. Chemical propulsion engines, modeled simply using impulsive maneuvers, are typically selected to conduct observations. If a continuous thrust engine is used, the plume produced will reflect light throughout the entire observation.

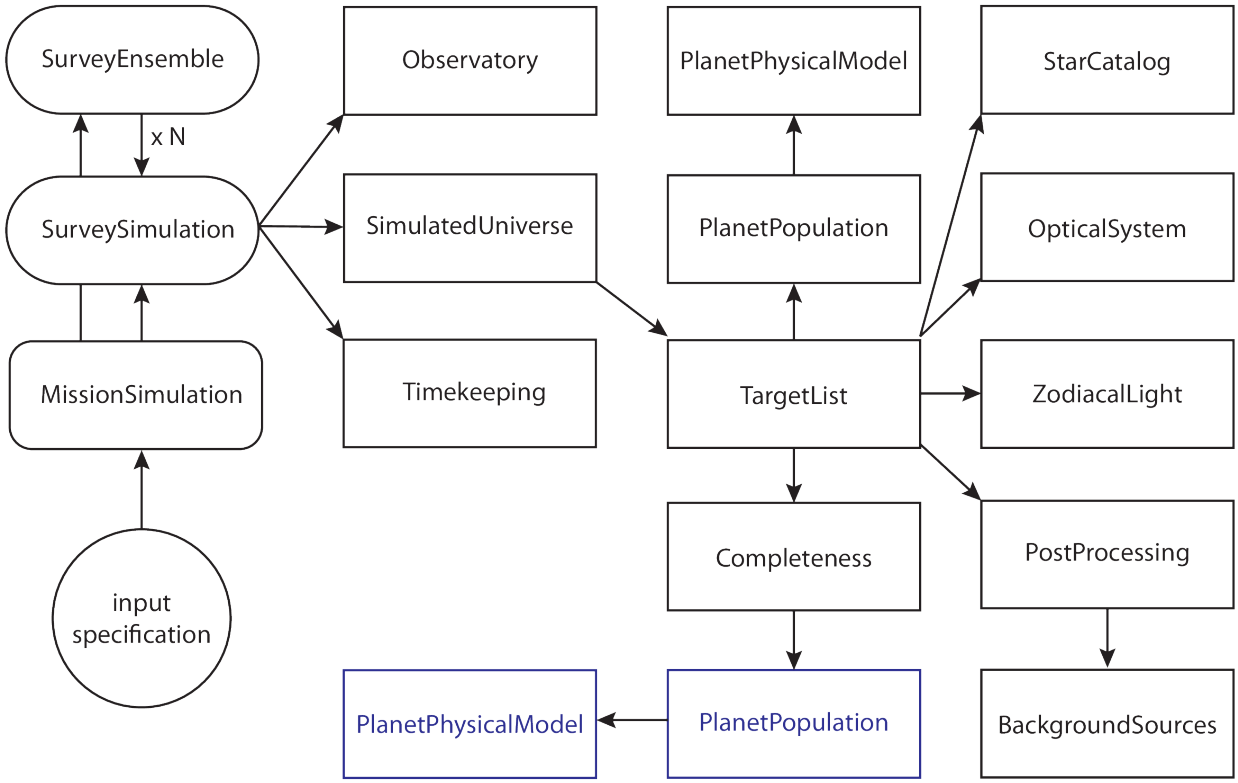


Fig. 2.25: The same schematic representation as in Fig. 2.1, but in the case where the *Input Specification* includes a `completeness_specs` dictionary. In this case, the `PlanetPopulation` and `PlanetPhysicalModel` objects accessed as attributes of the `Completeness` object (i.e., `Completeness.PlanetPopulation`) are distinct from the ones accessed as attributes of the `TargetList` object. In the default behaivor (as in Fig. 2.1), they are the same objects, such that `TargetList.PlanetPopulation is TargetList.Completeness.PlanetPopulation` evaluates to True. In this case, however, the same statement would evaluate as False.

During slewing, the starshade is transfering to the line of sight of a new star at some future time. It uses fuel to initiate this transfer. Either a chemical propulsion or continuous thrust engine can be used for these maneuver. Continuous thrust engines (something like solar electric propulsion) typically use fuel more efficiently. They are typically slower than chemical propulsion in achieving a desired Δ v and have practical limits to how fast they can reach a LOS with certain stars.

2.10.1.1 Prototype

The `Observatory` module contains some basic methods for estimating starshade fuel costs during orbital maneuvers. Note: some of the assumptions in these methods may be insufficiently accurate for fuel modeling.

Prototype - Slew Maneuvers

In the method `SurveySimulation.next_target` we consider starshade slewing maneuvers. For each star left in the target list object `TL`, a slew time is calculated using `Observatory.calculate_slewTimes` as shown in the following code-block:

```
def calculate_slewTimes(self,TL,old_sInd,sInds,sd,obsTimes,currentTime):
   self.ao = self.thrust/self.scMass
    slewTime_fac = (2.*self.occulterSep/np.abs(self.ao)/(self.defburnPortion/2. -
        self.defburnPortion**2./4.)).decompose().to('d2')

    if old_sInd is None:
        slewTimes = np.zeros(TL.nStars)*u.d
    else:
        slewTimes = np.sqrt(slewTime_fac*np.sin(abs(sd)/2.)) #an issue exists if sd is␣
↪negative
        assert np.where(np.isnan(slewTimes))[0].shape[0] == 0, 'At least one slewTime is␣
↪nan'

    return slewTimes
```

The input `sd` (angular separations) is calculated previously within `SurveySimulation.next_target` using the method `Observatory.star_angularSep`. The method assumes a constant acceleration a_0 equal to the thrust value of the occulter slew thrust T_{occ} given by the input `Observatory.thrust` in mN and the current occulter mass m_{occ} `Observatory.scMass` in kg units.

$$a_0 = \frac{T_{occ}}{m_{occ}}$$

The slew time selected for each star is then calculated using the following formula:

$$\Delta t_{slew} = \sqrt{\frac{2d_{occ}}{a_0(\frac{\Delta}{2} - \frac{\Delta^2}{4})} \sin\frac{|\psi|}{2}}$$

This is a function of the following:

- d_{occ} - the occulter separation `Observatory.occulterSep`
- Δ - the default burn portion `Observatory.defburnPortion`
- ψ - the angular separation of the stars in `sInds` to `old_sInd` within the input `sd`
- T_{occ} - the slew thrust `Observatory.thrust`
- m_{occ} - the occulter mass `Observatory.scMass`

Within SurveySimulation.next_target, targets are filtered out of the target list depending on whether the selected slew time coincides with a star being in telescope keepout.

After a star is selected and an observation is complete, the slew has to be conducted to start the next observation. This is done using the Observatory.log_occulterResults method which is called within SurveySimulation.next_target as follows:

```
def log_occulterResults(self,DRM,slewTimes,sInd,sd,dV):
    DRM['slew_time'] = slewTimes.to('day')
    DRM['slew_angle'] = sd.to('deg')

    slew_mass_used = slewTimes*self.defburnPortion*self.flowRate
    DRM['slew_dV'] = (slewTimes*self.ao*self.defburnPortion).to('m/s')
    DRM['slew_mass_used'] = slew_mass_used.to('kg')
    self.scMass = self.scMass - slew_mass_used
    DRM['scMass'] = self.scMass.to('kg')

    return DRM
```

This method updates the given DRM dictionary and populates it with occulter slew parameters. These include:

- 'slew_time' - or the slew time Δt_{slew}
- 'slew_angle' - or the slew angle ψ
- 'slew_mass_used' - or the slew mass used $\Delta m_{slew} = \dot{m}_{slew}\Delta t_{slew}\Delta$
- 'slew_dV' - or the slew delta-v $\Delta v_{slew} = a_0\Delta t_{slew}\Delta$
- 'scMass' - or the occulter mass $m_{occ,new} = m_{occ} - \Delta m_{slew}$

The mass flow rate $\dot{m}_{slew}$ is

$$\dot{m}_{slew} = \frac{T_{occ}}{g_0 I_{sp,slew}}$$

which is a function of the attribute $I_{sp,slew}$ the specific impulse Observatory.slewIsp of the slewing engine. The attribute Observatory.scMass is also updated by subtracting the fuel mass used.

Prototype - Station-Keeping Maneuvers

The station-keeping maneuvers for starshade formation flying are not used in the decision-making process of the SurveySimulation.Prototype module. The fuel costs are only estimated and used to decrease the starshade mass accordingly after an observation is conducted. Station-keeping fuel usage can occur during different modes of observation, including det (detection) and char (characterization). The fuel costs are used in SurveySimulation.run_sim which calls the SurveySimulation.update_occulter_mass method. This method is used as follows:

```
def update_occulter_mass(self, DRM, sInd, t_int, skMode):
    TL = self.TargetList
    Obs = self.Observatory
    TK = self.TimeKeeping

    assert skMode in ('det', 'char'), "Observing mode type must be 'det' or 'char'."

    dF_lateral, dF_axial, intMdot, mass_used, deltaV = Obs.mass_dec_sk(TL, \
            sInd, TK.currentTimeAbs.copy(), t_int)
```

(continues on next page)

(continued from previous page)

```
    DRM[skMode + '_dV'] = deltaV.to('m/s')
    DRM[skMode + '_mass_used'] = mass_used.to('kg')
    DRM[skMode + '_dF_lateral'] = dF_lateral.to('N')
    DRM[skMode + '_dF_axial'] = dF_axial.to('N')

    Obs.scMass = Obs.scMass - mass_used
    DRM['scMass'] = Obs.scMass.to('kg')

    return DRM
```

In this method, a station-keeping mode is specified as an input. It then calculates station-keeping costs and uses them to update the `DRM` dictionary. The dictionary entries, assuming a specific skMode, are:

- skMode + '_dV' - or the station-keeping delta-v Δv_{sk}
- skMode + '_mass_used' - or the station-keeping mass used Δm_{sk}
- skMode + '_dF_lateral' - or the lateral differential force on the starshade ΔF_{lat}
- skMode + '_dF_axial' - or the axial differential force on the starshade ΔF_{ax}
- 'scMass' - or the occulter mass $m_{occ,new} = m_{occ} - \Delta m_{sk}$

The attribute `Observatory.scMass` is also updated by subtracting the fuel mass used during station-keeping regardless of the selected mode. `SurveySimulation.update_occulter_mass` calls on methods from the `Observatory.Prototype` module. The main method called is `Observatory.mass_dec_sk` shown below:

```
def mass_dec_sk(self, TL, sInd, currentTime, t_int):
    dF_lateral, dF_axial = self.distForces(TL, sInd, currentTime)
    intMdot, mass_used, deltaV = self.mass_dec(dF_lateral, t_int)

    return dF_lateral, dF_axial, intMdot, mass_used, deltaV
```

This method then calls on two separate methods within the `Observatory.Prototype` module. The first is `Observatory.distForces` which calculates the disturbance forces on the starshade when aligned with the LOS to target star `sInd` within target list ``TL at `currentTime`. It first calculates the position of the telescope using `Observatory.orbit` and finds the net force on the telescope $\Sigma\mathbf{F}_{tel}$ due to the Sun and Earth gravity. Next, it finds the net force on the starshade as $\Sigma\mathbf{F}_{occ}$. The net disturbance force is then

$$\Delta\mathbf{F} = \Sigma\mathbf{F}_{occ} - \frac{m_{occ}}{m_{tel}}\Sigma\mathbf{F}_{tel}$$

The method then returns two components of this disturbance force: the component lateral to the LOS ΔF_{lat} and the component axial to the LOS ΔF_{ax}.

The other method used in `Observatory.mass_dec_sk` is called `Observatory.mass_dec` which estimates fuel usage during station-keeping. The method is shown below:

```
def mass_dec(self, dF_lateral, t_int):
    intMdot = (1./np.cos(np.radians(45))*np.cos(np.radians(5))*
            dF_lateral/const.g0/self.skIsp).to('kg/s')
    mass_used = (intMdot*t_int).to('kg')
    deltaV = (dF_lateral/self.scMass*t_int).to('km/s')

    return intMdot, mass_used, deltaV
```

It only takes two inputs, the lateral disturbance force on the starshade `dF_lateral` and the integration time `t_int` or Δt_{int}. First it calculates the mass flow rate as

$$\dot{m}_{int} = \frac{\cos 5^\circ}{\cos 45^\circ} \frac{\Delta t_{int} \Delta F_{lat}}{g_0 I_{sp,sk}}$$

where the first cosine terms represent cosine losses and $I_{sp,sk}$ is the specific impulse `Observatory.skIsp` of the station-keeping engine. The fuel mass used by the starshade to station-keep throughout the integration time is then

$$\Delta m_{sk} = \dot{m}_{int} \Delta t_{int}$$

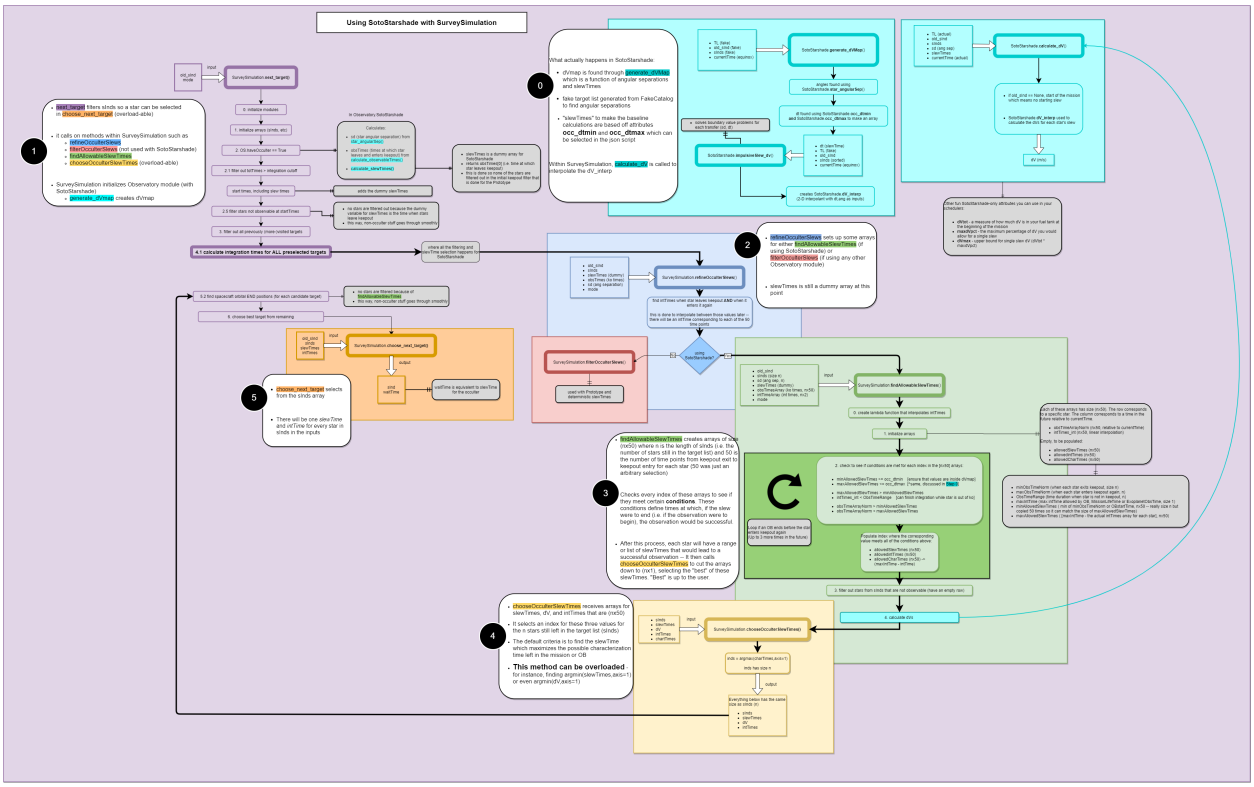
and the Δv_{sk} is

$$\Delta v_{sk} = \Delta t_{int} \frac{\Delta F_{lat}}{m_{occ}}$$

The outputs of both of these methods are combined in `Observatory.mass_dec_sk` and used in `SurveySimulation.update_occulter_mass` to update the occulter mass after using fuel during an observation.

2.10.1.2 SotoStarshade

The usage of the `Observatory.SotoStarshade` module with `SurveySimulation.Prototype` is described in the diagram below. Clicking on the diagram will open a version in which you can zoom in and see each step more clearly.



The new `SotoStarshade` module introduces a higher fidelity model for starshade slewing maneuvers. It also doesn't use fixed slew times for each star in a target list. We can then explore ranges of slew times and select them strategically around mission and keepout constraints.

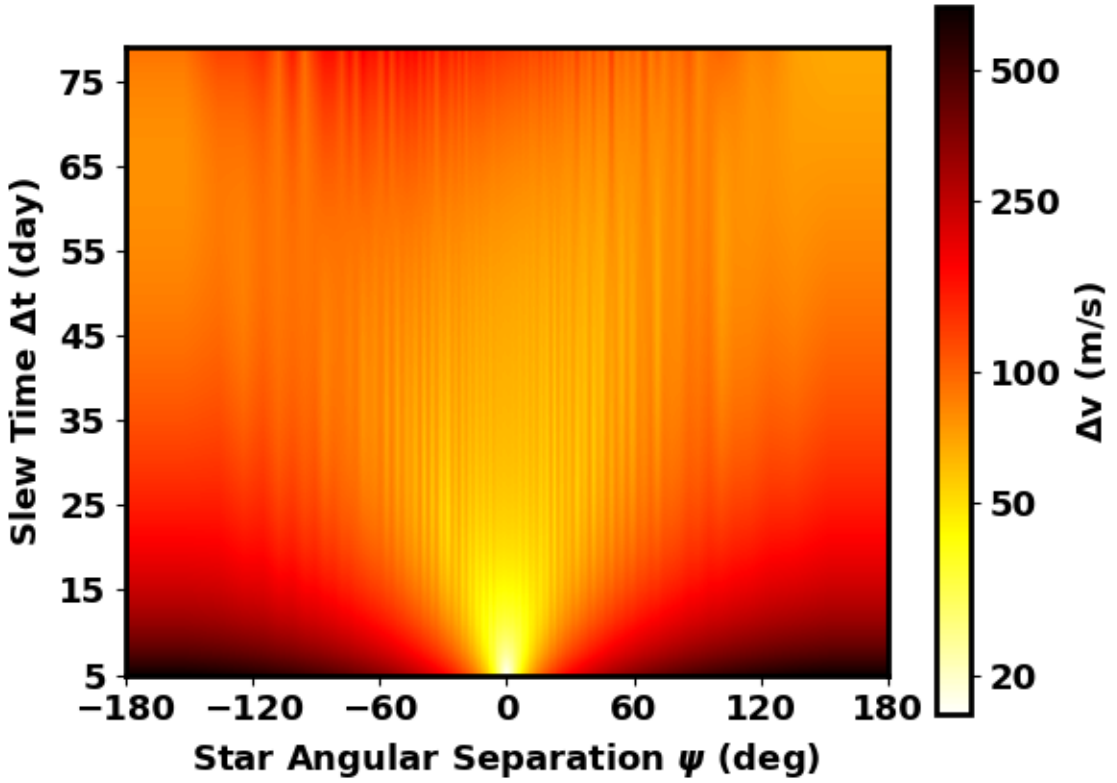
Note: The `SotoStarshade` module at the moment only overloads previous slewing methods in the `Observatory.Prototype` module. It therefore still uses the prototype station-keeping model described previously. The

SotoStarshade_SKi module contains methods for higher fidelity station-keeping costs and simulations but needs to be incorporated into SurveySimulation.

Each section of the diagram above describes different aspects of how SotoStarshade is implemented within SurveySimulation.

0 - Slew Calculations in SotoStarshade

The main idea behind the slew calculations is to generate a fuel cost map offline, prior to running simulations. We then refer to this fuel cost map to extract a fuel cost for whatever trajectory we want to know the cost of. It is therefore important to parameterize this fuel cost map in a convenient way for referencing during a simulation. We do this by selecting two parameters: ψ and Δt. These are the angular separation from a star to a reference star and the slew time, respectively. The resulting fuel cost map looks like the figure below.



The relationship between Δv and the two input parameters ψ and Δt is sufficiently continuous for creating a 2-D interpolant. More information can be found in this journal publication.

A fuel cost map is generated in the instantiantion of the SotoStarshade module in the __init__ method. The fuel cost map is generated using a "fake" catalog of stars generated with the StarCatalog.FakeCatalog module. The star catalog features stars placed in sky coordinates such that their angular separation from a reference star creates a logistic distribution.

Note: An important parameter to consider in the json script is the f_nStars parameter which specifies the number

of stars used to generate the fake catalog. Optional inputs for the `StarCatalog.FakeCatalog` that are NOT included in `SotoStarshade` at the moment are the location of the reference star `ra0` and `dec0`.

The fuel cost map is then created through the `SotoStarshade.generate_dVMap` method. In a simplified form, the method looks like this:

```
def generate_dVMap(self,TL,old_sInd,sInds,currentTime):

    # generating hash name
    filename  = 'dVMap_'
    extstr = ''
    extstr += '%s: ' % 'occulterSep'  + str(getattr(self,'occulterSep'))  + ' '
    extstr += '%s: ' % 'period_halo'  + str(getattr(self,'period_halo'))  + ' '
    extstr += '%s: ' % 'f_nStars'  + str(getattr(self,'f_nStars'))  + ' '
    extstr += '%s: ' % 'occ_dtmin'  + str(getattr(self,'occ_dtmin'))  + ' '
    extstr += '%s: ' % 'occ_dtmax'  + str(getattr(self,'occ_dtmax'))  + ' '
    ext = hashlib.md5(extstr.encode('utf-8')).hexdigest()
    filename += ext
    dVpath = os.path.join(self.cachedir, filename + '.dVmap')

    # initiating slew Times for starshade
    dt = np.arange(self.occ_dtmin.value,self.occ_dtmax.value,1)

    # angular separation of stars in target list from old_sInd
    ang =  self.star_angularSep(TL, old_sInd, sInds, currentTime)
    sInd_sorted = np.argsort(ang)
    angles  = ang[sInd_sorted].to('deg').value

    # initializing dV map
    dVMap   = np.zeros([len(dt),len(sInds)])

    #checking to see if map exists or needs to be calculated
    if os.path.exists(dVpath):
        with open(dVpath, "rb") as ff:
                A = pickle.load(ff)
        dVMap = A['dVMap']
    else:
        for i in range(len(dt)):
            dVMap[i,:] = self.impulsiveSlew_dV(dt[i],TL,old_sInd,sInd_sorted,
↪currentTime) #sorted
        B = {'dVMap':dVMap}
        with open(dVpath, 'wb') as ff:
            pickle.dump(B, ff)

    return dVMap,angles,dt
```

The method generates a hash name using several different attributes as shown. A range of slew times is created using the two attributes `occ_dtmin` and `occ_dtmax` which are specified in the json script or default to 10 and 61 days. The angular separations are then generated for all the fake stars relative to the reference star (the first entry of the target list `TL`). With the sorted angles and the slew times, the dV map is generated per slew time using the `SotoStarshade.impulsiveSlew_dV` method. This only happens if the cached file with the generated hashname is not found in the cache directory. Otherwise, the file is loaded from the cached file. The `SotoStarshade.impulsiveSlew_dV` method essentially solves boundary value problems to find the impulsive slew maneuver Δv for trajectories from the reference star to all other stars given. More information, again, can be found here.

Once the dVmap is generated, a 2-D interpolant is created within the SotoStarshade.__init__ method. An attribute called dV_interp containts the 2-D interpolant which can be referenced from the Observatory object within SurveySimulation.

Note: To facilitate referencing that 2-D interpolant, there exists a method called SotoStarshade.calculate_dV. This method should be used to extract Δv values from an input of angular separations sd and slewTimes. If there are n stars, sd should have dimension n and slewTimes should have dimensions n x m. By default, we use m = 50 within the code.

---

1 - Procedures in next_target

Here, the procedures in SurveySimulation.next_target are outlined. Some of the default procedures within this method are designed to work with the Observatory.Prototype definition of starshade slewing. That is, a single slew is selected for each target list star based on angular separation. Some changes were added to the SurveySimulation.next_target method to accomodate both implementations of starshade slews. Four new methods are incorporated in SurverySimulation.Prototype:

- SurveySimulation.refineOcculterSlews - distinguishes between using the Observatory prototype or SotoStarshade modules
- SurveySimulation.filterOcculterSlews - selected if using the prototype, runs things as normal
- SurveySimulation.findAllowableSlewTimes - selected if using SotoStarshade, searches through ranges of slew times
- SurveySimulation.chooseOcculterSlewTimes - chooses the 'best' slew time over the final ranges for each star based on some user-selected criteria

There are many filtering steps within the SurveySimulation.next_target method based on keepout, integration times, etc. We bypass these steps by overloading the calculate_slewTimes method. SotoStarshade.calculate_slewTimes, as opposed to Observatory.Prototype.calculate_slewTimes, returns a dummy array for slewTimes. In our case, it just returns the time at which each star leaves keepout so that no stars get filtered until we want to filter them. It uses the output of the Observatory.calculate_observableTimes method which returns two values for each star: the next times when the star leaves and enters keepout. This defines the start and end of an observability window. After filtering the target list using keepout, integration times, and other constraints, the SurveySimulation.choose_next_target method is called to select the next star to observe. At this stage, regardless of which Observatory module is used, each star in the target list has one associated slew time.

2 - Distinguishing Between Prototype and SotoStarshade

Here, we discuss the SurverySimulation.refineOcculterSlews method. The method is shown here:

```
def refineOcculterSlews(self, old_sInd, sInds, slewTimes, obsTimes, sd, mode):

    Obs = self.Observatory
    TL  = self.TargetList

    # initializing arrays
    obsTimeArray = np.zeros([TL.nStars,50])*u.d
    intTimeArray = np.zeros([TL.nStars,2])*u.d

    for n in sInds:
```

(continues on next page)

(continued from previous page)

```
        obsTimeArray[n,:] = np.linspace(obsTimes[0,n].value,obsTimes[1,n].value,50)*u.d
    intTimeArray[sInds,0] = self.calc_targ_intTime(sInds, Time(obsTimeArray[sInds, 0],
↪format='mjd',scale='tai'), mode)
    intTimeArray[sInds,1] = self.calc_targ_intTime(sInds, Time(obsTimeArray[sInds,-1],
↪format='mjd',scale='tai'), mode)

    # determining which scheme to use to filter slews
    obsModName = Obs.__class__.__name__

    # slew times have not been calculated/decided yet (SotoStarshade)
    if obsModName == 'SotoStarshade':
        sInds,intTimes,slewTimes,dV = self.findAllowableOcculterSlews(sInds, old_sInd,␣
↪sd[sInds], \
                                        slewTimes[sInds], obsTimeArray[sInds,:],␣
↪intTimeArray[sInds,:], mode)

    # slew times were calculated/decided beforehand (Observatory Prototype)
    else:
        sInds, intTimes, slewTimes = self.filterOcculterSlews(sInds, slewTimes[sInds], \
                                            obsTimeArray[sInds,:], intTimeArray[sInds,:],
↪ mode)
        dV = np.zeros(len(sInds))*u.m/u.s

    return sInds, slewTimes, intTimes, dV
```

One of its inputs is `obsTimes` which defines a start and end time when the star is not in keepout. It then creates a `obsTimesArray` which just creates a range of 50 times in between the `obsTimes` dates (with `n` stars, this array is `n x 50`). It also calculates two separate integration times at the start and end time of `obsTimes`. This is done to create an interpolant in a later step, estimating the varying integration time at different observation times through linear interpolation. This later step happens in `SurveySimulation.findAllowableSlewTimes`.

Note: If `Observatory.SotoStarshade` is not selected, the `SurveySimulation.refineOcculterSlews` method calls the `SurveySimulation.filterOcculterSlews` method instead. This method assumes that a single slew time has already been selected based on angular separation. It then filters targets in the target list based on whether they will be in keepout at that future slew time, what the integration times will be, etc.

---

3 - Finding Ranges of Slew Times

4 - Selecting a Slew Time for Each Target

5 - Choosing the Next Target

2.10.1.3 SotoStarshade_SKi

This is a future module that inherits the `SotoStarshade` module. It contains various methods for the simulation of station-keeping during an observation.

2.11 OpticalSystem

Optical system modules describe the science instrument and starlight suppression system, and provide methods for integration time calculation.

2.11.1 Optical System Definition

An optical system is defined by the effective collecting area and three sets of objects:

- Science Instrument(s)
- Starlight Suppression System(s)
- Observing Mode(s)

Each of these is encoded as a list of dictionaries with a minimum size of 1. A science instrument is a description of a detector and any associated optics not belonging to the starlight suppression system. A science instrument must be classified as an imager (no spectral resolution) or a spectrometer (finite spectral resolution). A starlight suppression system is a description of all of the optics responsible for producing regions of high contrast. It must be classified as a coronagraph (internal) or occulter (starshade; external). Finally, an observing mode is the combination of a starlight suppression system and a science instrument, along with a target *SNR* for all integrations performed with that observing mode.

The effective collecting area (A) is the area of the primary mirror, minus all obscurations due to the secondary (and any other obscuring optics) as well as their support structures. It is defined via three inputs:

- The primary mirror diameter (D) - in cases of non-circular mirrors, this is the major diameter
- The obscuration factor (F_o) - the fraction of the primary mirror area that is obscured
- The shape factor (F_s) - defined such that the total primary mirror area is given by $F_s D^2$. That is, for a circular mirror $F_s = \pi/4$

Given these quantities, the effective collecting area is computed as:

$$A = (1 - F_o)F_s D^2$$

Many quantities defining the optical system must be parametrizable by wavelength or angular separation (or both, or other quantities). In cases where only a single value exists at your current design stage, these must still be structured as callable (just returning the same value regardless of input). As an example, quantum efficiency is a function of wavelength for nearly all physical devices. If you have not yet selected a specific device (or do not happen to have the QE curve for the device you are modeling) your QE input can be a scalar value, but will be automatically wrapped in a lambda function that always returns your constant QE value for any inputs.

In general, each dictionary describing each of these objects can have essentially any keywords in a particular implementation. This description allows for optical system definitions to be highly flexible and extensible, but can also lead to inescapable complexity. To attempt to make the code more parsable, a few conventions are maintained, as outlined below. For all three dictionary types (`scienceInstrument`, `starlightSuppressionSystem`, and `observingMode`), missing required values will be filled from defaults, which are all inputs to the `OpticalSystem __init__` and may be modified in the *Input Specification*. The utility of this approach is that values that are expected to be the same for all instances of a dictionary (for example the number of detector pixels for all instruments using the same detector) can be set just once via the default, rather than needing to be set in each instrument definition.

2.11.1.1 Science Instruments

Each `scienceInstrument` dictionary must contain a unique `name` keyword. This string must include a substring of the form `imager` or `spectro`. For example, an optical system might contain science instruments called `imager-EMCCD` or `spectro-CCD`, describing a photon counting electron multiplying CCD imager and a mid-resolution imaging spectrometer. In cases where the same physical detector hardware is expected to be used in different modes (i.e., a single chip serving as an imager and polarizer, and integral field unit by the introduction of additional removable optics), you must still set up separate science instruments for each configuration.

Required Prototype Instrument Parameters

These values will be set from defaults if missing in the instrument dictionary. Defaults for all instruments can be set as top-level values in the *Input Specification*. These top-level values also carry their own default values in the prototype implementation.

- **name (string):**
  Instrument name (e.g. imager-EMCCD, spectro-CCD). Must contain the type of instrument ('imager' or 'spectro'). Every instrument must have a unique name.
- **QE (callable):**
  Detector quantum efficiency, parametrized by wavelength. The input value can be a scalar (fixed QE for all wavelengths) or a the full path to a file with the QE profile. The file may be a FITS file or a CSV file. FITS files must contain two rows (or columns) where the first represents wavelength and the second represents QE. The FITS header should specify the wavelength unit in keyword `UNITS` which should contain a string that is parsable as an astropy length unit (https://docs.astropy.org/en/stable/units/format.html). If the keyword does not exist, units of nm will be assumed. For CSV files, the wavelength should be in nm in a column with header `lambda` and the QE must be in a column with header `QE`. There is not required order for the two columns in a CSV file. The top-level default defaults to 0.9.
- **optics (float):**
  Attenuation due to optics specific to the science instrument. The top-level default defaults to 0.5.
- **sread (float):**
  Detector effective read noise per frame per pixel. The top-level default defaults to 1e-6.
- **idark (Quantity):**
  Detector dark-current per pixel in units of 1/time. Input values are assumed to have units of 1/second. The top-level default defaults to 1e-4.
- **texp (Quantity):**
  Exposure time per frame in units of Time. Input values are assumed to have units of seconds. The top-level default defaults to 100.
- **Rs (float):**
  (Specific to spectrometers) Spectral resolving power: $\lambda/\Delta\lambda$. The top-level default defaults to 50.
- **lenslSamp (float):**
  (Specific to spectrometers) Lenslet sampling: number of pixels per lenslet row (or column). The top-level default defaults to 2.
- **pixelNumber (int):**
  Detector array format, number of pixels per detector line/column. The top-level default defaults to 1000.
- **pixelSize (Quantity):**
  Pixel pitch in units of length. Input values are assumed to have units of meters. The top-level default defaults to 1e-5 (10 microns).

- **FoV (Quantity):**
  Angular half-field of view (i.e., field of view radius). Input values are assumed to have units of arcseconds. The `FoV` value is used only for determining the maximum outer working angle in observing modes where the starlight suppression system has an infinite *OWA* (or an OWA larger than the science instrument FoV). The top-level default defaults to 10.
- **pixelScale (Quantity):**
  Pixel scale (instantaneous field of view of each detector pixel). Input values are assumed to have units of arcseconds. The top-level default defaults to 0.02.

Field of View and Pixel Scale

Naively, the field of view and pixel scale should be related to one another. Assuming that the detector size is given by $n_{\text{pix}} \times d_{\text{pix}}$, where n_{pix} and d_{pix} are the `pixelNumber` and `pixelSize` values, respectively, then we can write:

$$\text{FoV} = 2\tan^{-1}\left(\frac{n_{\text{pix}} d_{\text{pix}}}{2f}\right)$$
$$\text{pixelScale} = 2\tan^{-1}\left(\frac{d_{\text{pix}}}{2f}\right)$$

where f is the focal length. Note that the FoV term in these equations is the full field of view (twice the `FoV` parameter). From this, we can relate the two as:

$$\text{FoV} = 2\tan^{-1}\left(n_{\text{pix}}\tan\left(\frac{\text{pixelScale}}{2}\right)\right)$$

However, the detector does not necessarily set the field of view of the imaging system (especially for some coronagraphic systems) and so these must be left as independent inputs. An example of real systems where the pixel scale and field of view are and are *not* linked are the Wide-Field Instrument (WFI) and Coronagraph Instrument (CGI) on the Roman Space Telescope, respectively (see: https://roman.ipac.caltech.edu/sims/Param_db.html). In the case of the WFI, the detectors set the field of view, whereas for the CGI, the field of view is limited by vignetting due to an aperture stop.

The field of view should always be less than or equal to the one predicted by the pixel scale. A warning will be generated if the input `FoV` is larger than the one computed from the `pixelScale`, but no errors will be raised (the warning is suppressed in cases where the two quantities are approximately equal to account for numerical errors).

Warning: If using a starshade and setting the `pixelScale` as a top-level default or per-instrument input, it is crucial to also set the field of view to its appropriate value (otherwise the original default will be used, which may be inconsistent with the input `pixelScale`).

Focal Length and f-Number

The instrument effective focal length (encoded in instrument parameter `focal`) can be specified either directly as a per-instrument value (in which case inputs are assumed to have units of meters) or as a per-instrument f-number input (encoded in instrument parameter `fnumber`). If both are given, the `fnumber` input is ignored and the value re-computed from the `focal` input. If neither input is given, both are computed from the `pixelScale` and `pixelSize` as:

$$f = \frac{d_{\text{pix}}}{2\tan\left(\text{pixelScale}/2\right)}$$

A warning will be generated if the input focal length (or equivalently f-number) do not approximately match this expression, but not errors will be raised.

Photon Counting Detectors

The prototype detector model does not account for photon-counting detectors and their particular noise sources. This is handled by implementation *Nemati*, which takes the following additional inputs (either as top-level defaults or per-instrument values):

- **CIC (float):**
  Clock-induced-charge per frame per pixel.
- **radDos (float):**
  Radiation dosage. Use of this quantity is highly specific to your particular optical system model.
- **PCeff (float):**
  Photon counting efficiency
- **ENF (float):**
  (Specific to EM-CCDs) Excess noise factor.

2.11.1.2 Starlight Suppression System

Each `starlighSuppressionSystem` dictionary must contain a unique `name` keyword identifying the starlight suppression system (coronagraph or occulter). There are no special requirements for this string (unlike the `name` in the *Science Instruments* dictionary). As with the science instruments, if you are modeling a reconfigurable coronagraph (i.e., multiple filter wheels with multiple masks) you must define a separate system for each unique configuration you wish to model. Occulters operating at multiple distances must also be set up this way.

Required Prototype Starlight Suppression System Parameters

These will be set from defaults if missing. Defaults for all systems for some parameters can be set as top-level values in the *Input Specification*.

- **name (string):**
  System name (e.g. HLC-565, SPC-660). By convention, this should also contain the central wavelength the system is optimized for (but this is not a requirement). Every system must have a unique name.
- **optics (float):**
  Attenuation due to optics specific to the coronagraph, but not captured in the various throughput values (see below). This value cannot be set as a top-level default and must be a per-system value. If missing, it defaults to 1 (no additional attenuation).

Important: Although they have the same name and purpose, the `optics` kewyord in the starlight suppression system is different from the one in the science instrument, and is not set from the top-level default. The `optics` value described here can only be set on a per-system basis.

- **lam (Quantity):**
  Central wavelength in units of length. Input values are assumed to be in nm. The top-level default defaults to 500.
- **deltaLam (Quantity):**
  Bandwidth in units of length. Input values are assumed to be in nm. This quantity has no top-level default and can only be set on a per-system basis.

- **BW (float):**
  Bandwidth fraction, such that $\lambda \times \mathrm{BW} = \Delta\lambda$. When present, `deltaLam` is used preferentially. The top-level default defaults to 0.2 (20% band).
- **ohTime (Quantity):**
  Overhead time for all integrations. Inputs are assumed to be in units of days. The top-level default defaults to 1.
- **occulter (boolean):**
  True if the system has an occulter (external or hybrid system) otherwise False (internal system). All systems have this key set to False by default. This key has no user-settable top-level default.
- **contrast_floor (float, optional):**
  An absolute limit on achievable core_contrast. Only scalar inputs (or None) are supported at this time. The top-level default defaults to None.
- **IWA (Quantity):**
  Inner working angle. Input values are assumed to be in units of arcsec. The top-level default defaults to 0.1.
- **OWA (Quantity):**
  Outer working angle. Input values are assumed to be in units of arcsec. Zero values are interpreted as infinity. Note that Python JSON also understands `Infinity` entries. The top-level default defaults to Infinity.

Warning: Any IWA/OWA values set as parameters of a starlight suppression system (or from top-level default values) will be overwritten if they disagree with angular separation ranges in table data used for other parameters. That is, if the data file used for `occ_trans` or `core_thruput`, etc., has a smallest angular separation that is larger than the currently set `IWA` or a largest separation that is smaller than the current system `OWA`, then the values will be updated to match the table data. A warning (but not error) will be generated when this happens. If no IWA/OWA inputs are supplied then both values will be set from the first data table read.

- **input_angle_units (str, optional):**
  The angle unit to assume for all starlight suppression system inputs that are angles (or are parametrized by an angular value). This also applies to data in CSV files (see below for details) and FITS files (if no superseding keyword is found in the FITS header). If None or `LAMBDA/D` or `unitless` then all angles are interpreted as λ/D values. Otherwise, this string must be parsable as an astropy length unit (see: https://docs.astropy.org/en/stable/units/format.html). The top-level default defaults to 'arcsec'. The `input_angle_units` value is used to compute the Quantity `syst["input_angle_unit_value"]` which is the value of the input angle unit in physical angle units (for λ/D inputs this uses the wavelength in `syst["lam"]`).
- **core_platescale_units (str, optional):**
  The angle unit to assume for `core_platescale`. Behaves exactly in the same way as `input_angle_units` but defaults to None (i.e., λ/D units).
- **core_platescale (float or None):**
  The pixel scale (angular extent of each pixel) for the coronagraph PSF and intensity maps. Scalar inputs are assumed to have units of `core_platescale_units`. This must be set if using `core_mean_intensity` and setting a scalar input or using a CSV file. If reading from a FITS file, the value can be encoded in the header (see below). If this value is not set at the end of populating a `starlightSuppressionSystem` dictionary and `core_mean_intensity` is not None, an error will be thrown. The top-level default defaults to None.

Important: It is crucial to differentiate between the *Science Instruments* `pixelScale` and the starlight suppression system `core_platescale`. While these might be the same value, frequently they are **not** as coronagraphs may be designed independently of the selection of the final focal-plane array. Both of these values are used to determine the

number of pixels in the photometric aperture, but these pixels represent two different things: The `pixelScale` tells you the number of detector pixels (needed for computing detector-level things like dark current and read noise). The `core_platescale` tells you the number of the pixels that the `core_mean_itensity` was computed from, which allows you to compute the total residual starlight intensity (internally called `core_intensity`) in the photometric aperture.

Note: Why do we have two separate unit inputs (one for `core_platescale` and one for everything else)? The answer, unsurprisingly, is lazy backwards compatibility. Originaly EXOSIMS forced users to convert everything to arcseconds externally to the program, *except* (confusingly) for the coronagraphic map platescale. To avoid breaking existing input scripts, the defaults for these inputs retain this behavior (but now allow the units to be specified however the user wishes).

- **csv_angsep_colname (str):**
  Column name to use for the angular separation column for CSV data for all files. This should be correlated with the `input_angle_units` value (assuming your CSV files have column names that specify units). The top-level default defaults to "r_as".
- **occ_trans (callable):**
  Intensity transmission of extended background sources such as zodiacal light, parametrized by wavelength and angular separation. This includes the pupil mask, occulter, Lyot stop and polarizer. Input values may be scalars or full paths to files containing input data to an interpolant. The top-level default defaults to 0.2.
- **core_thruput (callable):**
  System throughput in a given photometric aperture (possibly corresponding to the FWHM, set by the `core_area` value) of the planet PSF core, parametrized by wavelength and angular separation. Input values may be scalars or full paths to files containing input data to an interpolant. The top-level default defaults to 0.1.
- **core_area (callable):**
  Area of the photometric aperture used to compute `core_thruput` and `core_mean_itensity`, parametrized by wavelength and angular separation. Input values may be scalars or full paths to files containing input data to an interpolant. Input scalar values are assumed to have units of `input_angle_units`. Outputs will be Quantities with units of arcsec^2. This key has no user-settable top-level default - mssing core areas will be set as $\pi/2\,(\lambda/D)^2$ (the area of an aperture with radius $\sqrt{2}/2\,\lambda/D$).
- **core_contrast (callable):**
  System contrast = mean_intensity / PSF_peak, parametrized by wavelength and angular separation. Input values may be scalars or full paths to files containing input data to an interpolant. Default values are only populated in cases where `core_mean_intensity` is None. The top-level default defaults to 1e-10.
- **core_mean_intensity (callable):**
  Mean starlight residual normalized intensity per map pixel (i.e., per pixel of the simulated PSF). The total core intensity is computed as `core_mean_intensity` times the number of intensity map pixels in the photometric aperture (the number of pixels is determined from the `core_platescale` and `core_area` values). If `core_mean_intensity` is not specified, then the total core intensity is computed as `core_contrast` * `core_thruput`. This value is parametrized by wavelength, angular separation, and angular star diameter. The diameter value defaults to 0 arcseconds (unresolved target star). If a scalar value (or CSV file, or FITS file without the relevant header keyword) is used, a `core_platescale` input *must* be set. This key has no user-settable top-level default and can only be set on a per-system basis.

Important: The top-level default value will only be populated for `core_contrast` in the case where no `core_mean_intensity` is provided in the starlightSuppressionSystem input dictionary. If both `core_mean_intensity` and `core_contrast` are present in the input, then both will be populated, but in this case, the protoype *only* uses `core_mean_intensity` and ignores `core_contrast`. However, other implementations

may wish to utilize both (see, e.g., *Nemati_2019*). The final starlightSuppressionSystem dictionary is guaranteed to include both `core_contrast` and `core_mean_intensity` keys, but their values will be None if they were not present in the input (except for the case where neither was specified, in which case `core_contrast` will contain the lambda function based on the top-level default and `core_mean_itensity` will be None.

When using input files for `occ_trans`, `core_thruput`, `core_contrast`, `core_mean_intensity`, or `core_area`, the file may be a FITS file or a CSV file. For `occ_trans`, `core_thruput`, `core_contrast`, and `core_area`, FITS files must contain two rows (or columns) where the first represents angular separation and the second represents the parameter value. The FITS header should specify the angular separation unit in keyword `UNITS` which should either be `unitless` or `LAMBDA/D` for λ/D units or contain a string that is parsable as an astropy length unit (https://docs.astropy.org/en/stable/units/format.html). If the keyword does not exist, units of `input_angle_units` will be assumed. For CSV files, the angular separation must be in a column with header `csv_angsep_colname` and the parameter value must be in a column with a header that is exactly the same as the keyword name (i.e., `occ_trans`, etc.). There is not a required order for the columns in a CSV file, and other columns may also be present in the same file. For CSV files, the angular separation unit is set by key `input_angle_units`.

For `core_mean_intensity` all data in FITS files **must** be in columns, with the first column containing the angular separation data. When more than two columns are present, FITS files must contain keywords of the form `DIAM???` where the `???` represent a zero-padded counter starting from 000 and up to the number of columns minus 1 (i.e., for a file with 32 columns, there are 31 intensity value columns, and we expect keywords of `DIAM0000` through `DIAM030`. The values in these keywords are the stellar angular diameters in units given by the `UNITS` keyword. In cases where there are only two columns present in the file, the data will be interpreted as for a stellar diameter of zero (unresolved).

Warning: Stellar diameter-dependent `core_mean_intensity` inputs are not supported using CSV files.

CSV file inputs for `core_mean_intensity` are treated exactly in the same way as all other CSV inputs: the angular separation must be in a column with header equal to `csv_angsep_colname`, there must be a column with header `core_mean_intensity`, which will be interpreted as the zero-diameter (unresolved) core mean intensity, and angle units are set by `input_angle_units`.

For all 5 of these inputs (both for files and scalars), a lambda function will be generated, which takes as inputs the central wavelength of the observation and angular separation (both quantities with length and angle units, respectively). For `core_mean_intensity`, there is a third input, representing the stellar diameter (again a quantity with angle units), which carries a default value of 0. When the original input is a scalar value, the lambda function just returns this scalar, or zero in cases where the input angular separation is outside of the IWA/OWA range or the input wavelength is outside of the system's bandpass. When the original input is table data, the lambda function returns the value of the interpolant defined over the table data at the relevant angular separation (and, in the case of `core_mean_intensity`, the stellar diameter). For internal coronagraphs, the original table data represents angular separation for a particular wavelength (nominally the central wavelength of the system, stored in `syst[lam]`, which we'll refer to as λ_0). This is true even if the table data lists angular separation in physical angle units. If the observing wavelength does not match the system wavelength, then inputs must be scaled properly so as to evaluate the interpolant correctly. The angular separation input to the interpolant is in physical angle units: to scale, we need to convert to λ/D (for observing wavelength λ - the input to the lambda function), which means dividing by λ/D, and then convert back to a physical angle by scaling by λ_0/D. This is equivalent to defining our lambda function g as:

$$g(\lambda, s) = f\left(s\frac{\lambda_0}{\lambda}\right)$$

where $f(s)$ is the original interpolant over the table data. To validate this, consider the case where the observing wavelength is twice the system wavelength (i.e. $\lambda = 2\lambda_0$). This results in computing the interpolated value at *half* the input angular separation. This makes sense, as the input angular separation for the redder wavelength corresponds to a smaller angular separation for the original, bluer system definition. The same scaling is also applied to the stellar diameter for `core_mean_intensity`. For starshades, this scaling is *not* applied.

All of the outputs are unitless scalars, except for `core_area`, which is returned as a quantity with units of square arcseconds. For internal coronagraphs, the `core_area` output must also be scaled to account for differences in wavelength between the original system definition and the observing mode. In this case, however, we are converting an output that is in angle units (squared) corresponding to the original wavelength λ_0 to a new wavelength λ and so the scaling is the reciprocal of the one used for the input angular separation:

$$g(\lambda, s) = f\left(s\frac{\lambda_0}{\lambda}\right)\left(\frac{\lambda}{\lambda_0}\right)^2$$

Once again, this scaling is *not* applied in the case of starshades. The `core_platescale` is always a scalar, and is stored internally as a scalar quantity (not callable) with physical angular units corresponding to the central wavelength of the original system definition. It must be scaled to the observing wavelength whenever used (see *Cp_Cb_Csp_helper()* for a reference implementation).

Warning: It is up to the user to ensure that angular values in physical angle units are properly computed at the central wavelength encoded in the `lam` parameter. Note that this value will be filled in from a global default if no user input is supplied, which may result in discrepancies in the optical system definition and lead to unexpected behavior. Similarly, the prototype implementation does *not* support using systems defined with one pupil diameter at a different pupil diameter. If attempting to do this (discouraged) you must scale the angular units in your input files prior to use.

Standardized Coronagraph Parameters

Chris Stark and John Krist have a standardized definition of coronagraph parameters (described in detail here: https://starkspace.com/yield_standards.pdf) consisting of 5 FITS files. `EXOSIMS` provides a utility method (*process_opticalsys_package()*) for translating from these files to `EXOSIMS` standard input files.

The sky transmission map (coronagraph mask throughput) is radially averaged and saved to a 2D FITS file of dimension $n \times 2$:, where n is the number of angular separations computed in the radial averaging (roughly one per pixel of radius about the image center in the original data). This file can then be used for input to the `occ_trans` system parameter. An example is show in Fig. 2.26

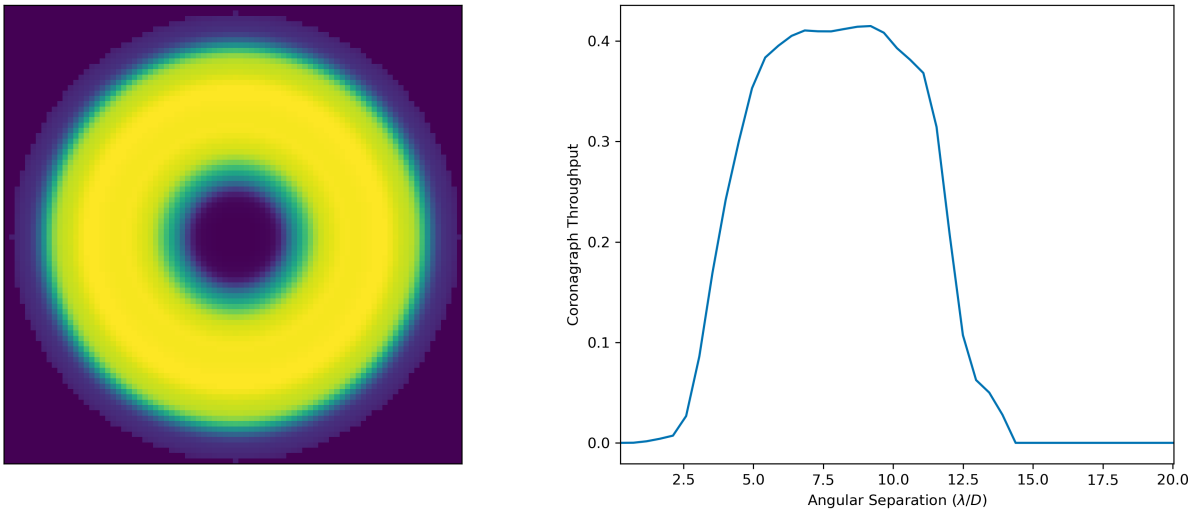


Fig. 2.26: Input sky transmission map (left) and output coronagraph throughput curve (right).

The off-axis *PSF* data is processed by finding the center of each PSF and then computing the total flux in an aperture around the center. The center is either found by computing the center of mass of an upsampled (by default by a factor of

4) copy of the input data, with an overlaid Hamming window overlaid at the location of the astrophysical PSF offset, or by fitting a 2D Gaussian to the upsampled (but non-windowed) image. In the former case, the throughput is computed in a fixed aperture (with default radius of $\sqrt{2}/2\,\lambda/D$). In the latter case, the throughput is computed within an area defined by the average of the *FWHM* values of the two axes of the fit Gaussian. It is also possible to specify a minimum photometric aperture in the case of Gaussian fits (via keyword `use_phot_aperture_as_min`).

Fig. 2.27: Input off-axis PSF data (left) and output throughput curves (right) for multiple different processing options. The black + symbol indicates the astrophysical offset of the PSF in the input data.

Fig. 2.27 shows an animation of the off-axis PSF centroiding and aperture photometry procedure and resulting throughput curves for a sample data set. The two methods implemented in *process_opticalsys_package()* (windowed center of mass and Gaussian fitting of upsampled images) are compared with quadratic centroiding and aperture photometry via the `photutils` package (https://photutils.readthedocs.io/). In all cases except for the Gaussian fit, a fixed aperture size of $\sqrt{2}/2\,\lambda/D$ is used. The Gaussian fit, in this case, typically generates a smaller FWHM measurement, resulting in a lower computed throughput. We can see that all fitting procedures fail, to varying degrees, when the PSF is partially or fully obscured by the coronagraphic masks or when it moves outside the field of view of the system. However, the 'true' throughput values in all such cases are near zero (and contrast is similarly negligible), and so these fitting errors will have no impact on simulations. The resulting throughput curve is saved to a 2D FITS file of dimension $m \times 2$, where m is the number of discrete astrophysical offsets in the original data set (i.e., the dimension of the data in the `offax_psf_offset_list` input. This file is then used as the input to the `core_thruput` parameter. In addition the area of the photometric aperture used in these computation is written out to a separate file (of the same dimensionality) to be used for the `core_area` input. In cases where a fixed aperture is used, all values of the core area file are identical, and the file can be replaced with a scalar input. However, the values will differ for Gaussian fits.

Finally, the stellar intensity data is processed by computing a radial average at each stellar diameter used. The results are written to a FITS file of dimension $k + 1 \times n$, where k is the number of stellar diameters in the original data (i.e., the dimension of the `stellar_intens_diam_list` input) and math:n is again the number of angular separations computed in the radial averaging. In cases where the image size of the stellar intensity maps is the same as that of the sky transmission map (and with the same center pixel), then these two n values should be identical. The first row of the data is the angular separations of the radial average. The stellar diameters themselves are written to the FITS header in keywords of the form `DIAM???` where the `???` represents a zero-padded number. So, if there are 31 discrete stellar diameters in the input data set, then the resulting FITS header will have keywords `DIAM000` through `DIAM030`.

Fig. 2.28: Input stellar intensity data (left) and output intensity curves (right) for various stellar diameters.

Fig. 2.28 shows an animation of the stellar intensity evolution as a function of stellar diameter.

2.11.1.3 Observing Mode

An observing mode is the combination of a science instrument with a starlight suppression system along with rules for determining integration times. The observing mode can also specify additional parameters overwriting the values in the two sub-systems. One observing mode in the optical system must be tagged as the default detection mode (by setting boolean keyword `detectionMode` to True). This is the mode used for all blind searches or initial target observations. If no observing mode is defined in the *Input Specification*, one will be created by combining the first science instrument with the first starlight suppression system.

Required Prototype Observing Mode Parameters

Some of these will be set from defaults if missing. Defaults for all modes for some parameters can be set as top-level values in the *Input Specification*.

- **instName (string):**
  Instrument name. Must match with the name of a defined Science Instrument.
- **systName (string):**
  System name. Must match with the name of a defined Starlight Suppression System.
- **detectionMode (boolean):**
  True if this observing mode is the detection mode, otherwise False. Only one detection mode can be specified. If missing, this is set to False. If, after processing all observing modes, none is set as the detection mode, the first observing mode will be made the detection mode.
- **SNR (float):**
  Signal-to-noise ratio requirement for all observations.
- **timeMultiplier (float):**
  Integration time multiplier applied for this mode. For example, if this mode requires two full rolls for every observation, the `timeMultiplier` should be set to 2. The top-level default defaults to 1.
- **lam (Quantity):**
  Central wavelength in units of length. This may be different from the *syst[lam]* value of the starlight suppression system belonging to this mode.
- **deltaLam (Quantity):**
  Bandwidth in units of length
- **BW (float):**
  Bandwidth fraction. If both `deltaLam` and `BW` are set, `deltaLam` will be used preferentially, and `BW` will be recalculated from `deltaLam` and `lam`.

If any bandpass values are not set in the `observingMode` inputs, they will be inherited from the mode's starlight suppression system. Similarly, the *IWA* and *OWA* will be copied from the starlight suppression system. However, if the OWA is larger than the instrument *FoV*, then it will be replaced with the *FoV* value.

Upon instantiation, each `ObservingMode` will define its bandpass (stored in attribute `bandpass`) as a `SpectralElement` object. The model used will be either a *`Box1D`* (default) or `Gaussian1D`, toggled by attribute `bandpass_model`. For a *`Box1D`* model, a step size can also be specified via attribute `bandpass_step` (default is 1 $\mathring{A}$).

2.11.1.4 Initialization

In order to build an optical system, the prototype `__init__` first assigns reserved inputs to attributes, and then collects all other inputs into a single attribute (*`default_vals`*), which are also copied to the *Output Specification*. It then calls three methods in sequence, as shown in Fig. 2.29.

Fig. 2.29: OpticalSystem Prototype `__init__`.

These are: *`populate_scienceInstruments()`*, *`populate_starlightSuppressionSystems()`* and *`populate_observingModes()`*, respectively. Each of these methods is responsible for populating all of the required elements of each aspect of the optical system, and copy the input values (or substituted defaults) into the *Output Specification*. Each method also calls (immediately before returning), a helper method of the same name with `_extra` appended (e.g. *`populate_scienceInstruments_extra()`*). These are there to allow overloaded implementation to expand the definitions of each optical system element, and are left blank in the prototype.

Each of the three top-level `populate_*` methods also creates an attribute called `allowed_*_kws` (e.g. `allowed_scienceInstrument_kws`, etc.). These lists contain all keywords that are allowed for that particular dictionary type for a given optical system implementation. If additional keywords are added via the `*_extra` methods, then the keyword names must be added to the relevant list as well (see *populate_scienceInstruments_extra()* for an implementation example). These lists are used to check for *Input Specification* consistency using method *check_opticalsystem_kws()*.

Important: It is up to each implementation to ensure proper handling of inputs and defaults values, to copy all new optical system elements to the `_outspec`, and to augment the allowed_*_kws lists.

Warning: When defining an optical system that inherits another implementation (rather than directly inheriting the prototype), be sure to call all levels of the `_extra` methods. That is, if the implementation you inherit has its own `populate_scienceInstruments_extra` and you wish to add to it, your method's first line should be something like `super().populate_scienceInstruments_extra()`.

2.11.2 Optical System Methods

Various different optical system models will have a variety of methods, but all optical systems are expected to provide the following:

2.11.2.1 Cp_Cb_Csp

This method computes the count rates (electrons or counts per unit time) for the planet (C_p), the background (C_b), and the residual speckle (C_{sp}). The last of these typically determines the systematic noise floor of the system. In a simple optical system model, the foreground and background rates are likely entirely independent of one another (i.e., C_b and C_{sp} have no dependence on C_p), but this is not actually a requirement. More complicated descriptions, including those of electron-multiplying CCDs run in photon counting mode, will have clock-induced-charge coupling the foreground and background counts. Given the fundamental definitions in *Photometry*, the basic elements are evaluated as follows:

- The count rate due to the star is:

$$C_{\text{star}} = F_S A \tau$$

  where F_S is the star flux density in the observing band and τ accounts for all non-coronagraphic, throughput losses. The total attenuation due to any fore-optics and any relay optics in the starlight suppression system and science instrument. This includes losses due to all reflective and transmissive elements *after* the primary, *excluding* the throughput of any coronagraphic pupil and focal plane masks. The detector *QE* is also factored into this expression, either by convolution with the bandpass used to integrate F_S, or as a scalar factor folded into τ (in which case the QE is evaluated at the bandpass central wavelength. Note that this expression represents the stellar count rate in the absence of the coronagraph (but including throughput losses due to all other optics up through the detector).

- The stellar residual count is:

$$C_{\text{speckle}} = C_{\text{star}} I_{\text{core}}$$

  where I_{core} is the coronagraph core intensity scaled by the size of the photometric aperture (this maps to the I definition from [StarkKrist2019]).

- Given a star-planet difference in magnitude Δmag in the observing band, the planet count rate is given by:

$$C_{\text{planet}} = C_{\text{star}} 10^{-0.4\Delta\text{mag}} \tau_{\text{core}}(\lambda_0, \alpha)$$

where τ_{core} is the coronagraphic core throughput, parametrized by the bandpass central wavelength (λ_0) and the angular separation of the planet (α). This maps to the term $Upsilon(x,y)$ in [StarkKrist2019]. In the absence of a specific planet spectrum, Δmag is assumed achromatic.

- Given the specific intensity of the local zodiacal light (I_{zodi}), the zodiacal light count rate is:

$$C_{\text{zodi}} = I_{\text{zodi}}\Omega\Delta\lambda\tau A\tau_{\text{occ}}$$

where Ω is the the solid angle of the photometric aperture being used and τ_{occ} is the occulter transmission. This is typically parametrized in the same way as τ_{core} and maps to the $T_{sky}(x, y)$ value as defined in [StarkKrist2019]. For further disucssion on I_{zodi}, see: *Zodiacal and Exozodiacal Light* and *ZodiacalLight*.

- Given the specific intensity of the exozodiacal light (I_{exozodi}), the exozodiacal light count rate is:

$$C_{\text{exozodi}} = I_{\text{exozodi}}\Omega\Delta\lambda\tau A\tau_{\text{core}}$$

Note that use of τ_{core} vs. τ_{sky} is a design decision for the prototype `OpticalSystem` and may be overridden by other `OpticalSystem` implementations.

- The dark current count rate is:

$$C_{\text{dark}} = n_{\text{pix}}\text{DC}$$

where n_{pix} is the number of pixels in the photometric aperture being used, while DC is the dark current rate in units of electrons/pixel/time.

- The read noise count rate is:

$$C_{\text{read}} = n_{\text{pix}}\frac{RN}{t_{\text{read}}}$$

where t_{read} is the time of each readout and RN is the read noise in units of electrons/pixel/read.

- The speckle residual is modeled as the variance of the residual starlight that cannot be removed via post-processing. This value (which is added in quadrature to the background to determine integration time) is defined as:

$$C_{\text{sp}} = C_{\text{speckle}}\text{pp}(\alpha)\text{SF}$$

where pp is the post-processing factor (defined as the reciprocal of the post-process gain, such that a reduction in speckle noise of 10x is equivalent to a pp of 0.1), parametrized by the planet's angular separation, and SF is a stability factor, used to model the overall PSF stability. Note that setting the stability factor to zero is equivalent to modeling a system with no inherent noise floor. See: *PostProcessing*.

Other detector-specific noise sources depend on the detector model and may include clock-induced charge, photon counting efficiency factors and degradation factors due to radiation dose and other effects. See: *Cp_Cb_Csp()*.

2.11.2.2 calc_intTime

Calculate the integration time required to reach the selected observing mode's target SNR on one or more targets for a planet of given Δmag at a given angular separation. If the SNR is unreachable by the selected observing mode, return NaN. See:*calc_intTime()*.

2.11.2.3 calc_dMag_per_intTime

Calculate the maximum $\Delta\mathrm{mag}$ planet observable at the observing mode's target SNR with the given integration time, at the given angular separation. This should be a strict inverse of `calc_intTime`. See: *calc_dMag_per_intTime()*.

2.11.2.4 ddMag_dt

Calculate:

$$\frac{\mathrm{d}}{\mathrm{d}t}\Delta\mathrm{mag}$$

This is used for integration time allocation optimization. See: *ddMag_dt()*.

2.12 PlanetPhysicalModel

Planet physical model modules provide methods for calculating planet properties as functions of planet parameters. These include density models (for converting between planet mass and radius) and any photometric models (for evaluating albedo/phase as a function of orbit size/location).

2.13 PlanetPopulation

Planet population modules encode the distributions defining a planet population, and provide methods for sampling from these distributions.

2.14 PostProcessing

Post-processing modules encode any processing to be carried out on observational data. This includes post-processing of image data to extract planet signals, as well as higher-level processing such as orbit fitting.

2.15 SimulatedUniverse

Simulated universe modules generate synthetic universes of real stars and simulated planets and provide methods for tracking the parameter values of these planetary systems in time.

2.16 StarCatalog

The star catalog modules are intended to be primarily static, providing only an interface between external catalog data and `EXOSIMS` standards. Any processing or augmentation of catalog data should be done in the `TargetList` module.

`StarCatalog` objects must contain equally sized arrays of stellar attributes (stored as `numpy.ndarray` attributes, or `astropy.units.quantity.Quantity` arrays, as appropriate). The prototype will generate dummy values for all attributes, with the catalog size set by input `ntargs`.

While different catalogs may contain different information, the *TargetList* will impose a minimum set of attributes (those listed in the target list's `required_catalog_atts` attribute). The contents of this list will vary depending on the particular implementation, but, for the prototype, include:

- Name: The star's name (must be unique). For Hipparcos strings referring to multiple stars, the component letter should be appended (as in CCDM, WDS, or SIMBAD).
- Vmag: Johnson V-band apparent magnitude
- BV: Johnson B-V color
- MV: Johnson V-band absolute magnitude
- BC: Bolometric correction
- L: Luminosity in solar luminosities

  Important: This must be the actual luinosity. *Not* $\log(L)$!

- coords: Target coordinates, encoded as a `astropy.coordinates.SkyCoord` array
- dist: Distances
- parx: Parallaxes
- pmra: Proper motions in RA
- pmdec: Proper motions in DEC
- rv: Proper motions along the line of sight
- Binary_Cut: Boolean array. True indicates that there is a star within 10 arcsec of the target.
- Spec: Spectral type string. Must be parsable by MeanStars.

In addition to these attributes, the prototype will also generate (all-zero) arrays for other band magnitudes. All of these have the same form as Vmag (e.g. Imag and Bmag) and include the UBRIJHK bands.

2.17 SurveySimulation

Survey simulation modules provide methods for the planning and execution of simulated surveys.

2.18 SurveyEnsemble

Survey ensemble modules facilitate the generation of ensembles of mission simulations.

2.18.1 Prototype

The *SurveyEnsemble* prototype does not provide any parallelization capability, but executes simulations in series. It is intended only to provide the basic interface specification for generating survey ensembles, and can be useful for debugging purposes where you wish the output from each simulation to be displayed as it is executed.

The method for generating an ensemble is `run_ensemble`, with inputs:

- `sim` - A *MissionSim* object
- `nb_runs` - The number of simulations to execute

- run_one - A method to execute for each simulation

There are two keywords: genNewPlanets and rewindPlanets, which are both True by default and are passed to the run_one method. All other keywords are passed directly to run_one.

In the prototype implementation, run_one cannot be overwritten and is always defined as follows:

```
def run_one(self, SS, genNewPlanets=True, rewindPlanets=True):
    SS.run_sim()
    res = SS.DRM[:]
    SS.reset_sim(genNewPlanets=genNewPlanets, rewindPlanets=rewindPlanets)
    return res
```

2.18.2 IPyParallel

This implementation uses the ipyparallel package for paralellization. This requires you to run a cluster, typically activated at the command line by `ipcluster start -n 10`, where 10 is the number of workers to create (Note: If a worker from a cluster that was not shut down properly persists when the new cluster is created, this can cause run_ipcluster_ensemble.py to hang. It is recommended to terminate all your active python sessions). Then, from an ipython command prompt, create the *MissionSim* object as usual with a script including the *IPClusterEnsemble* module:

```
import EXOSIMS, EXOSIMS.MissionSim, os.path
scriptfile = os.path.join(EXOSIMS.__path__[0],'Scripts','sampleScript_parallel_ensemble.
↪json')
sim = EXOSIMS.MissionSim.MissionSim(scriptfile)
```

Warning: It is important that you first run a version of your script with the prototype SurveyEnsemble to ensure that there are no errors and that all cached products are built on disk before execution on the cluster. The MissionSim constructor takes a nopar keyword input. Building the sim object with nopar=True will ignore any non-prototype SurveyEnsemble in your script file and build with the prototype.

The sim object will have a SurveyEnsemble attribute with a *run_ensemble* method, as described above. This method takes an argument of the run_one function, which must be defined in the main intepreter scope.

Note: Because of the requirement for run_one to be in the main scope, you cannot import a run_one method from a file, or use an object method from an instantiated object. Your easiest options are either to copy and paste your run_one method directly into the interpreter (preferably using the ipython %paste magic command), or to place the run_one method by itself in a file on disk, and then use the run command from the intepreter to load it into the main scope. In either case, run_one should be defined in the interpreter session as <function __main__.run_one>.

IPClusterEnsemble` imports the following modules to all workers in the cluster:

- EXOSIMS, EXOSIMS.util.get_module
- os, os.path
- time
- random
- cPickle
- traceback

and executes `SS = EXOSIMS.util.get_module.get_module(specs['modules']['SurveySimulation'], 'SurveySimulation')(**specs)` on each worker, generating a `SurveySimulation` object called `SS` on each worker using the parameters of your original input script. If your particular `run_one` requires additional inputs or common pre-simulation commands to be executed, then you must modify the constructor (preferably by implementing a new `SurveyEnsemble` implementation that inherits `IPClusterEnsemble`.

A simple `run_one` implementation is provided below:

```
def run_one(genNewPlanets=True, rewindPlanets=True):

    SS.run_sim()
    res = SS.DRM[:]
    SS.reset_sim(genNewPlanets=genNewPlanets, rewindPlanets=rewindPlanets)

    return res
```

Warning: This version of `run_one` returns the full `DRM` list, meaning that all outputs will need to be collected in the main scope after the ensemble execution, potentially adding considerable overheads. A better approach for large ensembles is to write each individual set of results to disk and return only a scalar value (or some other small output) to the main scope.

Once defined, the `run_one` method is executed in parallel by running:

```
res = sim.run_ensemble(N, run_one=run_one, **kwargs)
```

where `kwargs` are any kewyord arguments, or a dictionary of arguments that are passed to `run_one`.

2.18.2.1 run_ipcluster_ensemble

To simplify parallel ensemble execution via `IPClusterEnsemble`, `EXOSIMS` provides a run script called `run_ipcluster_ensemble.py` (located in the `run` directory - see: *Directory Layout*). This script is intended to be called from the command line, and is executed as:

```
>python run_ipcluster_ensemble scriptname nruns
```

where `scriptname` is the full path to the JSON script to use, and `nruns` is the number of simulations to execute. For full usage information, execute:

```
>python run_ipcluster_ensemble --help
```

This script saves the results of each individual simulation to disk as a pickle file, containing a dictionary with two keys:

- `DRM`: The full `DRM` list of dictionaries encoding the mission simulation
- `systems`: A dictionary of planet parameters generated by the `dump_systems` method of the `SimulatedUniverse` object.

In addition, the script saves the output specification (generated by `sim.genOutSpec()`) to the same directory as the rest of the results, and saves the traceback of any error generated on any worker during ensemble execution to a `log.err` file in the output directory.

2.18.2.2 read_ipcluster_ensemble

To read in and parse the pickle files generated by `run_ipcluster_ensemble` we use *EXOSIMS.util.read_ipcluster_ensemble* which provides a *gen_summary()* method. This generates lists of detection and characterization parameters for all missions in an ensemble.

2.19 TargetList

The target list modules take catalog data from a *StarCatalog* object and, using additional information and methods from *Completeness*, *OpticalSystem*, and *ZodiacalLight* objects, generate an input target list for a survey simulation. The basic functionality is fully implemented by the *TargetList* prototype, with other implementations focused on special cases, such as the *KnownRVPlanetsTargetList*. The end result is an object analogous to the *StarCatalog*, with target attributes stored in equally sized `numpy.ndarray`s. Fig. 2.30 shows the initialization of the *TargetList* prototype.

Fig. 2.30: TargetList Prototype `__init__`.

After parsing keyword inputs and instantiating objects of *StarCatalog*, *OpticalSystem*, *PostProcessing*, *ZodiacalLight*, and *Completeness*, the prototype *TargetList* initialization calls `populate_target_list()`, which makes *StarCatalog* attribute property arrays attributes of the `TargetList` object, fills in missing photometric data (if the `fillPhotometry` keyword input is set to True), and assigns each target system additional, computed attributes:

- The true and approximate stellar mass (attributes *MsEst* and *MsTrue* , respectively), calculated in *stellar_mass()*. The estimated mass is based on the Mass-Luminosity relationship from [Henry1993] and the 'true' mass is equal to the estimated mass of each star plus a randomly generated Gaussian value with mean 0 and standard deviation of 7% (the error associated with the fit in that publication).
- The inclination of the target system's orbital plane (attribute `I`), calculated in *gen_inclinations()*. This is used only if the `commonSystemInclinations` keyword input to the *SimulatedUniverse* is set to True. The inclinations are sinusoidally distributed, within the bounds set by the *PlanetPopulation* attribute `Irange`.
- The Δmag and completeness values associated with the integration cutoff time set in the *OpticalSystem* and the saturation integration time (i.e., the point at which these values stop changing). For optical systems where there is no fundamental noise floor (i.e., where *SNR* can always be increased with additional integration time) the saturation Δmag is effectively infinite, but the saturation completeness is limited to the maximum *obscurational completeness* for that system (see [Brown2005] for details). These values, along with the user-selectable $\Delta\text{mag}_{\text{int}}$ and WA_{int} are calculated in *calc_saturation_and_intCutoff_vals()*, which calls helper methods `calc_saturation_dMag()` and `calc_intCutoff_dMag()`.
- The single-visit *Completeness* (attribute *int_comp*) based on $\Delta\text{mag}_{\text{int}}$.

Finally, the whole target list is filtered by *filter_target_list()*, based on filters selected by input keywords. The default filter set removes binary stars (or stars with close companions), systems where *obscurational completeness* is zero (i.e., all planets are inside the *IWA* or outside the *OWA*), and systems for which the integration cutoff completeness is less than the `minComp` input value.

2.20 TimeKeeping

Time-keeping modules keep track of mission time and provide methods for determining when observations are allowed to be scheduled based on mission rules.

2.20.1 Mission End

The `mission_is_over` method provides functionality to determine whether a mission simulation should be stopped. The prototype implementation (*mission_is_over()*) checks for the following:

- Whether the current time (plus overheads on the next integration) exceeds the total mission lifetime
- Whether currently bookkept time for exoplanet science (plus overheads on the next integration) exceeds the total allowable science time
- If the mission uses and occulter, whether fuel has been exhausted

If a mission description supports refueling, the test on remaining fuel is also responsible for topping off the tanks to their maximum capacities (or with however much fuel remains in the external reservoir. Fig. 2.31 shows the logical flow of the prototype implementation.

2.21 ZodiacalLight

Zodiacal light modules encode models of solar system and exosystem zodiacal light.

2.22 EXOSIMS Prototype Inputs

EXOSIMS contains a large number of user-settable parameters (either via the input JSON script or passed directly to the constructors of various modules at instantiation). All inputs have associated defaults that are automatically filled in if not set by the user.

The table below includes a list of all Prototype module inputs.

Table 2.1: Prototype Arguments

Argument	Modules
allowRefueling	*Observatory*
arange	*PlanetPopulation*
bandpass_model	*OpticalSystem*
bandpass_step	*OpticalSystem*
binaryleakfilepath	*OpticalSystem*
BW	*OpticalSystem*
cachedir	*PlanetPopulation*, *PlanetPhysicalModel*, *OpticalSystem*, *ZodiacalLight*, *BackgroundSources*, *PostProcessing*, *Completeness*, *TargetList*, *SimulatedUniverse*, *Observatory*, *TimeKeeping*, *SurveySimulation*, *SurveyEnsemble*, *StarCatalog*
charMargin	*SurveySimulation*
checkInputs	*MissionSim*
checkKeepoutEnd	*Observatory*
cherryPickStars	*TargetList*
coMass	*Observatory*

continues on next page

Table 2.1 – continued from previous page

Argument	Modules
commonSystemfEZ	*ZodiacalLight*
commonSystemPlane	*SimulatedUniverse*
commonSystemPlaneParams	*SimulatedUniverse*
constrainOrbits	*PlanetPopulation*
constTOF	*Observatory*
contrast_floor	*OpticalSystem*
core_contrast	*OpticalSystem*
core_platescale	*OpticalSystem*
core_platescale_units	*OpticalSystem*
core_thruput	*OpticalSystem*
csv_angsep_colname	*OpticalSystem*
defaultAddExoplanetObsTime	*SurveySimulation*
defburnPortion	*Observatory*
dryMass	*Observatory*
dt_max	*SurveySimulation*
earths_only	*TargetList*
emission_coefficient_back	*Observatory*
emission_coefficient_front	*Observatory*
erange	*PlanetPopulation*
eta	*PlanetPopulation*
explainFiltering	*TargetList*
external_fuel_mass	*Observatory*
FAdMag0	*PostProcessing*
FAP	*PostProcessing*
fillMissingBandMags	*TargetList*
fillPhotometry	*TargetList*
filter_for_char	*TargetList*
filterBinaries	*TargetList*
find_known_RV	*SurveySimulation*
fixedPlanPerStar	*SimulatedUniverse*
forceStaticEphem	*Observatory*
FoV	*OpticalSystem*
getKnownPlanets	*TargetList*
idark	*OpticalSystem*
include_known_RV	*SurveySimulation*
input_angle_units	*OpticalSystem*
int_dMag	*TargetList*
int_WA	*TargetList*
intCutoff	*OpticalSystem*
Irange	*PlanetPopulation*
IWA	*OpticalSystem*
keepStarCatalog	*TargetList*
ko_dtStep	*Observatory*
koAngles_Earth	*OpticalSystem*
koAngles_Moon	*OpticalSystem*
koAngles_Small	*OpticalSystem*
koAngles_SolarPanel	*Observatory*
koAngles_Sun	*OpticalSystem*
lam	*OpticalSystem*

continues on next page

Table 2.1 – continued from previous page

Argument	Modules
lenslSamp	*OpticalSystem*
logfile	*MissionSim*
loglevel	*MissionSim*
lucky_planets	*SimulatedUniverse*
magEZ	*ZodiacalLight*
magZ	*ZodiacalLight*
maxdVpct	*Observatory*
MDP	*PostProcessing*
Min	*SimulatedUniverse*
minComp	*Completeness*
missionLife	*TimeKeeping*
missionPortion	*TimeKeeping*
missionSchedule	*TimeKeeping*
missionStart	*TargetList*, *TimeKeeping*
Mprange	*PlanetPopulation*
nofZ	*SurveySimulation*
nokoMap	*SurveySimulation*
non_lambertian_coefficient_back	*Observatory*
non_lambertian_coefficient_front	*Observatory*
nopar	*MissionSim*
nreflection_coefficient	*Observatory*
ntargs	*StarCatalog*
ntFlux	*SurveySimulation*
nVisitsMax	*SurveySimulation*
OBduration	*TimeKeeping*
obscurFac	*OpticalSystem*
observingModes	*OpticalSystem*
occ_dtmax	*Observatory*
occ_dtmin	*Observatory*
occ_trans	*OpticalSystem*
occulterSep	*Observatory*
ohTime	*OpticalSystem*
optics	*OpticalSystem*
Orange	*PlanetPopulation*
OWA	*OpticalSystem*
pixelNumber	*OpticalSystem*
pixelScale	*OpticalSystem*
pixelSize	*OpticalSystem*
popStars	*TargetList*
ppFact	*PostProcessing*
ppFact_char	*PostProcessing*
prange	*PlanetPopulation*
pupilDiam	*OpticalSystem*
QE	*OpticalSystem*
record_counts_path	*SurveySimulation*
Rprange	*PlanetPopulation*
Rs	*OpticalSystem*
scaleOrbits	*PlanetPopulation*
scaleWAdMag	*TargetList*

continues on next page

Table 2.1 – continued from previous page

Argument	Modules
scienceInstruments	*OpticalSystem*
scMass	*Observatory*
settlingTime	*Observatory*
shapeFac	*OpticalSystem*
sk_Tmax	*Observatory*
sk_Tmin	*Observatory*
skEff	*Observatory*
skipSaturationCalcs	*TargetList*
skIsp	*Observatory*
skMass	*Observatory*
slewEff	*Observatory*
slewIsp	*Observatory*
slewMass	*Observatory*
SNR	*OpticalSystem*
specular_reflection_factor	*Observatory*
spkpath	*Observatory*
sread	*OpticalSystem*
SRP	*Observatory*
stabilityFact	*OpticalSystem*
starlightSuppressionSystems	*OpticalSystem*
staticStars	*TargetList*
texp	*OpticalSystem*
texp_flag	*OpticalSystem*
thrust	*Observatory*
timeMultiplier	*OpticalSystem*
twotanks	*Observatory*
use_core_thruput_for_ez	*OpticalSystem*
varEZ	*ZodiacalLight*
verbose	*MissionSim*
VmagFill	*StarCatalog*
whichPlanetPhaseFunction	*PlanetPhysicalModel*
wrange	*PlanetPopulation*

2.23 Documentation Guide

This documentation is written using Sphinx.

2.23.1 Installing Sphinx

You will need the sphinx documentation tool and its numpydoc extension, napoleon extension, and mermaid extension, along with the readthedocs theme.

Assuming you have the pip python package installer, you can easily install the required tools from PyPI:

```
>>> pip install Sphinx, numpydoc, sphinxcontrib-napoleon, sphinxcontrib-mermaid, sphinx_
↪rtd_theme
```

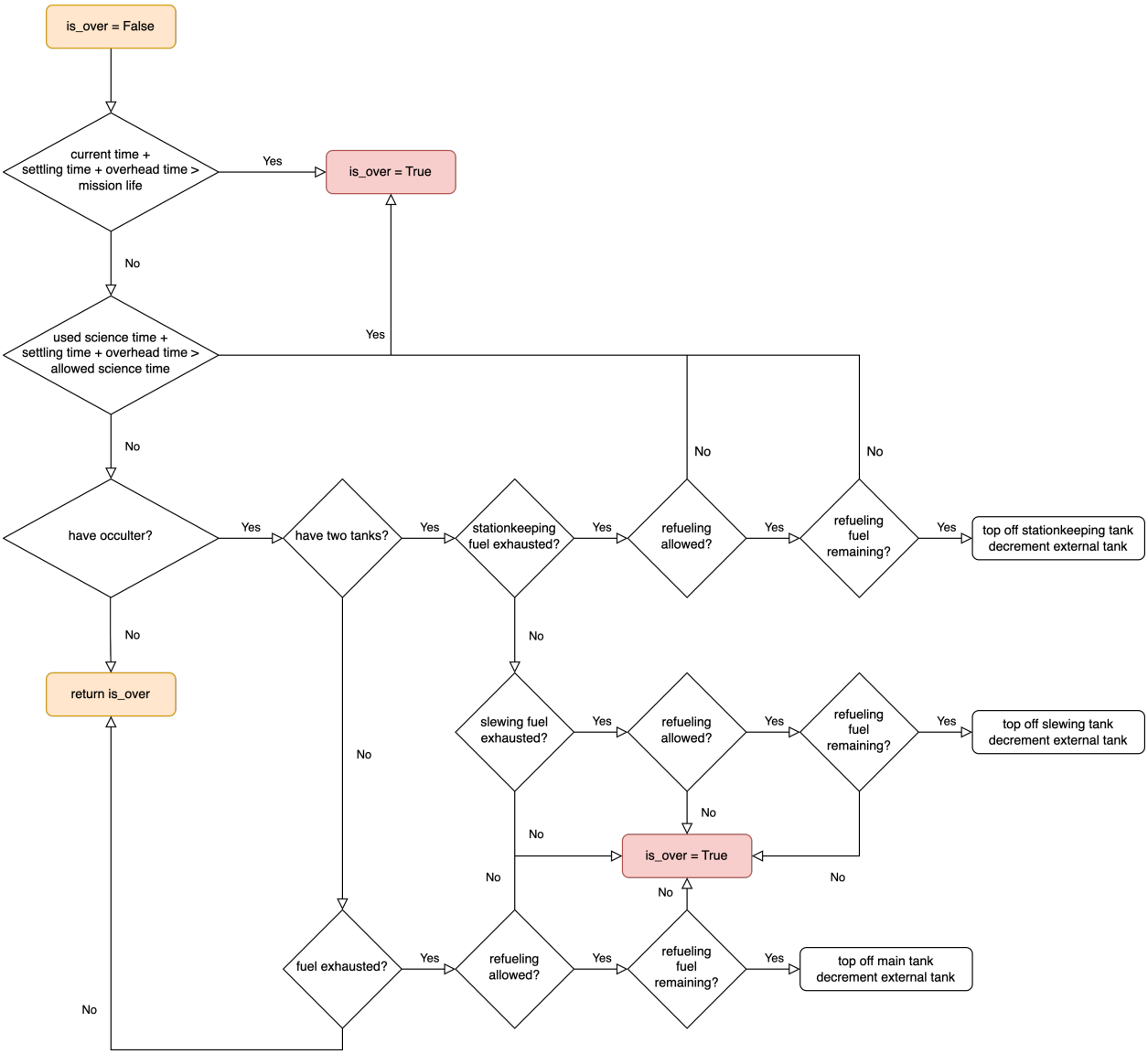


Fig. 2.31: Schematic of the logical flow of the prototype *mission_is_over()* implementation.

2.23.2 Building the docs

Navigate to the `documentation` directory (see *Directory Layout*). If Sphinx is propery installed, you should be able to just run `make html`:

```
>>> make html
[huge printout of compiling status informational messages]

Build finished. The HTML pages are in _build/html
```

The documentation will be built to HTML and you can view it locally by opening `_build/html/index.html`

You can also similarly build the documentation to PDF, LaTeX, Windows Help, or several other formats as desired. Use `make help` to see the available output formats.

2.23.3 Rebuilding from Scratch

If there are structural changes made to the EXOSIMS directory (i.e., file/folder additions or deletions) you will need to run `sphinx-apidoc`. From the `documentation` directory run:

```
>>> sphinx-apidoc -f -o . ../EXOSIMS/
```

This will rebuild all of the `EXOSIMS.*` rst files, after which you can run `make html` to rebuild the html documentation. Note that this command also generates a `modules.rst` file that we do not use, and which can be safely deleted (but should not be added to the repository).

The prototype input table (*EXOSIMS Prototype Inputs*) is auto-generated by utility `buildargdoc.py` in the `documentation` directory. It can be called as:

```
>>> python buildargdoc.py
```

This entire rebuild procedure is also packaged into a bash script for POSIX systems (`builddocs.sh`) and a batch file for Windows systems (`builddocs.bat`). Both are intended to be run from the `documentation` directory.

2.23.4 Docstrings

Valid and well-formatted docstrings are required for all EXOSIMS code. The file `docstring_example.py` in the `documentation` directory (see *Directory Layout*) provides some examples of the prefered code documentation style. Here we list the most important requirements:

2.23.4.1 Classes

All Class docstrings (occurring immediately after the class declaration) must include a description of the purpose of the class followed by a full list of arguments to the class `__init__`, in the exact order of method declaration, and then a full list of class attributes, preferably in alphabetical order. Each entry of both lists must include a well-defined type (see *Intersphinx*) and a detailed description. For arguments, keyword inputs must list their defaults, and optional inputs must be tagged as such in the type definition. An example of the preferred format is:

```
class ExampleClass():
    """An explanation of what this is for

    Args:
```

(continues on next page)

(continued from previous page)

```
        input1 (type):
            Description of argument input1
        input2 (type):
            Description of argument input2
        input3 (type):
            Description of keyword input3. Defaults to defaultval
        input4 (type, optional):
            Description of optional keyword input4. Defaults to None

    Attributes:
        attribute1 (type):
            Description of attribute 1
        attribute2 (type):
            Description of attribute 2
    """

    def __init__(
        self,
        input1,
        input2,
        input3=defaultval
        input4=None
    ):

        self.attribute1 = input1 + input2
        if input4 is not None:
            self.attribute2 = input3
        else:
            self.attribute2 = None
```

Note that the `__init__` (unlike all other methods) does not require its own docstring.

2.23.4.2 Methods

Every method (other than a class `__init__`) must have a docstring identifying the purpose of the method, its inputs and outputs. Input arguments should be formatted exactly as in the class example, above. The return is described first by its type (not a name). The type is followed by a detailed description of the returns. If a method is returning multiple objects, then the return type is a tuple. Method docstring for class methods should *not* list `self` as an argument, and should also note any attributes that are updated by the method. An example of a docstring for a method returning a single value is:

```
def example_method1(input1):
    """Does some computation on input1

    Args:
        input1 (type):
            Description of input1

    Returns:
        type:
            A description of the return
```

(continues on next page)

(continued from previous page)

```
    """

    out = some_operation_on(input1)

    return out
```

For multiple outputs:

```
def example_method2(input1):
    """Does some computation on input1

    Args:
        input1 (type):
            Description of input1

    Returns:
        tuple:
            output1 (type):
                A description of the first output
            output2 (type):
                A description of the second output
    """

    out1 = some_operation_on(input1)
    out2 = some_other_operation_on(input2)

    return out1, out2
```

2.23.4.3 Comments

Commenting your code is great! You should do as much of that as possible. However, unless your code and comment fit within 88 characters, your comment should always *precede* the code. Comments should follow the same line length limits (88 characters) as the code. Comments should use a single # symbol followed by a single space. For example:

```
# This is an inline comment about the next line:
ouptut = operation(input)
```

2.23.5 Intersphinx

EXOSIMS docs make use of intersphinx to connect to documentation for both python and other projects (in particular numpy, scipy, and astropy). In order for this to work, it is necessary to make sure that types used in docstrings are well defined. Python native types should be written as:

- str
- float
- int
- dict
- list
- bool

For third party modules, types should be written as full class strings. For example, numpy arrays should be typed as `numpy.ndarray`. Adding a leading tilde (`~numpy.ndarray`) will suppress all of the leading packages/modules (so in this case, only `ndarray`) will be written in the compiled documentation. Try to be as specific as possible about the expected contents of a variable. For example, a list of dictionaries should be typed as `list(dict)` and a numpy array of booleans should be typed as `~numpy.ndarray(bool)`. Astropy quantities should be typed as `~astropy.units.Quantity`. An astropy quantity array has type `~astropy.units.Quantity(~numpy.ndarray(float))`. The associated description of the variable should explicitly state the physical unit type unless it is obvious from the description itself (i.e., a mass is expected to have mass units, etc.). For EXOSIMS conventions, see the prototype docstrings.

EXOSIMS modules should be typed by referencing the module-specific page in this documentation. For example, if a method has a TimeKeeping object as an input, the docstring should look like:

```
def example_method2(TK):
    """Does some computation requiring TimeKeeping

    Args:
        TK (:ref:`TimeKeeping`):
            TimeKeeping object

    """
```

Similarly, `**specs` inputs should reference the *Input Specification* (by writing `:ref:`sec:inputespec`` in the docstring).

2.24 Testing

EXOSIMS provides both unit tests for component level input/output and functionality validation as well as an end-to-end test suite for full scale operational tests. Unit tests are intended to be run as part of EXOSIMS's continuous integration process, while the end to-end tests are typically deployed prior to major merges and releases to ensure continuity of baseline functionality.

2.24.1 End-to-End Testing

End-to-end testing is provided by the *e2eTests* script in the main EXOSIMS directory. This utility is intended to be executed directly (i.e. `python e2eTests.py`) and looks for test scripts in the `Scripts/TestScripts/` directory in the EXOSIMS folder hierarchy. New scripts can be added at any time and will be automatically used the next time the test suite is executed.

For each test script, `e2eTests` will:

- Create a `MissionSim` object
- Execute a simulation via `MissionSim.run_sim()`
- Reset the simulation via `MissionSim.reset_sim()`
- Execute a second simulation from the same object via `MissionSim.run_sim()`

The test suite will record a `PASS` or `FAIL` condition for each individual step, and will print a summary of results for all scripts at the end of execution.

2.24.2 Unit Testing

Unit tests are implemented using Python's `unittest` framework (see https://docs.python.org/3/library/unittest.html). The unit tests are in the `tests` directory under the EXOSIMS root directory (see *Directory Layout*). The `tests` directory contains the same folder hierarchy as the EXOSIMS directory, with a separate folder for each module type, a folder for module prototypes, a folder for utility method unit tests. Additionally, there is a `TestSupport` folder for test-specific code and input scripts, and a `TestModules` folder containing test-specific module implementations (typically for testing error condition checks in the prototypes).

There are three types of unit tests:

- In folder `ModuleName` the `test_ModuleName.py` object is used to test basic functionality of all implementations of that module (including the prototype). Test coverage in these objects is limited only to methods found the prototype, and is only executed on specifically overloaded methods in module implementations.
- In folder `ModuleName`, any `test_ImplementationName.py` object is used to test specific functionality of _only_ that module implementation. These tests are only needed to test specific expected behavior of implementations, or methods in implementations that are not overloading a prototype method.
- In folder `Prototypes`, the `test_ModuleName.py` object is used to test specific functionality of prototype implementations.

Unit tests can be executed (from the command line) by running:

```
python -m unittest discover -v
```

from the EXOSIMS root directory. This will execute all available unit tests (but not the end-to-end tests). Individual tests can be executed by running:

```
python -m unittest -v tests.ModuleName.testName
```

to run all tests within that object or:

```
python -m unittest -v tests.ModuleName.testName.testMethodName
```

to run a single individual test.

Unit-testing is performed via GitHub actions on the main repository, as configured by the `ci.yml` file. Alternatively, CircleCI can be used. CircleCI feeds tests names into `runtests.py`, which then returns:

- an XML file which CircleCI uses to split the timings
- The coverage results of the primary EXOSIMS which are sent to coveralls.io.

The basic steps to create a CircleCI testing setup for a fork of the repository are:

- Create a fork of EXOSIMS on GitHub.
- Link your GitHub account to CircleCI
- Navigate to the project page of CircleCI, and find the associated GitHub repository. Set up a new project. A config.yml sample file is provided below.
- Link up your github account to coveralls if not done already. Link EXOSIMS to coveralls. Copy the repository token.
- In the CircleCI project, navigate to "Project Settings">"Environment Variables" and add a new variable with name `COVERALLS_REPO_TOKEN` and value set to the repsitory token.

`runtests.py` can also be used standalone to create XML files and coverage file by running:

```
python runtests.py $TESTFILES
```

This will generate xml files inside of test-report and a .coverage file.

Sample `.circlci/config.yml`:

```
version: 2.1
jobs:
  test:
    parallelism: 4
    #run 4 test runs at the same time
    working_directory: ~/circleci-python
    docker:
      - image: "circleci/python:3.9.5"
    # uses python 3.9.5
    steps:
      - checkout
      - run:
          name: Grant local/bin permission
          command: sudo chown -R circleci:circleci /usr/local/bin
      - run:
          name: Grant lib/python3.9 permission
          command: sudo chown -R circleci:circleci /usr/local/lib/python3.9/site-packages
      - restore_cache:
          name: install dependencies start (install cache if one exists)
          key: -v3-{{ checksum "requirements.txt" }}
      - run: pip install --upgrade pip
      - run: pip install --upgrade -r requirements.txt
      - run: pip install coverage
      - run: pip install unittest-xml-reporting
      - run: pip install -e .
      - save_cache:
          key: -v3-{{ checksum "requirements.txt" }}
          name: install dependencies end (generate cache if one doesn't exist)
          paths:
            - ".venv"
            - "/usr/local/lib/python3.9/site-packages"
            - "/usr/local/bin"
      # build the environment. the caching steps restore the dependencies from CircleCI
↪'s servers from past runs to make things a bit faster
      - run: mkdir -p coverages
      # coverage files for each parallel run is stored in here
      - run:
          name: find and run tests and generate coverage files
          no_output_timeout: 60m
          # some tests take quite a long time to finish so the default time has to be␣
↪increased, particularly when circleci first generates the timing data for the tests
          command: |
            TESTFILES=$(circleci tests glob "tests/**/test_*.py" | circleci tests split -
↪-split-by=timings)
            python runtests.py $TESTFILES
      - store_test_results:
          path: test-reports
```

(continues on next page)

(continued from previous page)

```
      # store test results for circleci's test splitting
      - store_artifacts:
          path: test-reports
      - persist_to_workspace:
          root: coverages
          paths:
            - ./*
      # save the .coverage to the project's workspace to be combined in the done step

  done:
    working_directory: ~/circleci-python
    docker:
    - image: "circleci/python:3.9.5"
    steps:
      - checkout
      - run: pip install coverage
      - run: pip install coveralls
      - attach_workspace:
          at: ~/
      - run: coverage combine ~/
      # combine the coverage files from the workspace in the home directory
      - run: coveralls

workflows:
  test_then_upload:
    jobs:
      - test
      - done:
          requires: [test]
```

2.25 Utilities

EXOSIMS provides multiple utilities for mission construction and ensemble analysis. Some of these are accessible via methods in the `MissionSim` object, and some as standalone modules.

2.25.1 MissionSim Utilities

2.25.1.1 genWaypoint

Generates a ballpark estimate of the expected number of star visits and the total completeness of these visits for a given mission duration

Args:

duration (int):
The length of time in days allowed for the waypoint calculation, defaults to 365

tofile (string):
Name of the file containing a plot of total completeness over mission time, by default genWaypoint does not create this plot

Returns:

out (dictionary):
Output dictionary containing the number of stars visited, the total completness achieved, and the amount of time spent integrating.

genWaypoint is intended to be run prior to a simulation to provide a general idea of what to expect within the simulation. By default, genWaypoint outputs a structure that looks like:

```
{'numStars': 191, 'Total Completeness': 88.439895817937568, 'Total intTime': <Quantity␣
↪362.4756365032982 d>}
```

containing the number of stars visited, the total completeness of all the stars visited, and the total time spent integrating, which is bounded by the duration.

If "tofile" is specified, genWaypoint also generates a graph of total completeness over total integration time.

To run genWaypoint:

```
sim = EXOSIMS.MissionSim.MissionSim(scriptfile)
sim.genWaypoint(tofile="mygraph.png")

OR

sim.genWaypoint(duration=730, tofile="mygraph.png")
```

2.25.1.2 checkScript

Calls CheckScript and checks the script file against the mission outspec.

Args:

scriptfile (string):
The path to the scriptfile being used by the sim

prettyprint (boolean):
Outputs the results of Checkscript in a readable format.

tofile (string):
Name of the file containing all output specifications (outspecs). Default to None.

Returns:

out (String):
Output string containing the results of the check.

checkScript takes in a scriptfile and examines it in comparison to the mission outspec. It identifies any inconsitancies it finds between the two. The possible warnings are:

```
WARNING 1: Catches parameters that are never used in the sim or are not in the outspec
WARNING 2: Catches parameters that are unspecified in the script file and notes default␣
↪value used
WARNING 3: Catches mismatches in the modules being imported
WARNING 4: Catches cases where the value in the script file does not match the value in␣
↪the outspec
```

checkScript has several output options. By default, it will retunr a string containing all information. If "prettyprint" is specified, then checkscript will output this information to the commandline in a readble format. if "tofile" is specified, then the method will save this information to a file.

To run checkScript:

```
sim = EXOSIMS.MissionSim.MissionSim(scriptfile)
sim.checkScript(scriptfile)

OR

sim.checkScript(scriptfile, prettyprint=True)

OR

sim.checkScript(scriptfile, tofile="check.txt")
```

2.26 References

2.27 Glossary

Δmag
: Difference in magnitude between star and planet. Equal to $\log_{10}\left(\frac{F_P}{F_S}\right)$ where F_P is the planet flux and F_S is the stellar flux.

completeness
: Typically this refers to obsucrational and photometric completeness: the probability of observing a planet, belonging to a particular population, and with a particular instrument, given that one exists about a given target star, subject to both the photometric and geometric constraints of the instrument. That is, the integral over the joint probability density function of the population of Δmag and angular separation over the Δmag limits, and between the *IWA* and *OWA* of the instrument. For a full definition, see [Brown2005].

edge-on
: Orbit with 90 degree inclination. The phase angle can be any value between 0 and 180 degrees.

face-on
: Orbit with 0 inclination. The orbital radius vector always lays within the plane of the sky and the phase angle is always 90 degrees.

FWHM
: Full-Width at Half-Maximum

IWA
: Inner working angle. Minimum angular separation between an observable planet and its host star.

MCMS
: Monte Carlo Mission Simulation. See [Savransky2010].

obscurational completeness
: The probability of observing a planet, belonging to a particular population, and with a particular instrument, given that one exists about a given target star, subject only to the geometric constraints. That is, the integral over the probability density function of the population of angular separation between the *IWA* and *OWA* of the instrument.

OWA
: Outer working angle. Maximum angular separation between an observable planet and its host star.

phase angle
: The illuminant-object-observer angle. The angle between the planet-star and planet-observer vectors.

photometric completeness
The probability of observing a planet, belonging to a particular population, and with a particular instrument, given that one exists about a given target star, subject only to the photometric constraints (i.e., contrast capabilities of that instrument on that star within the allotted integration time). That is, the integral over the probability density function of the population of Δmag over the Δmag limits imposed by the instrument.

PSF
Point spread function. The impulse response of an optical system.

QE
Quantum Efficiency. The ratio of the number of charge carriers collected at the readout terminal to the number of photons hitting the detector surface.

SNR
Signal to noise ratio. Typically the estimated signal (i.e., planet flux) scaled by the variance of the estimated noise.

2.28 EXOSIMS package

2.28.1 Subpackages

2.28.1.1 EXOSIMS.BackgroundSources package

Submodules

EXOSIMS.BackgroundSources.GalaxiesFaintStars module

`class` EXOSIMS.BackgroundSources.GalaxiesFaintStars.**GalaxiesFaintStars**(***specs*)

Bases: *BackgroundSources*

GalaxiesFaintStars class

This class calculates the total number background sources in number per square arcminute, including galaxies and faint stars.

dNbackground(*coords*, *intDepths*)

Return total number counts per square arcmin

Parameters

- **coords** (*astropy SkyCoord array*) – SkyCoord object containing right ascension, declination, and distance to star of the planets of interest in units of deg, deg and pc
- **intDepths** (*float ndarray*) – Integration depths equal to the planet magnitude (Vmag+dMag), i.e. the V magnitude of the dark hole to be produced for each target. Must be of same length as coords.

Returns
Number densities of background sources for given targets in units of 1/arcmin2. Same length as inputs.

Return type
dN (astropy Quantity array)

2.28.1.2 EXOSIMS.Completeness package

Submodules

EXOSIMS.Completeness.BrownCompleteness module

class EXOSIMS.Completeness.BrownCompleteness.**BrownCompleteness**(*Nplanets=100000000.0*, ***specs*)

Bases: *Completeness*

Completeness class template

This class contains all variables and methods necessary to perform Completeness Module calculations in exoplanet mission simulation.

Parameters
specs – user specified values

Nplanets

Number of planets for initial completeness Monte Carlo simulation

Type
integer

classpath

Path on disk to Brown Completeness

Type
string

filename

Name of file where completeness interpolant is stored

Type
string

updates

Completeness values of successive observations of each star in the target list (initialized in gen_update)

Type
float nx5 ndarray

calc_fdmag(*dMag*, *smin*, *smax*)

Calculates probability density of dMag by integrating over projected separation

Parameters

- **dMag** (*float ndarray*) – Planet delta magnitude(s)
- **smin** (*float ndarray*) – Value of minimum projected separation (AU) from instrument
- **smax** (*float ndarray*) – Value of maximum projected separation (AU) from instrument

Returns
Value of probability density

Return type
float

comp_calc(*smin*, *smax*, *dMag*)

Calculates completeness for given minimum and maximum separations and dMag

Note: this method assumes scaling orbits when scaleOrbits == True has already occurred for smin, smax, dMag inputs

Parameters

- **smin** (*float ndarray*) – Minimum separation(s) in AU
- **smax** (*float ndarray*) – Maximum separation(s) in AU
- **dMag** (*float ndarray*) – Difference in brightness magnitude

Returns
Completeness values

Return type
float ndarray

comp_per_intTime(*intTimes*, *TL*, *sInds*, *fZ*, *fEZ*, *WA*, *mode*, *C_b=None*, *C_sp=None*, *TK=None*)

Calculates completeness for integration time

Parameters

- **intTimes** (*astropy Quantity array*) – Integration times
- **TL** (*TargetList module*) – TargetList class object
- **sInds** (*integer ndarray*) – Integer indices of the stars of interest
- **fZ** (*astropy Quantity array*) – Surface brightness of local zodiacal light in units of 1/arcsec2
- **fEZ** (*astropy Quantity array*) – Surface brightness of exo-zodiacal light in units of 1/arcsec2
- **WA** (*astropy Quantity*) – Working angle of the planet of interest in units of arcsec
- **mode** (*dict*) – Selected observing mode
- **C_b** (*astropy Quantity array*) – Background noise electron count rate in units of 1/s (optional)
- **C_sp** (*astropy Quantity array*) – Residual speckle spatial structure (systematic error) in units of 1/s (optional)

Returns
Completeness values

Return type
flat ndarray

completeness_setup()

Preform any preliminary calculations needed for this flavor of completeness

For BrownCompleteness this includes generating a 2D histogram of s vs. dMag for the planet population and creating interpolants over it.

completeness_update(*TL*, *sInds*, *visits*, *dt*)

Updates completeness value for stars previously observed by selecting the appropriate value from the updates array

Parameters

- **TL** (*TargetList module*) – TargetList class object
- **sInds** (*integer array*) – Indices of stars to update
- **visits** (*integer array*) – Number of visits for each star
- **dt** (*astropy Quantity array*) – Time since previous observation

Returns
Completeness values for each star

Return type
float ndarray

comps_input_reshape(*intTimes*, *TL*, *sInds*, *fZ*, *fEZ*, *WA*, *mode*, *C_b=None*, *C_sp=None*, *TK=None*)

Reshapes inputs for comp_per_intTime and dcomp_dt as necessary

Parameters

- **intTimes** (*astropy Quantity array*) – Integration times
- **TL** (*TargetList module*) – TargetList class object
- **sInds** (*integer ndarray*) – Integer indices of the stars of interest
- **fZ** (*astropy Quantity array*) – Surface bright ness of local zodiacal light in units of 1/arcsec2
- **fEZ** (*astropy Quantity array*) – Surface brightness of exo-zodiacal light in units of 1/arcsec2
- **WA** (*astropy Quantity*) – Working angle of the planet of interest in units of arcsec
- **mode** (*dict*) – Selected observing mode
- **C_b** (*astropy Quantity array*) – Background noise electron count rate in units of 1/s (optional)
- **C_sp** (*astropy Quantity array*) – Residual speckle spatial structure (systematic error) in units of 1/s (optional)

Returns

intTimes (astropy Quantity array):
Integration times

sInds (integer ndarray):
Integer indices of the stars of interest

fZ (astropy Quantity array):
Surface brightness of local zodiacal light in units of 1/arcsec2

fEZ (astropy Quantity array):
Surface brightness of exo-zodiacal light in units of 1/arcsec2

WA (astropy Quantity):
Working angle of the planet of interest in units of arcsec

smin (ndarray):
Minimum projected separations in AU

smax (ndarray):
Maximum projected separations in AU

dMag (ndarray):
Difference in brightness magnitude

Return type
tuple

dcomp_dt(*intTimes*, *TL*, *sInds*, *fZ*, *fEZ*, *WA*, *mode*, *C_b=None*, *C_sp=None*, *TK=None*)

Calculates derivative of completeness with respect to integration time

Parameters

- **intTimes** (*astropy Quantity array*) – Integration times
- **TL** (*TargetList module*) – TargetList class object
- **sInds** (*integer ndarray*) – Integer indices of the stars of interest
- **fZ** (*astropy Quantity array*) – Surface brightness of local zodiacal light in units of 1/arcsec2
- **fEZ** (*astropy Quantity array*) – Surface brightness of exo-zodiacal light in units of 1/arcsec2
- **WA** (*astropy Quantity*) – Working angle of the planet of interest in units of arcsec
- **mode** (*dict*) – Selected observing mode
- **C_b** (*astropy Quantity array*) – Background noise electron count rate in units of 1/s (optional)
- **C_sp** (*astropy Quantity array*) – Residual speckle spatial structure (systematic error) in units of 1/s (optional)

Returns
Derivative of completeness with respect to integration time (units 1/time)

Return type
astropy Quantity array

genC(*Cpath*, *nplan*, *xedges*, *yedges*, *steps*, *remainder=0*)

Gets completeness interpolant for initial completeness

This function either loads a completeness .comp file based on specified Planet Population module or performs Monte Carlo simulations to get the 2D completeness values needed for interpolation.

Parameters

- **Cpath** (*string*) – path to 2D completeness value array
- **nplan** (*float*) – number of planets used in each simulation
- **xedges** (*float ndarray*) – x edge of 2d histogram (separation)
- **yedges** (*float ndarray*) – y edge of 2d histogram (dMag)
- **steps** (*integer*) – number of nplan simulations to perform
- **remainder** (*integer*) – residual number of planets to simulate

Returns
2D numpy ndarray containing completeness probability density values

Return type
float ndarray

gen_update(*TL*)

Generates dynamic completeness values for multiple visits of each star in the target list

Parameters
TL (TargetList) – TargetList class object

generate_cache_names(***specs*)

Generate unique filenames for cached products

genplans(*nplan*)

Generates planet data needed for Monte Carlo simulation

Parameters
nplan (*integer*) – Number of planets

Returns

s (astropy Quantity array):
Planet apparent separations in units of AU

dMag (ndarray):
Difference in brightness

Return type
tuple

hist(*nplan*, *xedges*, *yedges*)

Returns completeness histogram for Monte Carlo simulation

This function uses the inherited Planet Population module.

Parameters

- **nplan** (*float*) – number of planets used
- **xedges** (*float ndarray*) – x edge of 2d histogram (separation)
- **yedges** (*float ndarray*) – y edge of 2d histogram (dMag)

Returns
2D numpy ndarray containing completeness frequencies

Return type
float ndarray

target_completeness(*TL*)

Generates completeness values for target stars using average case values

This method is called from TargetList __init__ method.

Parameters
TL (*TargetList module*) – TargetList class object

Returns
Completeness values for each target star

Return type
float ndarray

EXOSIMS.Completeness.GarrettCompleteness module

class EXOSIMS.Completeness.GarrettCompleteness.**GarrettCompleteness**(*order_of_quadrature=15*, ***specs*)

Bases: *BrownCompleteness*

Analytical Completeness class

This class contains all variables and methods necessary to perform Completeness Module calculations based on Garrett and Savransky 2016 in exoplanet mission simulation.

The completeness calculations performed by this method assume that all planetary parameters are independently distributed. The probability density functions used here are either independent or marginalized from a joint probability density function.

Parameters

- **order_of_quadrature** (*int*) – The order of quadrature used in the comp_dmag function's fixed quad integration. Higher values will give marginal improvements in the comp_calc completeness values, but are slower.
- ****specs** – *Input Specification*

updates

Completeness values of successive observations of each star in the target list (initialized in gen_update)

Type
nx5 ndarray

Jac(*b*)

Calculates determinant of the Jacobian transformation matrix to get the joint probability density of dMag and s

Parameters
b (*ndarray*) – Phase angles

Returns
Determinant of Jacobian transformation matrix

Return type
f (ndarray)

Rgrand(*R*, *z*)

Calculates integrand for determining probability density of albedo times planetary radius squared

Parameters

- **R** (*ndarray*) – Values of planetary radius
- **z** (*float*) – Value of albedo times planetary radius squared

Returns
Values of integrand

Return type
f (ndarray)

calc_fdmag(*dMag*, *smin*, *smax*)

Calculates probability density of dMag by integrating over projected separation

Parameters

- **dMag** (*float ndarray*) – Planet delta magnitude(s)

- **smin** (*float ndarray*) – Value of minimum projected separation (AU) from instrument
- **smax** (*float ndarray*) – Value of maximum projected separation (AU) from instrument

Returns
Value of probability density

Return type
f (float)

comp_calc(*smin*, *smax*, *dMag*)

Calculates completeness for given minimum and maximum separations and dMag

Note: this method assumes scaling orbits when scaleOrbits == True has already occurred for smin, smax, dMag inputs

Parameters

- **smin** (*float ndarray*) – Minimum separation(s) in AU
- **smax** (*float ndarray*) – Maximum separation(s) in AU
- **dMag** (*float ndarray*) – Difference in brightness magnitude

Returns
Completeness value(s)

Return type
comp (float ndarray)

comp_dmag(*smin*, *smax*, *max_dMag*)

Calculates completeness by first integrating over projected separation and then dMag.

Parameters

- **smin** (*ndarray*) – Values of minimum projected separation (AU) from instrument
- **smax** (*ndarray*) – Value of maximum projected separation (AU) from instrument
- **max_dMag** (*float ndarray*) – Maximum planet delta magnitude

Returns
Completeness values

Return type
comp (ndarray)

comp_s(*smin*, *smax*, *dMag*)

Calculates completeness by first integrating over dMag and then projected separation.

Parameters

- **smin** (*ndarray*) – Values of minimum projected separation (AU) from instrument
- **smax** (*ndarray*) – Value of maximum projected separation (AU) from instrument
- **dMag** (*ndarray*) – Planet delta magnitude

Returns
Completeness values

Return type
comp (ndarray)

completeness_setup()

Preform any preliminary calculations needed for this flavor of completeness

For GarrettCompleteness this involves setting up various interpolants. See [Garrett2016] for details.

f_dmag()

f_dmags()

f_dmagsRp(*Rp*, *dmag*, *s*)

Calculates the joint probability density of planetary radius, dMag, and projected separation

Parameters

- **Rp** (*ndarray*) – Values of planetary radius
- **dmag** (*float*) – Planet delta magnitude
- **s** (*float*) – Value of projected separation

Returns

Values of joint probability density

Return type

f (ndarray)

f_dmagsz(*z*, *dmag*, *s*)

Calculates the joint probability density of albedo times planetary radius squared, dMag, and projected separation

Parameters

- **z** (*ndarray*) – Values of albedo times planetary radius squared
- **dmag** (*float*) – Planet delta magnitude
- **s** (*float*) – Value of projected separation

Returns

Values of joint probability density

Return type

f (ndarray)

f_r(*r*)

Calculates the probability density of orbital radius

Parameters

r (*float*) – Value of semi-major axis in AU

Returns

Value of probability density

Return type

f (float)

f_s()

f_sdmag(*s*, *dmag*)

Calculates the joint probability density of projected separation and dMag by flipping the order of f_dmags

Parameters

- **s** (*float*) – Value of projected separation (AU)

- **dmag** (*float*) – Planet delta magnitude

Returns
Value of joint probability density

Return type
f (float)

f_z(*z*)

Calculates probability density of albedo times planetary radius squared

Parameters
z (*float*) – Value of albedo times planetary radius squared

Returns
Probability density

Return type
f (float)

genComp(*Cpath*, *TL*)

Generates function to get completeness values

Parameters

- **Cpath** (*str*) – Path to pickled dictionary containing interpolant function
- **TL** (*TargetList module*) – TargetList class object

Returns
Marginalized to self.mode_dMag probability density function for projected separation

Return type
dist_s (callable(s))

maxdmag(*s*)

Calculates the maximum value of dMag for projected separation

Parameters
s (*float*) – Projected separation (AU)

Returns
Maximum planet delta magnitude

Return type
maxdmag (float)

mindmag(*s*)

Calculates the minimum value of dMag for projected separation

Parameters
s (*float*) – Projected separations (AU)

Returns
Minimum planet delta magnitude

Return type
mindmag (float)

rgrand1(*e*, *a*, *r*)

Calculates first integrand for determinining probability density of orbital radius

Parameters

- **e** (*ndarray*) – Values of eccentricity
- **a** (*float*) – Values of semi-major axis in AU
- **r** (*float*) – Values of orbital radius in AU

Returns
Values of first integrand

Return type
f (ndarray)

rgrand2(*a*, *r*)

Calculates second integrand for determining probability density of orbital radius

Parameters

- **a** (*float*) – Value of semi-major axis in AU
- **r** (*float*) – Value of orbital radius in AU

Returns
Value of second integrand

Return type
f (float)

rgrandac(*e*, *a*, *r*)

Calculates integrand for determining probability density of orbital radius when semi-major axis is constant

Parameters

- **e** (*ndarray*) – Values of eccentricity
- **a** (*float*) – Value of semi-major axis in AU
- **r** (*float*) – Value of orbital radius in AU

Returns
Value of integrand

Return type
f (ndarray)

rgrandec(*a*, *e*, *r*)

Calculates integrand for determining probability density of orbital radius when eccentricity is constant

Parameters

- **a** (*ndarray*) – Values of semi-major axis in AU
- **e** (*float*) – Value of eccentricity
- **r** (*float*) – Value of orbital radius in AU

Returns
Value of integrand

Return type
f (float)

s_bound(*dmag*, *smax*)

Calculates the bounding value of projected separation for dMag

Parameters

- **dmag** (*float*) – Planet delta magnitude
- **smax** (*float*) – maximum projected separation (AU)

Returns
boundary value of projected separation (AU)

Return type
sb (float)

`target_completeness`(*TL*)

Generates completeness values for target stars

This method is called from TargetList __init__ method.

Parameters
TL (*TargetList module*) – TargetList class object

Returns
int_comp: 1D numpy array of completeness values for each target star

Return type
ndarray(float))

EXOSIMS.Completeness.IntegrationTimeAdjustedCompleteness module

`class` EXOSIMS.Completeness.IntegrationTimeAdjustedCompleteness.**IntegrationTimeAdjustedCompleteness**(*Nplar*
***spe*

Bases: *SubtypeCompleteness*

Integration time-adjusted Completeness. See [Keithly2021b]

Parameters

- **Nplanets** (*int*) – number of planets to simulate in IAC
- **specs** – user specified values

`Nplanets`

Number of planets for initial completeness Monte Carlo simulation

Type
int

`classpath`

Path on disk to Brown Completeness

Type
str

`filename`

Name of file where completeness interpolant is stored

Type
str

`updates`

Completeness values of successive observations of each star in the target list (initialized in gen_update)

Type
float nx5 numpy.ndarray

comp_calc(*smin*, *smax*, *dMag*, *subpop=-2*, *tmax=0.0*, *starMass=<<class 'astropy.constants.iau2015.IAU2015'> name='Solar mass' value=1.988409870698051e+30 uncertainty=4.468805426856864e+25 unit='kg' reference='IAU 2015 Resolution B 3 + CODATA 2018'>*, *IACbool=False*)

Calculates completeness for given minimum and maximum separations and dMag

Parameters

- **smin** (*float numpy.ndarray*) – Minimum separation(s) in AU
- **smax** (*float numpy.ndarray*) – Maximum separation(s) in AU
- **dMag** (*float numpy.ndarray*) – Difference in brightness magnitude
- **subpop** (*int*) – planet subtype to use for calculation of int_comp -2 - planet population -1 - earthLike population (i,j) - kopparapu planet subtypes
- **tmax** (*float*) – the integration time of the observation in days
- **starMass** (*float*) – star mass in units of M_sun
- **IACbool** (*bool*) – Use integration time-adjusted completeness or normal Brown completeness If False, tmax does nothing. Defaults False.

Returns

comp, SubtypeCompleteness Completeness values (Brown's method mixed with classification) or integration time adjusted completeness totalCompleteness_maxIntTimeCorrected

Return type

numpy.ndarray

..note:

```
This method assumes scaling orbits when scaleOrbits == True has already
occurred for smin, smax, dMag inputs
```

EXOSIMS.Completeness.SubtypeCompleteness module

class EXOSIMS.Completeness.SubtypeCompleteness.**SubtypeCompleteness**(*binTypes='kopparapuBins_extended'*, ***specs*)

Bases: *BrownCompleteness*

Completeness by planet subtype

Parameters

- **binTypes** (*str*) – string specifying the kopparapuBin Types to use
- ****specs** – *Input Specification*

Nplanets

Number of planets for initial completeness Monte Carlo simulation

Type

int

classpath

Path on disk to Brown Completeness

Type

string

`filename`

Name of file where completeness interpolant is stored

Type
str

`updates`

Completeness values of successive observations of each star in the target list (initialized in gen_update)

Type
float nx5 numpy.ndarray

`binTypes`

string specifying the kopparapuBin Types to use

Type
str

`SubtypeHist`(*nplan*, *xedges*, *yedges*, *TL*)

Returns completeness histogram for Monte Carlo simulation

This function uses the inherited Planet Population module.

Parameters

- **nplan** (*float*) – number of planets used
- **xedges** (*float ndarray*) – x edge of 2d histogram (separation)
- **yedges** (*float ndarray*) – y edge of 2d histogram (dMag)
- **TL** (*target list object*) –

Returns

float (numpy.ndarray):
2D numpy ndarray containing completeness frequencies

hs (dict):
dict with index [bini,binj] containing arrays of counts per Array bin

bini (float):
planet size type index

binj (float):
planet incident stellar flux index

h_earthLike (float):
number of earth like exoplanets

h (numpy.ndarray):
2D numpy array of bin counts over all dmag vs s

xedges (numpy.ndarray):
array of bin edges originally input and used in histograms

yedges (numpy.ndarray):
array of bin edges originally input and used in histograms

counts (dict):
dict with index [bini,binj] containing total number of planets in bini,binj

count (float):
total number of planets in the whole populatiom

count_earthLike (float):
total number of earth-like planets in the whole population

Return type
tuple

`calc_fdmag`(*dMag*, *smin*, *smax*, *subpop=-2*)

Calculates probability density of dMag by integrating over projected separation

Parameters

- **dMag** (*float ndarray*) – Planet delta magnitude(s)
- **smin** (*float ndarray*) – Value of minimum projected separation (AU) from instrument
- **smax** (*float ndarray*) – Value of maximum projected separation (AU) from instrument
- **subpop** (*int*) – planet subtype to use for calculation of int_comp -2 - planet population -1 - earthLike population (i,j) - kopparapu planet subtypes

Returns
Value of probability density

Return type
float

`classifyEarthlikePlanets`(*Rp*, *TL*, *starind*, *sma*, *ej*)

Determine Kopparapu bin of an individual planet.

Verified with Kopparapu Extended

Parameters

- **Rp** (*float*) – planet radius in Earth Radii
- **TL** (*object*) – EXOSIMS target list object
- **sma** (*float*) – planet semi-major axis in AU
- **ej** (*float*) – planet eccentricity

Returns

bini (int):
planet size-type: 0-Smaller than Earthlike, 1- Earthlike, 2- Larger than Earth-like

binj (int):
planet incident stellar-flux: 0- lower than Earthlike, 1- flux of Earthlike, 2- higher flux than Earth-like

Return type
tuple

`classifyPlanet`(*Rp*, *TL*, *starind*, *sma*, *ej*)

Determine Kopparapu bin of an individual planet

Parameters

- **Rp** (*float*) – planet radius in Earth Radii
- **TL** (*object*) – EXOSIMS target list object
- **sma** (*float*) – planet semi-major axis in AU
- **ej** (*float*) – planet eccentricity

Returns

bini (int):
planet size-type: 0-rocky, 1- Super-Earths, 2- sub-Neptunes, 3- sub-Jovians, 4- Jovians

binj (int):
planet incident stellar-flux: 0- hot, 1- warm, 2- cold

earthLike (bool):
boolean indicating whether the planet is earthLike or not earthLike

Return type
tuple

`classifyPlanets`(*Rp*, *TL*, *starind*, *sma*, *ej*)

Determine Kopparapu bin of an individual planet.

Verified with Kopparapu Extended

Parameters

- **Rp** (*float*) – planet radius in Earth Radii
- **TL** (*object*) – EXOSIMS target list object
- **sma** (*float*) – planet semi-major axis in AU
- **ej** (*float*) – planet eccentricity

Returns

bini (int):
planet size-type: 0-rocky, 1- Super-Earths, 2- sub-Neptunes, 3- sub-Jovians, 4- Jovians

binj (int):
planet incident stellar-flux: 0- hot, 1- warm, 2- cold

earthLike (bool):
boolean indicating whether the planet is earthLike or not earthLike

Return type
tuple

`comp_calc`(*smin*, *smax*, *dMag*, *subpop=-2*)

Calculates completeness for given minimum and maximum separations and dMag

Note: this method assumes scaling orbits when scaleOrbits == True has already occurred for smin, smax, dMag inputs

Parameters

- **smin** (*float ndarray*) – Minimum separation(s) in AU
- **smax** (*float ndarray*) – Maximum separation(s) in AU
- **dMag** (*float ndarray*) – Difference in brightness magnitude
- **subpop** (*int*) – planet subtype to use for calculation of int_comp -2 - planet population -1 - earthLike population (i,j) - kopparapu planet subtypes

Returns
Completeness values

Return type
float ndarray

comp_calc2(*smin*, *smax*, *dMag_min*, *dMag_max*, *subpop=-2*)

Calculates completeness for given minimum and maximum separations and dMag

Note: this method assumes scaling orbits when scaleOrbits == True has already occurred for smin, smax, dMag inputs

Parameters

- **smin** (*float ndarray*) – Minimum separation(s) in AU
- **smax** (*float ndarray*) – Maximum separation(s) in AU
- **dMag_min** (*float ndarray*) – Minimum difference in brightness magnitude
- **dMag_max** (*float ndarray*) – Maximum difference in brightness magnitude
- **subpop** (*int*) – planet subtype to use for calculation of int_comp -2 - planet population -1 - earthLike population (i,j) - kopparapu planet subtypes

Returns
Completeness values

Return type
float ndarray

comp_per_intTime(*intTimes*, *TL*, *sInds*, *fZ*, *fEZ*, *WA*, *mode*, *C_b=None*, *C_sp=None*, *TK=None*)

Calculates completeness for integration time

Parameters

- **intTimes** (*astropy Quantity array*) – Integration times
- **TL** (*TargetList module*) – TargetList class object
- **sInds** (*integer ndarray*) – Integer indices of the stars of interest
- **fZ** (*astropy Quantity array*) – Surface brightness of local zodiacal light in units of 1/arcsec2
- **fEZ** (*astropy Quantity array*) – Surface brightness of exo-zodiacal light in units of 1/arcsec2
- **WA** (*astropy Quantity*) – Working angle of the planet of interest in units of arcsec
- **mode** (*dict*) – Selected observing mode
- **C_b** (*astropy Quantity array*) – Background noise electron count rate in units of 1/s (optional)
- **C_sp** (*astropy Quantity array*) – Residual speckle spatial structure (systematic error) in units of 1/s (optional)
- **TK** (*Timekeeping object*) – timekeeping object for compatability with SLSQPScheduler

Returns
Completeness values

Return type
flat ndarray

completeness_setup()

Preform any preliminary calculations needed for this flavor of completeness

For SubtypeCompleteness this generates comleteness by planet bin

comps_input_reshape(*intTimes*, *TL*, *sInds*, *fZ*, *fEZ*, *WA*, *mode*, *C_b=None*, *C_sp=None*, *TK=None*)

Reshapes inputs for comp_per_intTime and dcomp_dt as necessary

Parameters

- **intTimes** (*astropy Quantity array*) – Integration times
- **TL** (*TargetList module*) – TargetList class object
- **sInds** (*integer ndarray*) – Integer indices of the stars of interest
- **fZ** (*astropy Quantity array*) – Surface bright ness of local zodiacal light in units of 1/arcsec2
- **fEZ** (*astropy Quantity array*) – Surface brightness of exo-zodiacal light in units of 1/arcsec2
- **WA** (*astropy Quantity*) – Working angle of the planet of interest in units of arcsec
- **mode** (*dict*) – Selected observing mode
- **C_b** (*astropy Quantity array*) – Background noise electron count rate in units of 1/s (optional)
- **C_sp** (*astropy Quantity array*) – Residual speckle spatial structure (systematic error) in units of 1/s (optional)

Returns

intTimes (astropy Quantity array):
Integration times

sInds (integer ndarray):
Integer indices of the stars of interest

fZ (astropy Quantity array):
Surface brightness of local zodiacal light in units of 1/arcsec2

fEZ (astropy Quantity array):
Surface brightness of exo-zodiacal light in units of 1/arcsec2

WA (astropy Quantity):
Working angle of the planet of interest in units of arcsec

smin (ndarray):
Minimum projected separations in AU

smax (ndarray):
Maximum projected separations in AU

dMag (ndarray):
Difference in brightness magnitude

Return type
tuple

dcomp_dt(*intTimes*, *TL*, *sInds*, *fZ*, *fEZ*, *WA*, *mode*, *C_b=None*, *C_sp=None*, *TK=None*)

Calculates derivative of completeness with respect to integration time

Parameters

- **intTimes** (*astropy Quantity array*) – Integration times
- **TL** (*TargetList module*) – TargetList class object
- **sInds** (*integer ndarray*) – Integer indices of the stars of interest

- **fZ** (*astropy Quantity array*) – Surface brightness of local zodiacal light in units of 1/arcsec2
- **fEZ** (*astropy Quantity array*) – Surface brightness of exo-zodiacal light in units of 1/arcsec2
- **WA** (*astropy Quantity*) – Working angle of the planet of interest in units of arcsec
- **mode** (*dict*) – Selected observing mode
- **C_b** (*astropy Quantity array*) – Background noise electron count rate in units of 1/s (optional)
- **C_sp** (*astropy Quantity array*) – Residual speckle spatial structure (systematic error) in units of 1/s (optional)
- **TK** (*Timekeeping object*) – timekeeping object for compatability with SLSQPScheduler

Returns
Derivative of completeness with respect to integration time (units 1/time)

Return type
astropy Quantity array

dmag_limits(*rmin*, *rmax*, *pmax*, *pmin*, *Rmax*, *Rmin*, *phaseFunc*)

Limits of delta magnitude as a function of separation

Limits on dmag vs s JPDF from [Garrett2016] See https://github.com/dgarrett622/FuncComp/blob/master/FuncComp/util.py

Parameters

- **rmin** (*float*) – minimum planet-star distance possible in AU
- **rmax** (*float*) – maximum planet-star distance possible in AU
- **pmax** (*float*) – maximum planet albedo
- **Rmax** (*float*) – maximum planet radius in earthRad
- **phaseFunc** (*callable*) – with input in units of rad

Returns

dmag_limit_functions (list):
list of lambda functions taking in 's' with units of AU

lower_limits (list):
list of floats representing lower bounds on 's'

upper_limits (list):
list of floats representing upper bounds on 's'

Return type
tuple

genSubtypeC(*Cpath*, *nplan*, *xedges*, *yedges*, *steps*, *TL*)

Gets completeness interpolant for initial completeness

This function either loads a completeness .comp file based on specified Planet Population module or performs Monte Carlo simulations to get the 2D completeness values needed for interpolation.

Parameters

- **Cpath** (*string*) – path to 2D completeness value array

- **nplan** (*float*) – number of planets used in each simulation
- **xedges** (*float ndarray*) – x edge of 2d histogram (separation)
- **yedges** (*float ndarray*) – y edge of 2d histogram (dMag)
- **steps** (*integer*) – number of simulations to perform TL (target list object):

Returns
2D numpy ndarray containing completeness probability density values

Return type
float ndarray

`gen_update`(*TL*)

Generates dynamic completeness values for multiple visits of each star in the target list

Parameters
TL (TargetList) – TargetList class object

`genplans`(*nplan*, *TL*)

Generates planet data needed for Monte Carlo simulation

Parameters

- **nplan** (*integer*) –
- **TL** (*target list object*) –

Returns

s (astropy Quantity array):
Planet apparent separations in units of AU

dMag (ndarray):
Difference in brightness

bini (int):
planet size-type: 0-rocky, 1- Super-Earths, 2- sub-Neptunes, 3- sub-Jovians, 4- Jovians

binj (int):
planet incident stellar-flux: 0- hot, 1- warm, 2- cold

earthLike (bool):
boolean indicating whether the planet is earthLike or not earthLike

Return type
tuple

`kopparapuBins`()

A function containing the Center 15 Kopparapu bins Updates the Rp_bins, Rp_lo, Rp_hi, L_bins, L_lo, and L_hi attributes

`kopparapuBins_extended`()

A function containing the Full 35 Kopparapu bins Updates the Rp_bins, Rp_lo, Rp_hi, L_bins, L_lo, L_hi, and type_names attribute

`kopparapuBins_old`()

A function containing the Inner 12 Kopparapu bins Updates the Rp_bins, Rp_lo, Rp_hi, L_bins, L_lo, and L_hi attributes

probDetectionIsOfType(*dmag*, *uncertainty_dmag*, *separation*, *uncertainty_s*, *sub=-2*)

Calculates the probability a planet is of the given type

Parameters

- **comp** (*completeness object*) – a completeness object with the EVPOC_hs, count_hs, count_pop, EVPOCpdf_pop attributes (generated by the subtype completeness module)
- **dmag** (*float*) – the mean dmag to evaluate at
- **()** (*uncertainty_s*) – the uncertainty in dmag to evaluate over
- **()** – the mean separation to evaluate at
- **()** – the uncertainty in separation to evaluate over
- **subpop** (*int*) – planet subtype to use for calculation of int_comp -2 - planet population -1 - earthLike population [i,j] - kopparapu planet subtypes

Returns

prob (float):
a float indicating the probability a planet is both from the given sub-population and the instrument probability density function.

normProbProp (float):
Probability normalized by the population density joint probability density function

Return type
tuple

putPlanetsInBoxes(*out*, *TL*)

Classifies planets in a gen_summary out file by their hot/warm/cold and rocky/superearth/subneptune/subjovian/jovian bins

Parameters

- **out** (*dict*) – a gen_summary output dict
- **TL** (*TargetList object*) – a target list object

Returns

aggbins (list):
dims [# simulations, 5x3 numpy array]

earthLikeBins (list):
dims [# simulations]

Return type
tuple

target_completeness(*TL*, *calc_char_int_comp=False*, *subpop=-2*)

Generates completeness values for target stars

This method is called from TargetList __init__ method.

Parameters

- **TL** (*TargetList module*) – TargetList class object
- **calc_char_int_comp** (*boolean*) –
- **subpop** (*int*) – planet subtype to use for calculation of int_comp -2 - planet population -1 - earthLike population 0-N - kopparapu planet subtypes

Returns
Completeness values for each target star

Return type
float ndarray

2.28.1.3 EXOSIMS.Observatory package

Submodules

EXOSIMS.Observatory.ObservatoryL2Halo module

class EXOSIMS.Observatory.ObservatoryL2Halo.**ObservatoryL2Halo**(*equinox=60575.25*, *haloStartTime=0*, *orbit_datapath=None*, ***specs*)

Bases: *Observatory*

Observatory at L2 implementation. The orbit method from the Observatory prototype is overloaded to implement a space telescope on a halo orbit about the Sun-Earth L2 point. This class

Orbit is stored in pickled dictionary on disk (generated by MATLAB code adapted from E. Kolemen (2008). Describes approx. 6 month halo which is then patched for the entire mission duration).

eclip2rot(*TL*, *sInd*, *currentTime*)

Rotates star position vectors from ecliptic to rotating frame in CRTBP

This method returns a star's position vector in the rotating frame of the Circular Restricted Three Body Problem.

Parameters

- **TL** (*TargetList module*) – TargetList class object
- **sInd** (*integer*) – Integer index of the star of interest
- **currentTime** (*astropy Time*) – Current absolute mission time in MJD

Returns
Star position vector in rotating frame in units of AU

Return type
astropy Quantity 1x3 array

equationsOfMotion_CRTBP(*t*, *state*)

Equations of motion of the CRTBP with Solar Radiation Pressure

Equations of motion for the Circular Restricted Three Body Problem (CRTBP). First order form of the equations for integration, returns 3 velocities and 3 accelerations in (x,y,z) rotating frame. All parameters are normalized so that time = 2*pi sidereal year. Distances are normalized to 1AU. Coordinates are taken in a rotating frame centered at the center of mass of the two primary bodies. Pitch angle of the starshade with respect to the Sun is assumed to be 60 degrees, meaning the 1/2 of the starshade cross sectional area is always facing the Sun on average

Parameters

- **t** (*float*) – Times in normalized units
- **state** (*float 6xn array*) – State vector consisting of stacked position and velocity vectors in normalized units

Returns
First derivative of the state vector consisting of stacked velocity and acceleration vectors in normalized units

Return type
float 6xn array

`haloPosition`(*currentTime*)

Finds orbit positions of spacecraft in a halo orbit in rotating frame

This method returns the telescope L2 Halo orbit position vector in an ecliptic, rotating frame as dictated by the Circular Restricted Three Body-Problem. The origin of this frame is the centroid of the Sun and Earth-Moon system.

Parameters
currentTime (*astropy Time array*) – Current absolute mission time in MJD

Returns
Observatory orbit positions vector in an ecliptic, rotating frame in units of AU

Return type
astropy Quantity nx3 array

`haloVelocity`(*currentTime*)

Finds orbit velocity of spacecraft in a halo orbit in rotating frame

This method returns the telescope L2 Halo orbit velocity vector in an ecliptic, rotating frame as dictated by the Circular Restricted Three Body-Problem.

Parameters
currentTime (*astropy Time array*) – Current absolute mission time in MJD

Returns
Observatory orbit velocity vector in an ecliptic, rotating frame in units of AU/year

Return type
astropy Quantity nx3 array

`inert2rotV`(*rR*, *vI*, *t_norm*)

Convert velocity from inertial frame to rotating frame

Parameters

- **rR** (*float nx3 array*) – Rotating frame position vectors
- **vI** (*float nx3 array*) – Inertial frame velocity vectors
- **t_norm** (*float*) – Normalized time units for current epoch

Returns
Rotating frame velocity vectors

Return type
float nx3 array

`integrate`(*s0*, *t*)

Integrates motion in the CRTBP given initial conditions

This method returns a star's position vector in the rotating frame of the Circular Restricted Three Body Problem.

Parameters

- **s0** (*float 1x6 array*) – Initial state vector consisting of stacked position and velocity vectors in normalized units
- **t** (*float*) – Times in normalized units

Returns
State vector consisting of stacked position and velocity vectors in normalized units

Return type
float nx6 array

jacobian_CRTBP(*t*, *s*)

Equations of motion of the CRTBP

Equations of motion for the Circular Restricted Three Body Problem (CRTBP). First order form of the equations for integration, returns 3 velocities and 3 accelerations in (x,y,z) rotating frame. All parameters are normalized so that time = 2*pi sidereal year. Distances are normalized to 1AU. Coordinates are taken in a rotating frame centered at the center of mass of the two primary bodies

Parameters

- **t** (*float*) – Times in normalized units
- **s** (*float nx6 array*) – State vector consisting of stacked position and velocity vectors in normalized units

Returns
Jacobian matrix of the state vector in normalized units

Return type
float nx6x6 array

lookVectors(*TL*, *N1*, *N2*, *tA*, *tB*)

Finds star angular separations relative to the halo orbit positions

This method returns the angular separation relative to the telescope on its halo orbit in the rotating frame of the CRTBP problem.

Parameters

- **TL** (*TargetList module*) – TargetList class object
- **N1** (*integer*) – Integer index of the most recently observed star
- **N2** (*integer*) – Integer index of the next star of interest
- **tA** (*astropy Time*) – Current absolute mission time in MJD
- **tB** (*astropy Time array*) – Time at which next star observation begins in MJD

Returns

float:
Angular separation between two target stars

float 3 array:
Unit vector point from telescope to star 1

float 3 array:
Unit vector point from telescope to star 2

float 3 array:
Position of telescope

Return type
tuple

orbit(*currentTime*, *eclip=False*)

Finds observatory orbit positions vector in heliocentric equatorial (default) or ecliptic frame for current time (MJD).

This method returns the telescope L2 Halo orbit position vector.

Parameters

- **currentTime** (*astropy Time array*) – Current absolute mission time in MJD
- **eclip** (*boolean*) – Boolean used to switch to heliocentric ecliptic frame. Defaults to False, corresponding to heliocentric equatorial frame.

Returns
Observatory orbit positions vector in heliocentric equatorial (default) or ecliptic frame in units of AU

Return type
astropy Quantity nx3 array

Note: Use eclip=True to get ecliptic coordinates.

rot2inertV(*rR*, *vR*, *t_norm*)

Convert velocity from rotating frame to inertial frame

Parameters

- **rR** (*float nx3 array*) – Rotating frame position vectors
- **vR** (*float nx3 array*) – Rotating frame velocity vectors
- **t_norm** (*float*) – Normalized time units for current epoch

Returns
Inertial frame velocity vectors

Return type
float nx3 array

EXOSIMS.Observatory.SotoStarshade module

class EXOSIMS.Observatory.SotoStarshade.**SotoStarshade**(*orbit_datapath=None*, *f_nStars=10*, ***specs*)

Bases: *ObservatoryL2Halo*

StarShade Observatory class This class is implemented at L2 and contains all variables, functions, and integrators to calculate occulter dynamics.

boundary_conditions(*rA*, *rB*)

Creates boundary conditions for solving a boundary value problem

This method returns the boundary conditions for the starshade transfer trajectory between the lines of sight of two different stars. Point A corresponds to the starshade alignment with star A; Point B, with star B.

Parameters

- **rA** (*float 1x3 ndarray*) – Starshade position vector aligned with current star of interest
- **rB** (*float 1x3 ndarray*) – Starshade position vector aligned with next star of interest

Returns
Star position vector in rotating frame in units of AU

Return type
float 1x6 ndarray

`calculate_dV`(*TL*, *old_sInd*, *sInds*, *sd*, *slewTimes*, *tmpCurrentTimeAbs*)

Finds the change in velocity needed to transfer to a new star line of sight

This method sums the total delta-V needed to transfer from one star line of sight to another. It determines the change in velocity to move from one station-keeping orbit to a transfer orbit at the current time, then from the transfer orbit to the next station-keeping orbit at currentTime + dt. Station-keeping orbits are modeled as discrete boundary value problems. This method can handle multiple indeces for the next target stars and calculates the dVs of each trajectory from the same starting star.

Parameters

- **dt** (*float 1x1 ndarray*) – Number of days corresponding to starshade slew time
- **TL** (*float 1x3 ndarray*) – TargetList class object
- **nA** (*integer*) – Integer index of the current star of interest
- **N** (*integer*) – Integer index of the next star(s) of interest
- **tA** (*astropy Time array*) – Current absolute mission time in MJD

Returns
State vectors in rotating frame in normalized units

Return type
float nx6 ndarray

`calculate_slewTimes`(*TL*, *old_sInd*, *sInds*, *sd*, *obsTimes*, *tmpCurrentTimeAbs*)

Finds slew times and separation angles between target stars

This method determines the slew times of an occulter spacecraft needed to transfer from one star's line of sight to all others in a given target list.

Parameters

- **TL** (*TargetList module*) – TargetList class object
- **old_sInd** (*integer*) – Integer index of the most recently observed star
- **sInds** (*integer*) – Integer indeces of the star of interest
- **currentTime** (*astropy Time*) – Current absolute mission time in MJD

Returns

sInds (integer):
Integer indeces of the star of interest

sd (astropy Quantity):
Angular separation between stars in rad

slewTimes (astropy Quantity):
Time to transfer to new star line of sight in units of days

dV (astropy Quantity):
Delta-V used to transfer to new star line of sight in units of m/s

Return type
tuple

generate_dVMap(*TL*, *old_sInd*, *sInds*, *currentTime*)

Creates dV map for an occulter slewing between targets.

This method returns a 2D array of the dV needed for an occulter to slew between all the different stars on the target list. The dV map is calculated relative to a reference star and the stars are ordered by their angular separation from the reference star (X-axis). The Y-axis represents the time of flight ("slew time") for a trajectory between two stars.

Parameters

- **TL** (*TargetList module*) – TargetList class object
- **old_sInd** (*integer*) – Integer index of the last star of interest
- **sInds** (*integer ndarray*) – Integer indices of the stars of interest
- **currentTime** (*astropy Time array*) – Current absolute mission time in MJD

Returns

dVMap (float ndarray):
Map of dV needed to transfer from a reference star to another. Each ordered pair (psi,t) of the dV map corresponds to a trajectory to a star an angular distance psi away with flight time of t. units of (m/s)

angles (float ndarray):
Range of angles (in deg) used in dVMap as the X-axis

dt (float ndarray):
Range of slew times (in days) used in dVMap as the Y-axis

Return type
tuple

impulsiveSlew_dV(*dt*, *TL*, *nA*, *N*, *tA*)

Finds the change in velocity needed to transfer to a new star line of sight

This method sums the total delta-V needed to transfer from one star line of sight to another. It determines the change in velocity to move from one station-keeping orbit to a transfer orbit at the current time, then from the transfer orbit to the next station-keeping orbit at currentTime + dt. Station-keeping orbits are modeled as discrete boundary value problems. This method can handle multiple indeces for the next target stars and calculates the dVs of each trajectory from the same starting star.

Parameters

- **dt** (*float 1x1 ndarray*) – Number of days corresponding to starshade slew time
- **TL** (*float 1x3 ndarray*) – TargetList class object
- **nA** (*integer*) – Integer index of the current star of interest
- **N** (*integer*) – Integer index of the next star(s) of interest
- **tA** (*astropy Time array*) – Current absolute mission time in MJD

Returns
State vectors in rotating frame in normalized units

Return type
float nx6 ndarray

log_occulterResults(*DRM*, *slewTimes*, *sInd*, *sd*, *dV*)

Updates the given DRM to include occulter values and results

Parameters

- **DRM** (*dict*) – Design Reference Mission, contains the results of one complete observation (detection and characterization)
- **slewTimes** (*astropy Quantity*) – Time to transfer to new star line of sight in units of days
- **sInd** (*integer*) – Integer index of the star of interest
- **sd** (*astropy Quantity*) – Angular separation between stars in rad
- **dV** (*astropy Quantity*) – Delta-V used to transfer to new star line of sight in units of m/s

Returns
Design Reference Mission dicitonary, contains the results of one complete observation (detection and characterization)

Return type
dict

minimize_fuelUsage(*TL*, *nA*, *nB*, *tA*)

Minimizes the fuel usage of a starshade transferring to a new star line of sight

This method uses scipy's optimization module to minimize the fuel usage for a starshade transferring between one star's line of sight to another's. The total slew time for the transfer is bounded with some dt_min and dt_max.

Parameters

- **TL** (*float 1x3 ndarray*) – TargetList class object
- **nA** (*integer*) – Integer index of the current star of interest
- **nB** (*integer*) – Integer index of the next star of interest
- **tA** (*astropy Time array*) – Current absolute mission time in MJD

Returns

opt_slewTime (float):
Optimal slew time in days for starshade transfer to a new line of sight

opt_dV (float):
Optimal total change in velocity in m/s for starshade line of sight transfer

Return type
tuple

minimize_slewTimes(*TL*, *nA*, *nB*, *tA*)

Minimizes the slew time for a starshade transferring to a new star line of sight

This method uses scipy's optimization module to minimize the slew time for a starshade transferring between one star's line of sight to another's under the constraint that the total change in velocity cannot exceed more than a certain percentage of the total fuel on board the starshade.

Parameters

- **TL** (*float 1x3 ndarray*) – TargetList class object
- **nA** (*integer*) – Integer index of the current star of interest
- **nB** (*integer*) – Integer index of the next star of interest
- **tA** (*astropy Time array*) – Current absolute mission time in MJD

Returns

opt_slewTime (float):
Optimal slew time in days for starshade transfer to a new line of sight

opt_dV (float):
Optimal total change in velocity in m/s for starshade line of sight transfer

Return type
tuple

send_it(*TL*, *nA*, *nB*, *tA*, *tB*)

Solves boundary value problem between starshade star alignments

This method solves the boundary value problem for starshade star alignments with two given stars at times tA and tB. It uses scipy's solve_bvp method.

Parameters

- **TL** (*float 1x3 ndarray*) – TargetList class object
- **nA** (*integer*) – Integer index of the current star of interest
- **nB** (*integer*) – Integer index of the next star of interest
- **tA** (*astropy Time array*) – Current absolute mission time in MJD
- **tB** (*astropy Time array*) – Absolute mission time for next star alignment in MJD

Returns
State vectors in rotating frame in normalized units

Return type
float nx6 ndarray

EXOSIMS.Observatory.SotoStarshade_ContThrust module

class EXOSIMS.Observatory.SotoStarshade_ContThrust.**SotoStarshade_ContThrust**(*orbit_datapath=None*, ***specs*)

Bases: *SotoStarshade_SKi*

StarShade Observatory class This class is implemented at L2 and contains all variables, functions, and integrators to calculate occulter dynamics.

DCM_i2r(*t*)

Direction cosine matrix to rotate from Inertial Frame to Rotating Frame

Finds rotation matrix for positions and velocities (6x6)

Parameters
t (*float*) – Rotation angle

Returns
rotation of full 6 dimensional state from I to R frame

Return type
float 6x6 array

DCM_r2i(*t*)

Direction cosine matrix to rotate from Rotating Frame to Inertial Frame

Finds rotation matrix for positions and velocities (6x6)

Parameters
t (*float*) – Rotation angle

Returns
rotation of full 6 dimensional state from R to I frame

Return type
float 6x6 array

`DCM_r2i_9`(*t*)

Direction cosine matrix to rotate from Rotating Frame to Inertial Frame

Finds rotation matrix for positions, velocities, and accelerations (9x9)

Parameters
t (*float*) – Rotation angle

Returns
rotation of full 9 dimensional state from R to I frame

Return type
float 9x9 array

`EoM_Adjoint`(*t*, *state*, *constrained=False*, *amax=False*, *integrate=False*)

Equations of Motion with costate vectors

Equations of motion for CR3BP with costates for control.

Parameters

- **t** (*float*) – Currently unused
- **state** (*array*) – The state and lagrange multipliers.
- **constrained** (*boolean*) – Currently unused
- **amax** (*float or boolean*) – Maximum acceleration attainable or False otherwise
- **integrate** (*boolean*) – If true, the array is flattened for integration.

Returns
The time derivatives of the states and costates.

Return type
array

`boundary_conditions_thruster`(*sA*, *sB*, *constrained=False*)

Creates boundary conditions for solving a boundary value problem

Function used in scipy solve_bvp function call. Returns residuals

Parameters

- **sA** (*6 or 7 array*) – To be compared with the initial state for boundary condition.
- **sB** (*6 or 7 array*) – To be compared with the final state for boundary condition.
- **constrained** (*boolean*) – Whether there are 6 (false) or 7 (true) boundary conditions.

Returns
Returns the residuals of the boundary conditions/constraint equation.

Return type
array

`calculate_dMmap`(*TL*, *tA*, *dtRange*, *filename*)

Calculates change in mass for transfers of various times and angular distances

These are stored in a cache .dmmap file.

Parameters

- **TL** (*TargetList module*) – Target list of stars
- **tA** (*astropy Time array*) – Initial absolute mission time in MJD
- **dtRange** (*astropy Time array*) – Transfer times to try
- **filename** (*string*) – File name to store the cached data.

calculate_dMmap_collocate(*TL*, *tA*, *dtRange*, *filename*)

Calculates change in mass for transfers of various times and angular distances

These are stored in a cache .dmmap file. Only collocation without the single shooting.

Parameters

- **TL** (*TargetList module*) – Target list of stars
- **tA** (*astropy Time array*) – Initial absolute mission time in MJD
- **dtRange** (*astropy Time array*) – Transfer times to try
- **filename** (*string*) – File name to store the cached data.

calculate_dMmap_collocateEnergy(*TL*, *tA*, *dtRange*, *filename*, *m0=1*, *seed=0*)

Calculates change in mass for transfers of various times and angular distances

These are stored in a cache .dmmap file. Only minimum energy collocation is used.

Parameters

- **TL** (*TargetList module*) – Target list of stars
- **tA** (*astropy Time array*) – Initial absolute mission time in MJD
- **dtRange** (*astropy Time array*) – Transfer times to try
- **filename** (*string*) – File name to store the cached data.
- **m0** (*float*) – Initial mass
- **seed** (*int*) – Seed random number for repeatability of experiments

calculate_dMmap_collocateEnergy_LatLon(*TL*, *tA*, *dtRange*, *nStars*, *filename*, *m0=1*, *seed=0*)

Calculates change in mass for transfers of various times and angular distances

These are stored in a cache .dmmap file. Only minimum energy collocation is used. All pairs between nStars random stars from the targetlist. All transfer times are used.

Parameters

- **TL** (*TargetList module*) – Target list of stars
- **tA** (*astropy Time array*) – Initial reference absolute mission time in MJD
- **dtRange** (*astropy Time array*) – Transfer times to try
- **nStars** (*int*) – Number of trials/combinations to perform
- **filename** (*string*) – File name to store the cached data.
- **m0** (*float*) – Initial mass
- **seed** (*int*) – Seed random number for repeatability of experiments

calculate_dMmap_collocateEnergy_angSepDist(*TL*, *tA*, *dtRange*, *nPairs*, *filename*, *m0=1*, *seed=0*)

Calculates change in mass for transfers of various times and angular distances

These are stored in a cache .dmmap file. Only minimum energy collocation is used. nPair random combinations of initial and final star are used while all transfer times are used.

Parameters

- **TL** (*TargetList module*) – Target list of stars
- **tA** (*astropy Time array*) – Initial reference absolute mission time in MJD
- **dtRange** (*astropy Time array*) – Transfer times to try
- **nPairs** (*int*) – Number of trials/combinations to perform
- **filename** (*string*) – File name to store the cached data.
- **m0** (*float*) – Initial mass
- **seed** (*int*) – Seed random number for repeatability of experiments

calculate_dMsols_collocateEnergy(*TL*, *tStart*, *tArange*, *dtRange*, *N*, *filename*, *m0=1*, *seed=0*)

Calculates change in mass for transfers of various times and angular distances

These are stored in a cache .dmmap file. Only minimum energy collocation is used. N random combinations of starting times, transfer times, and initial, and final stars are used.

Parameters

- **TL** (*TargetList module*) – Target list of stars
- **tStart** (*astropy Time array*) – Initial reference absolute mission time in MJD
- **tArange** (*astropy Time array*) – Potential times to add to tStart
- **dtRange** (*astropy Time array*) – Transfer times to try
- **N** (*int*) – Number of trials/combinations to perform
- **filename** (*string*) – File name to store the cached data.
- **m0** (*float*) – Initial mass
- **seed** (*int*) – Seed random number for repeatability of experiments

collocate_Trajectory(*TL*, *nA*, *nB*, *tA*, *dt*)

Solves minimum energy and minimum fuel cases for continuous thrust

Parameters

- **TL** (*TargetList module*) – Target list
- **nA** (*int*) – The index for the starting star in the target list
- **nB** (*int*) – The index for the final star in the target list
- **tA** (*astropy Time array*) – Initial absolute mission time in MJD
- **dt** (*astropy Time array*) – A time step between the two voundary conditions

Returns

array:
Trajectory

array:
Times corresponding to trajectory

float:
last epsilon that fully converged (2 if minimum energy didn't work) Parameterizes minimum energy to minimum fuel solution.

astropy Newton array:
Range of thrusts (Newtons) considered.

Return type
tuple

`collocate_Trajectory_minEnergy`(*TL*, *nA*, *nB*, *tA*, *dt*, *m0=1*)

Solves minimum energy and minimum fuel cases for continuous thrust

Parameters

- **TL** (*TargetList module*) – Target list
- **nA** (*int*) – The index for the starting star in the target list
- **nB** (*int*) – The index for the final star in the target list
- **tA** (*astropy Time array*) – Initial absolute mission time in MJD
- **dt** (*astropy Time array*) – A time step between the two voundary conditions
- **m0** (*float*) – Initial mass

Returns

array:
Trajectory

array:
Times corresponding to trajectory

float:
last epsilon that fully converged (2 if minimum energy didn't work) Parameterizes minimum energy to minimum fuel solution.

astropy Newton array:
Range of thrusts (Newtons) considered.

Return type
tuple

`conFun_singleShoot`(*w*, *t0*, *tF*, *Tmax*, *returnLog=False*)

Objective Function for single shooting thruster

Parameters

- **w** (*array*) – Costate vector
- **t0** (*astropy Time object*) – Initial time
- **tf** (*astropy Time object*) – Final time
- **Tmax** (*astropy force*) – Maximum thrust attainable
- **returnLog** (*boolean*) – Return the states and times of the solution

Returns

float:
Norm of difference between current state and boundary value

array:
Trajectory states

astropy Time array:
Times corresponding to states

Return type
tuple

`determineThrottle`(*state*)

Determines throttle based on instantaneous switching function value.

Typically being used during collocation algorithms. A zero-crossing of the switching function is highly unlikely between the large number of nodes.

Parameters
state (*float array*) – State vector and lagrange multipliers

Returns
Throttle between 0 and 1, and in the same shape as state

Return type
float array

`findInitialTmax`(*TL*, *nA*, *nB*, *tA*, *dt*, *m0=1*, *s_init=array([], dtype=float64)*)

Finding initial guess for starting Thrust

Parameters

- **TL** (*TargetList module*) – Target list
- **nA** (*int*) – The index for the starting star in the target list
- **nB** (*int*) – The index for the final star in the target list
- **tA** (*astropy Time array*) – Initial absolute mission time in MJD
- **dt** (*astropy Time array*) – A time step between the two voundary conditions
- **m0** (*float*) – Initial mass
- **s_init** (*array*) – An initial guess for the state

Returns

astropy Newtons:
The maximum initial thrust.

array:
States from the solved bvp.

array:
Times at corrsponding to each state.

Return type
tuple

`findTmaxGrid`(*TL*, *tA*, *dtRange*)

Create grid of Tmax values using unconstrained thruster

This method is used purely for creating figures.

Parameters

- **TL** (*TargetList module*) – Target list

- **tA** (*astropy Time array*) – Initial absolute mission time in MJD
- **dtRange** (*astropy Time array*) – An array of delta times to consider

Returns
Max thrust in Newtons (dtRange dimensions by TL.nStars)

Return type
astropy Newtons array

integrate_thruster(*sGuess*, *tGuess*, *Tmax*, *verbose=False*)

Integrates thruster trajectory with thrust case switches

This methods integrates an initial guess for the spacecraft state forwards in time. It uses event functions to find the next zero of the switching function which means a new thrust case is needed (full, medium or no thrust).

Parameters

- **sGuess** (*array*) – Initial state and costate guess
- **tGuess** (*astropy Time array*) – Times corresponding to the guess of the state at each time
- **Tmax** (*astropy force*) – Maximum thrust attainable

Returns

array:
Trajectory states

astropy Time array:
Times corresponding to states

Return type
tuple

lagrangeMult()

Generate a random lagrange multiplier for initial guess (6x1)

Generates an array of 6 numbers with two pairs of 3 coordinates describing position on a sphere. The first two represent the x and y positions on a sphere with random radius between 1 and 5, and the last coordinate represents the z coordinate scaled by the radius.

Returns
6x1 Random lagrange multipliers for an initial guess

Return type
ndarray(float)

minimize_TerminalState(*s_best*, *t_best*, *Tmax*, *method*)

Minimizes boundary conditions for thruster

Parameters

- **sGuess** (*array*) – Initial state and costate guess
- **tGuess** (*astropy Time array*) – Times corresponding to the guess of the state at each time
- **Tmax** (*astropy force*) – Maximum thrust attainable
- **method** (*string*) – Optimization method for Scipy minimize call

Returns

float:
Norm of difference between current state and boundary value

array:
Trajectory states

astropy Time array:
Times corresponding to states

Return type
tuple

`newStar_angularSep`(*TL*, *iStars*, *jStars*, *currentTime*, *dt*)

Finds angular separation from old star to given list of stars

This method returns the angular separation from the last observed star to all others on the given list at the currentTime. This is distinct from the prototype version of the method in its signing of the angular separation based on halo velocity direction

Parameters

- **TL** (*TargetList module*) – TargetList class object
- **old_sInd** (*integer*) – Integer index of the last star of interest
- **sInds** (*integer ndarray*) – Integer indices of the stars of interest
- **currentTime** (*astropy Time array*) – Current absolute mission time in MJD
- **dt** (*float*) – Timestep in units of days for determining halo velocity frame

Returns
Angular separation between two target stars

Return type
float

`selectEventFunctions`(*s0*)

Selects the proper event function for integration.

This method calculates the switching function and its derivative at a single specific state. It then determines which thrust case it will be in: full, medium, or no thrust. If the value of the switching function is within a certain tolerance of the boundaries, it uses the derivative to determine the direction it is heading in. Then the proper event functions are created for the integrator to determine the next crossing (i.e. the next case change).

Parameters
s0 (*float array*) – Iniitial state vector and lagrange multipliers

Returns

function list:
A list of functions which are the switching function adjusted by 1e-10. These take in an unused first parameter, and then the state. Depending on the case (2), multiple functions are returned. These functions will return 0 when the switching function crosses 1e-10 or -1e-10.

int:
An integer 0,1,2 representing whether the switching function of the initial state is within a 1e-10 tolerance of the of zero (2), less than -1e-10 (1), or greater than 1e-10 (0).

Return type
tuple

selectPairsOfStars(*TL*, *nPairs*, *currentTime*, *dt*, *nSamples*)

Rejection sampling of star pairings using desired distribution and sampling from that final distribution

Parameters

- **TL** (*TargetList module*) – TargetList class object
- **nPairs** (*int*) – Number of pairs to produce
- **currentTime** (*astropy Time array*) – Current absolute mission time in MJD
- **dt** (*float*) – Timestep in units of days for determining halo velocity frame
- **nSamples** (*int*) – Number of samples to be used when generating random stars for pairing

Returns

int list:
The list of indices corresponding to starting stars in TargetList

int list:
The list of indices corresponding to the ending stars in TargetList

float array:
Angular distance between pairs from iFinal and jFinal

Return type
tuple

send_it_thruster(*sGuess*, *tGuess*, *aMax=False*, *constrained=False*, *m0=1*, *maxNodes=100000.0*, *verbose=False*)

Solving generic bvp from t0 to tF using states and costates

Uses the Scipy solve_bvp method.

Parameters

- **sGuess** (*array*) – Initial state and costate guess
- **tGuess** (*astropy Time array*) – Times corresponding to the guess of the state at each time
- **aMax** (*astropy Newton or boolean*) – Maximum attainable acceleration
- **constrained** (*boolean*) – Flag for whether this is constrained or unconstrained problem This determines dimensions of the state inputs.
- **m0** (*float*) – Initial mass
- **maxNodes** (*int*) – Maximum number of nodes to use in the BVP problem solution
- **verbose** (*boolean*) – Flag passed to solve_bvp

Returns

array:
State and costate at the various mesh times

array:
Corresponding times to the sampled states.

boolean:
Status returned by the bvp_solve method

Return type
tuple

singleShoot_Trajectory(*stateLog*, *timeLog*, *e_best*, *TmaxRange*, *method='SLSQP'*)

Perform single shooting to solve the boundary value problem.

Parameters

- **stateLog** (*array*) – Approximate trajectory typically determined by collocation
- **timeLog** (*astropy Time array*) – Corresponging time values for the trajectory
- **e_best** (*float*) – Epsilon value corresponding to previous approxmiate trajectory
- **TmaxRange** (*astropy Newton array*) – Range of thrusts (Newtons) considered.
- **method** (*string*) – Optimization method for Scipy minimize call

Returns

array:
Trajectory states

astropy Time array:
Times corresponding to states

float:
Epsilon value determining how fuel vs energy optimal the trajectory is.

Return type
tuple

star_angularSepDesiredDist(*psiPop*, *nSamples=1000000.0*, *angBinSize=3*)

Rejection sample from psiPop to achieve nSamples fitting logistic distribution

Parameters

- **psiPop** (*array float*) – List of floats between -180 and 180
- **nSamples** (*int*) – Number of samples to return
- **angBinSize** (*float*) – Size of a bin for the histogram used in rejection sampling

Returns

array float:
Angles (nSamples) fitting the logistic distribution

array int:
An array of indices of original psiPop that were accepted

Return type
tuple

starshadeBoundaryVelocity(*TL*, *sInd*, *currentTime*, *SRP=True*)

Calculates starshade and telescope velocities in R- or I-frames during stationkeeping

Calculates the rotating system position and velocity of the starshade at insertion into the optimal initial state for observation of the given star.

Parameters

- **TL** (*TargetList module*) – Target List
- **sInd** (*int*) – Index of star for observation in the target list
- **currentTime** (*astropy Time array*) – Current absolute mission time in MJD
- **SRP** (*boolean*) – Whether or not solar radiation pressure should be used in calculating the insertion state for optimal starshade stationkeeping observation position.

Returns

array:
Position in the rotating frame

array:
Velocity in the rotating frame

Return type
tuple

`switchingFunction`(*state*)

Evaluates the switching function at specific states.

Parameters
state (*float array*) – State vector and lagrange multipliers

Returns
Value of the switching function

Return type
float

`switchingFunctionDer`(*state*)

Evaluates the time derivative of the switching function.

Switching function derivative evaluated for specific states.

Parameters
state (*float array*) – State vector and lagrange multipliers

Returns
Value of the switching function time derivative

Return type
float

EXOSIMS.Observatory.SotoStarshade_SKi module

`class EXOSIMS.Observatory.SotoStarshade_SKi.SotoStarshade_SKi`(*latDist=0.9*, *latDistOuter=0.95*, *latDistFull=1*, *axlDist=250*, ***specs*)

Bases: *SotoStarshade*

StarShade Observatory class

This class is implemented at L2 and contains all variables, functions, and integrators to calculate occulter dynamics.

`Bframe`(*TL*, *sInd*, *currentTime*, *tRange=array([0])*)

Calculates unit vectors defining B-frame of telescope

The B-frame is placed at the inertial location of the telescope on its halo orbit. The third axis points directly towards the target star sInd. The second axis, by our definition, points parallel to the ecliptic plane of the Sun-Earth.

Parameters

- **TL** (*TargetList module*) – TargetList class object
- **sInd** (*integer ndarray*) – Integer index of some target star

- **currentTime** (*astropy Time array*) – Current absolute mission time in MJD
- **tRange** (*float ndarray*) – Array of times relative to currentTime to calculate values. The array has size m

Returns

b1_I (float 3xm numpy.ndarray):
First axis B-frame unit vector for each time in dimension m

b2_I (float 3xm numpy.ndarray):
Second axis B-frame unit vector for each time in dimension m

b3_I (float 3xm numpy.ndarray):
Third axis B-frame unit vector for each time in dimension m. This one points towards the star sInd

Return type
tuple

EulerAngleAndDerivatives(*TL*, *sInd*, *currentTime*, *tRange=array([0])*)

Calculates Euler angles and rates for LOS from telescope to star sInd

This method calculates Euler angles defining the line of sight (LOS) from the telescope to some star sInd in the target list TL. The Euler angles are defined relative to some B-frame placed at the inertial location of the telescope on its halo orbit. Derivatives of the Euler angles, representing slewing rates of the LOS, are also calculated.

Parameters

- **TL** (*TargetList module*) – TargetList class object
- **sInd** (*integer ndarray*) – Integer index of some target star
- **currentTime** (*astropy Time array*) – Current absolute mission time in MJD
- **tRange** (*float ndarray*) – Array of times relative to currentTime to calculate values. The array has size m

Returns

theta (float m numpy.ndarray):
Azimuthal angle to define star LOS in rad

phi (float m numpy.ndarray):
Polar angle to define star LOS in rad

dtheta (float m numpy.ndarray):
Azimuthal angle to define star LOS in canonical units

dphi (float m numpy.ndarray):
Polar angle to define star LOS in canonical units

Return type
tuple

SRPforce(*TL*, *sInd*, *currentTime*, *tRange*, *radius=36*)

Solar radiation pressure force for starshade

This method calculate the solar radiation pressure force on a starshade on a nominal trajectory aligned with some star sInd from the target list TL.

Parameters

- **TL** (*TargetList module*) – TargetList class object

- **sInd** (*integer ndarray*) – Integer index of some target star
- **currentTime** (*astropy Time array*) – Current absolute mission time in MJD
- **tRange** (*float ndarray*) – Array of times relative to currentTime to calculate values. The array has size m
- **radius** (*float array*) – Radius of the starshade in meter

Returns
Solar radiation pressure force in canonical units

Return type
f_SRP (float 6xn array)

convertAcc_to_canonical(*dimAcc*)

Convert array of accelerations from dimensional units to canonical units

Method converts the accelerationss inside the array from the given dimensional unit (doesn't matter which, it converts to units of au/yr^2 in an intermediate step) into canonical units of the CR3BP.

Parameters
dimAcc (*float numpy.ndarray*) – Array of accelerations in some acceleration unit

Returns
Array of accelerations in canonical units

Return type
canonicalAcc (float numpy.ndarray)

convertAcc_to_dim(*canonicalAcc*)

Convert array of accelerations from canonical units to dimensional units

Method converts the accelerations inside the array from canonical units of the CR3BP into units of au/yr^2.

Parameters
canonicalAcc (*float numpy.ndarray*) – Array of accelerations in canonical units

Returns
Array of accelerations in units of AU/yr^2

Return type
dimAcc (float numpy.ndarray)

convertAngAcc_to_canonical(*dimAngAcc*)

Convert array of angular accelerations from dimensional units to canonical units

Method converts the angular accelerationss inside the array from the given dimensional unit (doesn't matter which, it converts to units of rad/yr^2 in an intermediate step) into canonical units of the CR3BP.

Parameters
dimAngAcc (*float numpy.ndarray*) – Array of angular accelerations in some angular acceleration unit

Returns
Array of angular accelerations in canonical units

Return type
canonicalAngAcc (float numpy.ndarray)

convertAngAcc_to_dim(*canonicalAngAcc*)

Convert array of angular accelerations from canonical units to dimensional units

Method converts the angular accelerations inside the array from canonical units of the CR3BP into units of rad/yr^2.

Parameters
canonicalAngAcc (*float numpy.ndarray*) – Array of accelerations in canonical units

Returns
Array of accelerations in units of rad/yr^2

Return type
dimAngAcc (float numpy.ndarray)

convertAngVel_to_canonical(*dimAngVel*)

Convert array of angular velocities from dimensional units to canonical units

Method converts the angular velocities inside the array from the given dimensional unit (doesn't matter which, it converts to units of rad/yr in an intermediate step) into canonical units of the CR3BP.

Parameters
dimAngVel (*float numpy.ndarray*) – Array of angular velocities in some angular velocity unit

Returns
Array of angular velocities in canonical units

Return type
canonicalAngVel (float numpy.ndarray)

convertAngVel_to_dim(*canonicalAngVel*)

Convert array of angular velocities from canonical units to dimensional units

Method converts the angular velocities inside the array from canonical units of the CR3BP into units of rad/yr.

Parameters
canonicalAngVel (*float numpy.ndarray*) – Array of angular velocities in canonical units

Returns
Array of angular velocities in units of rad/yr

Return type
dimAngVel (float numpy.ndarray)

convertPos_to_canonical(*dimPos*)

Convert array of positions from dimensional units to canonical units

Method converts the positions inside the array from the given dimensional unit (doesn't matter which, it converts to units of AU in an intermediate step) into canonical units of the CR3BP. 1 au = 1 DU where DU are the canonical position units.

Parameters
dimPos (*float numpy.ndarray*) – Array of positions in some distance unit

Returns
Array of distance in canonical units

Return type
canonicalPos (float numpy.ndarray)

convertPos_to_dim(*canonicalPos*)

Convert array of positions from canonical units to dimensional units

Method converts the positions inside the array from canonical units of the CR3BP into units of AU.

Parameters
canonicalPos (*float numpy.ndarray*) – Array of distance in canonical units

Returns
Array of positions in units of AU

Return type
dimPos (float numpy.ndarray)

convertTime_to_canonical(*dimTime*)

Convert array of times from dimensional units to canonical units

Method converts the times inside the array from the given dimensional unit (doesn't matter which, it converts to units of years in an intermediate step) into canonical units of the CR3BP. 1 yr = 2 pi TU where TU are the canonical time units.

Parameters
dimTime (*float numpy.ndarray*) – Array of times in some time unit

Returns
Array of times in canonical units

Return type
canonicalTime (float numpy.ndarray)

convertTime_to_dim(*canonicalTime*)

Convert array of times from canonical units to unit of years

Method converts the times inside the array from canonical units of the CR3BP into year units. 1 yr = 2 pi TU where TU are the canonical time units.

Parameters
canonicalTime (*float numpy.ndarray*) – Array of times in canonical units

Returns
Array of times in units of years

Return type
dimTime (float numpy.ndarray)

convertVel_to_canonical(*dimVel*)

Convert array of velocities from dimensional units to canonical units

Method converts the velocities inside the array from the given dimensional unit (doesn't matter which, it converts to units of AU/yr in an intermediate step) into canonical units of the CR3BP.

Parameters
dimVel (*float numpy.ndarray)*) – Array of velocities in some speed unit

Returns
Array of velocities in canonical units

Return type
canonicalVel (float numpy.ndarray))

convertVel_to_dim(*canonicalVel*)

Convert array of velocities from canonical units to dimensional units

Method converts the velocities inside the array from canonical units of the CR3BP into units of AU/yr.

Parameters
canonicalVel (*float numpy.ndarray*) – Array of velocities in canonical units

Returns
Array of velocities in units of AU/yr

Return type
dimVel (float numpy.ndarray)

crossThreshholdEvent(*t*, *s*, *TL*, *sInd*, *trajStartTime*, *latDist*, *SRP=False*, *Moon=False*)

Event function for when starshade crosses deadbanding limit

This method is used as an event function in solve_ivp and returns the current distance of the starshade centroid from the lateral deadbanding limit for observations. Takes an input latDist which can be changed if the user selects inner and outer thresholds.

Parameters

- **t** (*float*) – Times in normalized units
- **s** (*float 6xn array*) – State vector consisting of stacked position and velocity vectors in normalized units
- **TL** (*TargetList module*) – TargetList class object
- **sInd** (*integer ndarray*) – Integer index of some target star
- **trajStartTime** (*astropy Time array*) – Current absolute mission time in MJD
- **latDist** (*float Quantity*) – The lateral deadbanding boundary for observations in meters
- **SRP** (*bool*) – Toggles whether or not to include solar radiation pressure force
- **Moon** (*bool*) – Toggles whether or not to include lunar gravity force

Returns
Distance from the deadbanding radius

Return type
distanceFromLim (float)

drift(*TL*, *sInd*, *trajStartTime*, *dt=<Quantity 20. min>*, *freshStart=True*, *s0=None*, *fullSol=False*, *SRP=False*, *Moon=False*)

Method to simulate drift between deadbanding burns for a starshade

This method simulates drifting between deadbanding burns during a starshade observation. Creates event functions for lateral deadbanding threshold crossings, both with inner and outer thresholds. Integrates relative equations of motion until event is triggered and resolves that event to see where the crossing happened.

Parameters

- **TL** (*TargetList module*) – TargetList class object
- **sInd** (*integer ndarray*) – Integer index of some target star
- **trajStartTime** (*astropy Time array*) – Current absolute mission time in MJD
- **dt** (*float Quantity*) – The initial guess for lateral drift time in minutes

- **freshStart** (*bool*) – Toggles whether starshade starts at the optimal initial point if True or at some given s0 if False
- **s0** (*float 6 ndarray*) – The initial state of the drift, set to None as default. Given in canonical units and in I-frame components
- **fullSol** (*bool*) – Optional flag, default False, set True to return additional information Returns full state solutions if True or just the crossing states
- **SRP** (*bool*) – Toggles whether or not to include solar radiation pressure force
- **Moon** (*bool*) – Toggles whether or not to include lunar gravity force

Returns

cross (float):
Flag where 0,1,or 2 mean Tolerance Not Crossed, Lateral Cross, or Axial Cross

driftTime (float astropy.units.Quantity):
Amount of time between threshold crossings in minutes

t_cross (float numpy.ndarray):
Time of lateral limit crossing in canonical units. If fullSol is True, this is t_full - Full time history of drift in canonical units

r_cross (float numpy.ndarray):
Position of lateral limit crossing in canonical units in C-frame components. If fullSol is True, this is r_full - Full position history of drift in canonical units in C-frame components

v_cross (float numpy.ndarray):
Velocity of lateral limit crossing in canonical units in C-frame components but inertial derivatives If fullSol is True, this is v_full - Full velocity history of drift in canonical units in C-frame components but inertial derivatives

Return type
tuple

equationsOfMotion_CRTBPInertial(*t*, *state*, *TL*, *sInd*, *integrate=False*, *SRP=False*, *Moon=False*)

Equations of motion in inertial frame with CRTBP framework

Equations of motion for an object under Sun and Earth's gravity. Forces and accelerations are framed relative to an inertial I-frame with origin at the Sun-Earth barycenter. Assumptions of the Circular Restricted Three Body Problem (CRTBP) are applied here, namely that the Earth and Sun orbit their common center of mass in circular orbits. All components and vectors are given in canonical units of the CRTBP. Two boolean inputs specify whether to add solar radiation pressure or lunar gravity as perturbation forces.

Parameters

- **t** (*float*) – Times in normalized units
- **state** (*float 6xn array*) – State vector consisting of stacked position and velocity vectors in normalized units
- **TL** (*TargetList module*) – TargetList class object
- **sInd** (*integer ndarray*) – Integer index of some target star
- **integrate** (*bool*) – If true, output array is flattened to ensure it is proper input in solve_ivp. Typically have it set to False if using solve_bvp
- **SRP** (*bool*) – Toggles whether or not to include solar radiation pressure force
- **Moon** (*bool*) – Toggles whether or not to include lunar gravity force

Returns
First derivative of the state vector consisting of stacked velocity and acceleration vectors in normalized units

Return type
f_P0_I (float 6xn array)

equationsOfMotion_aboutS(*t*, *state*, *TL*, *sInd*, *trajStartTime*, *integrate=False*, *SRP=False*, *Moon=False*)

Equations of motion of starshade relative to nominal trajectory

Equations of motion for a starshade relative to the nominal trajectory, which is defined as following the LOS perfectly to a star sInd from target list TL. Motion is defined relative to the nominal point S; the offset motion is labeled as O and origin of the solar system barycenter is 0. All components are given in inertial frame components, all vector derivatives are inertial frame derivatives. Units are canonical units.

Parameters

- **t** (*float*) – Times in normalized units
- **state** (*float 6xn array*) – State vector consisting of stacked position and velocity vectors in normalized units
- **TL** (*TargetList module*) – TargetList class object
- **sInd** (*integer ndarray*) – Integer index of some target star
- **trajStartTime** (*astropy Time array*) – Current absolute mission time in MJD
- **integrate** (*bool*) – If true, output array is flattened to ensure it is proper input in solve_ivp. Typically have it set to False if using solve_bvp
- **SRP** (*bool*) – Toggles whether or not to include solar radiation pressure force
- **Moon** (*bool*) – Toggles whether or not to include lunar gravity force

Returns
First derivative of the state vector consisting of stacked relative velocity and acceleration vectors in normalized units

Return type
ds (float 6xn array)

globalStationkeep(*TL*, *trajStartTime*, *tau=<Quantity 0. d>*, *dt=<Quantity 30. min>*, *simTime=<Quantity 1. h>*, *SRP=False*, *Moon=False*, *axlBurn=True*)

Method to simulate global stationkeeping with all target list stars

This method simulates full observations in a loop for all stars in a target list. It logs the same metrics as the stationkeep method and saves it onto a file specified in the body of the method. This method returns nothing.

Parameters

- **TL** (*TargetList module*) – TargetList class object
- **trajStartTime** (*astropy Time array*) – Current absolute mission time in MJD
- **tau** (*float Quantity*) – Time relative to trajStartTime at which to simulate observations in units of days
- **dt** (*float Quantity*) – First guess of trajectory drift time in units of minutes
- **simTime** (*float Quantity*) – Total simulated observation time in units of hours
- **SRP** (*bool*) – Toggles whether or not to include solar radiation pressure force

- **Moon** (*bool*) – Toggles whether or not to include lunar gravity force

Returns:

guessAParabola(*TL*, *sInd*, *trajStartTime*, *r_OS_C*, *Iv_OS_C*, *latDist=<Quantity 0.9 m>*, *fullSol=False*, *SRP=False*, *Moon=False*, *axlBurn=True*)

Method to simulate ideal starshade drift with parabolic motion

This method assumes an ideal, unperturbed trajectory in between lateral deadbanding burns. It assumes that the differential lateral force is constant throughout the entire trajectory and therefore motion is parabolic. Everything is calculated in C-frame components but I-frame derivatives.

Parameters

- **TL** (*TargetList module*) – TargetList class object
- **sInd** (*integer ndarray*) – Integer index of some target star
- **trajStartTime** (*astropy Time array*) – Current absolute mission time in MJD
- **r_OS_C** (*float Quantity*) – The initial guess for lateral drift time in minutes
- **Iv_OS_C** (*bool*) – Toggles whether starshade starts at the optimal initial point if True or at some given s0 if False
- **latDist** (*float 6 ndarray*) – The initial state of the drift, set to None as default. Given in canonical units and in I-frame components
- **fullSol** (*bool*) – Optional flag, default False, set True to return additional information:
- **SRP** (*bool*) – Toggles whether or not to include solar radiation pressure force
- **Moon** (*bool*) – Toggles whether or not to include lunar gravity force

Returns

dt_newTOF (float):
New time of flight for parabolic trajectory in canonical units of CRTBP

Iv_PS_C_newIC (float 3 numpy.ndarray):
New velocity of parabolic trajectory (P) relative to nominal starshade (S) starting at previous lateral burn in canonical units of CRTBP

dv_dim (float numpy.ndarray):
Delta-v of initial lateral burn in dimensional units of m/s

r_PS_C (float ndarray):
Full time history of drift in canonical units (just x-y plane of the C-frame in canonical units of CRTBP). Only returned if fullSol is True.

Return type
tuple

lunarPerturbation(*TL*, *sInd*, *currentTime*, *tRange*, *nodalRegression=True*)

Lunar gravity force for starshade

This method calculate the lunar gravity force on a starshade on a nominal trajectory aligned with some star sInd from the target list TL. Assumes a perfectly circular lunar orbit about the Earth which is inclined at 5.15 degrees from the ecliptic plane and has a period of 29.53 days. We also include precession of the lunar nodes when calculating the lunar position.

Parameters

- **TL** (*TargetList module*) – TargetList class object

- **sInd** (*integer ndarray*) – Integer index of some target star
- **currentTime** (*astropy Time array*) – Current absolute mission time in MJD
- **tRange** (*float ndarray*) – Array of times relative to currentTime to calculate values. The array has size m

Returns
Lunar gravity force in canonical units

Return type
f_Moon (float 6xn array)

rotateComponents2NewFrame(*TL*, *sInd*, *trajStartTime*, *s_int*, *t_int*, *final_frame='C'*, *SRP=False*, *Moon=False*)

Rotates state vector at different times into an ideal dynamics frame

We introduce a new frame (the C-frame) rotated from the B-frame by an angle psi. Psi is found through the self.starshadeIdealDynamics. The C-frame is defined so that the lateral component of the differential force on the starshade always points down (in the -c2 direction). This method rotates a state vector s_int at every given respective time t_int.

Parameters

- **TL** (*TargetList module*) – TargetList class object
- **sInd** (*integer ndarray*) – Integer index of some target star
- **trajStartTime** (*astropy Time array*) – Current absolute mission time in MJD
- **s_int** (*float 6xn array*) – Array of n state vectors with inertial velocities. Components are given in canonical units and are in either I-frame or C-frame.
- **t_int** (*float numpy.ndarray*) – Array of times for each of the n state vectors in s_int given in canonical units
- **final_frame** (*string*) – String entry that rotates states to the C-frame if input is 'C' or I-frame otherwise
- **SRP** (*bool*) – Toggles whether or not to include solar radiation pressure force
- **Moon** (*bool*) – Toggles whether or not to include lunar gravity force

Returns

Array of n position vectors rotated to frame specified by
final_frame

Iv_f (float 3xn array):
Array of n velocity vectors rotated to frame specified by final_frame

Return type
r_f (float 3xn array)

starshadeIdealDynamics(*TL*, *sInd*, *currentTime*, *tRange=array([0])*, *SRP=False*, *Moon=False*)

Calculates ideal dynamics of nominal starshade positioning at LOS

This method calculates things to define ideal dynamics of a starshade under an nominal trajectory. The starshade is assumed to be on the nominal trajectory (on the LOS at some separation distance) and experiences gravity from the Sun and Earth. SRP and Moon forces can be included but are optional inputs. Method returns differential forces, the difference between the forces on the starshade and the acceleration it must have to remain on the nominal path. This difference pushes the starshade away from the nominal trajectory onto some offset trajectory.

Parameters

- **TL** (*TargetList module*) – TargetList class object
- **sInd** (*integer ndarray*) – Integer index of some target star
- **currentTime** (*astropy Time array*) – Current absolute mission time in MJD
- **tRange** (*float ndarray*) – Array of times relative to currentTime to calculate values. The array has size m
- **SRP** (*bool*) – Toggles whether or not to include solar radiation pressure force
- **Moon** (*bool*) – Toggles whether or not to include lunar gravity force

Returns

The third Euler angle that completes the set. Roll angle that
rotates B frame to some new frame where lateral component of dF points in the negative 2nd axis direction

dfL_I (float 3xm array):
Lateral component of the differential force on starshade in canonical units (lateral to LOS)

dfA (float m array):
Axial component of the differential force on starshade in canonical units (along LOS)

df_S0_I (float 3xm array):
Full differential force on starshade in canonical units (net force - nominal accelerations of S)

f_S0_I (float 3xm array):
Full net force on starshade in canonical units

Return type
psi (float m array)

starshadeInjectionVelocity(*TL*, *sInd*, *trajStartTime*, *SRP=False*, *Moon=False*)

Method to find injection velocity of starshade to start observation

This method returns the ideal injection velocity and position of a starshade to begin an observation with a star sInd from target list TL. Position and velocity are given in C-frame components but I-frame derivatives.

Parameters

- **TL** (*TargetList module*) – TargetList class object
- **sInd** (*integer ndarray*) – Integer index of some target star
- **trajStartTime** (*astropy Time array*) – Current absolute mission time in MJD
- **SRP** (*bool*) – Toggles whether or not to include solar radiation pressure force
- **Moon** (*bool*) – Toggles whether or not to include lunar gravity force

Returns

New position of offset trajectory (O) relative to nominal
starshade (S) starting at previous lateral burn in canonical units of CRTBP

Iv_OS_C_newIC (float 3 ndarray):
New velocity of offset trajectory (O) relative to nominal starshade (S) starting at previous lateral burn in canonical units of CRTBP and inertial derivatives

Return type
r_OS_C (float)

starshadeKinematics(*TL*, *sInd*, *currentTime*, *tRange=array([0])*)

Calculates full kinematics of nominal starshade positioning at LOS

This method calculates the full kinematics (positions, velocities, and accelerations) of the nominal starshade trajectory during an observation. The nominal trajectory is one that follows the changing LOS from telescope to star at a constant separation distance. Kinematics are given in inertial frame components and derivates are taken as inertial vector derivatives. Also returns the inertial kinematics relative to the telescope.

Parameters

- **TL** (*TargetList module*) – TargetList class object
- **sInd** (*integer ndarray*) – Integer index of some target star
- **currentTime** (*astropy Time array*) – Current absolute mission time in MJD
- **tRange** (*float ndarray*) – Array of times relative to currentTime to calculate values. The array has size m

Returns

Nominal position (r) of starshade (S) relative to inertial frame
origin (0) given in inertial frame components (_I)

Iv_S0_I (float 3xm array):
Nominal inertial velocity (Iv) of starshade (S) relative to inertial frame origin (0) given in inertial frame components (_I)

Ia_S0_I (float 3xm array):
Nominal inertial acceleration (Ia) of starshade (S) relative to inertial frame origin (0) given in inertial frame components (_I)

r_ST_I (float 3xm array):
Nominal position (r) of starshade (S) relative to telescope location (T) given in inertial frame components (_I)

Iv_ST_I (float 3xm array):
Nominal inertial velocity (Iv) of starshade (S) relative to telescope location (T) given in inertial frame components (_I)

Ia_ST_I (float 3xm array):
Nominal inertial acceleration (Ia) of starshade (S) relative to telescope location (T) given in inertial frame components (_I)

Return type
r_S0_I (float 3xm array)

stationkeep(*TL*, *sInd*, *trajStartTime*, *dt=<Quantity 30. min>*, *simTime=<Quantity 1. h>*, *SRP=False*, *Moon=False*, *axlBurn=True*)

Method to simulate full stationkeeping with a given star

This method simulates a full observation for a star sInd in target list TL. It calculates drifts in a sequence until the allotted simulation time simTime is over. It then logs various metrics including delta-v, drift times and number of thruster firings to be catalogued by the user.

Parameters

- **TL** (*TargetList module*) – TargetList class object
- **sInd** (*integer ndarray*) – Integer index of some target star
- **trajStartTime** (*astropy Time array*) – Current absolute mission time in MJD

- **dt** (*float Quantity*) – First guess of trajectory drift time in units of minutes
- **simTime** (*float Quantity*) – Total simulated observation time in units of hours
- **SRP** (*bool*) – Toggles whether or not to include solar radiation pressure force
- **Moon** (*bool*) – Toggles whether or not to include lunar gravity force

Returns

nBounces (float):
Number of thruster firings throughout observation

timeLeft (float Quantity):
Amount of time left when simulation ended in units of hours

dvLog (float n Quantity):
Log of delta-v's with size n where n is equal to nBounces and units of m/s

dvAxialLog (float n Quantity):
Log of delta-v's purely in the axial direction with size n where n is equal to nBounces and units of m/s

driftLog (float n Quantity):
Log of all drift times with size n where n is equal to nBounces and units of minutes

Return type
tuple

unitVector(*p*)

Normalizes an array and returns associated unit vector

Takes in some array p that represents a vector with dimensions 3xn. It then calculates the norm of that vector and also normalizes it to create a unit vector.

Parameters
p (*float 3xn numpy.ndarray*) – Array of values

Returns

p (float 3xn numpy.ndarray):
Unit vector associated with p, same dimensions

pnorm (float numpy.ndarray):
Norm of the given vector for each value n

Return type
tuple

EXOSIMS.Observatory.SotoStarshade_parallel module

class EXOSIMS.Observatory.SotoStarshade_parallel.**SotoStarshade_parallel**(*orbit_datapath=None*, ***specs*)

Bases: *SotoStarshade_ContThrust*

StarShade Observatory class This class is implemented at L2 and contains all variables, functions, and integrators to calculate occulter dynamics.

run_ensemble(*fun*, *nStars*, *tA*, *dtRange*, *m0*, *seed*)

EXOSIMS.Observatory.WFIRSTObservatoryL2 module

class EXOSIMS.Observatory.WFIRSTObservatoryL2.**WFIRSTObservatoryL2**(***specs*)

Bases: *ObservatoryL2Halo*

WFIRST Observatory at L2 implementation. Contains methods and parameters unique to the WFIRST mission.

2.28.1.4 EXOSIMS.OpticalSystem package

Submodules

EXOSIMS.OpticalSystem.KasdinBraems module

class EXOSIMS.OpticalSystem.KasdinBraems.**KasdinBraems**(*PSF=array([[1., 1., 1.], [1., 1., 1.], [1., 1., 1.]])*, ***specs*)

Bases: *OpticalSystem*

KasdinBraems Optical System class

This class contains all variables and methods necessary to perform Optical System Module calculations in exoplanet mission simulation using the model from Kasdin & Braems 2006.

Parameters

- **PSF** (*ndarray(float)*) – Default point spread function suppression system. Only used when not set in starlight suppression system definition. Defaults to np.ones((3,3))
- ****specs** – user specified values

calc_intTime(*TL*, *sInds*, *fZ*, *fEZ*, *dMag*, *WA*, *mode*, *TK=None*)

Finds integration times of target systems for a specific observing mode (imaging or characterization), based on Kasdin and Braems 2006.

Parameters

- **TL** (*TargetList module*) – TargetList class object
- **sInds** (*integer ndarray*) – Integer indices of the stars of interest
- **fZ** (*astropy Quantity array*) – Surface brightness of local zodiacal light in units of 1/arcsec2
- **fEZ** (*astropy Quantity array*) – Surface brightness of exo-zodiacal light in units of 1/arcsec2
- **dMag** (*float ndarray*) – Differences in magnitude between planets and their host star
- **WA** (*astropy Quantity array*) – Working angles of the planets of interest in units of arcsec
- **mode** (*dict*) – Selected observing mode

Returns
Integration times in units of day

Return type
intTime (astropy Quantity array)

populate_starlightSuppressionSystems_extra()

Additional setup for starlight suppression systems. This is intended for overloading in downstream implementations and is intentionally left blank in the prototype.

EXOSIMS.OpticalSystem.Nemati module

class EXOSIMS.OpticalSystem.Nemati.**Nemati**(*CIC=0.001*, *radDos=0*, *PCeff=0.8*, *ENF=1*, *ref_dMag=3*, *ref_Time=0*, ***specs*)

Bases: *OpticalSystem*

Nemati Optical System class

Optical System Module based on [Nemati2014].

Parameters

- **CIC** (*float*) – Default clock-induced-charge (in electrons/pixel/read). Only used when not set in science instrument definition. Defaults to 1e-3
- **radDos** (*float*) – Default radiation dose. Only used when not set in mode definition. Specific defintion depends on particular optical system. Defaults to 0.
- **PCeff** (*float*) – Default photon counting efficiency. Only used when not set in science instrument definition. Defaults to 0.8
- **ENF** (*float*) – Default excess noise factor. Only used when not set in science instrument definition. Defaults to 1.
- **ref_dMag** (*float*) – Reference star $\Delta\mathrm{mag}$ for reference differential imaging. Defaults to 3. Unused if `ref_Time` input is 0
- **ref_Time** (*float*) – Faction of time used on reference star imaging. Must be between 0 and 1. Defaults to 0
- ****specs** – *Input Specification*

default_vals_extra

Dictionary of input values to be filled in as defaults in the instrument, starlight supporession system and observing modes. These values are specific to this module.

Type
dict

ref_dMag

Reference star $\Delta\mathrm{mag}$ for reference differential imaging. Unused if `ref_Time` input is 0

Type
float

ref_Time

Faction of time used on reference star imaging.

Type
float

Cp_Cb_Csp(*TL*, *sInds*, *fZ*, *fEZ*, *dMag*, *WA*, *mode*, *returnExtra=False*, *TK=None*)

Calculates electron count rates for planet signal, background noise, and speckle residuals.

Parameters

- **TL** (*TargetList*) – TargetList class object

- **sInds** (*ndarray(int)*) – Integer indices of the stars of interest
- **fZ** (*Quantity(ndarray(float))*) – Surface brightness of local zodiacal light in units of 1/arcsec2
- **fEZ** (*Quantity(ndarray(float))*) – Surface brightness of exo-zodiacal light in units of 1/arcsec2
- **dMag** (*ndarray(float)*) – Differences in magnitude between planets and their host star
- **WA** (*Quantity(ndarray(float))*) – Working angles of the planets of interest in units of arcsec
- **mode** (*dict*) – Selected observing mode
- **returnExtra** (*bool*) – Optional flag, default False, set True to return additional rates for validation
- **TK** (*TimeKeeping*, optional) – Optional TimeKeeping object (default None), used to model detector degradation effects where applicable.

Returns

C_p (~astropy.units.Quantity(~numpy.ndarray(float))):
Planet signal electron count rate in units of 1/s

C_b (~astropy.units.Quantity(~numpy.ndarray(float))):
Background noise electron count rate in units of 1/s

C_sp (~astropy.units.Quantity(~numpy.ndarray(float))):
Residual speckle spatial structure (systematic error) in units of 1/s

Return type
tuple

calc_dMag_per_intTime(*intTimes*, *TL*, *sInds*, *fZ*, *fEZ*, *WA*, *mode*, *C_b=None*, *C_sp=None*, *TK=None*)

Finds achievable dMag for one integration time per star in the input list at one working angle.

Parameters

- **intTimes** (*astropy Quantity array*) – Integration times
- **TL** (*TargetList module*) – TargetList class object
- **sInds** (*integer ndarray*) – Integer indices of the stars of interest
- **fZ** (*astropy Quantity array*) – Surface brightness of local zodiacal light for each star in sInds in units of 1/arcsec2
- **fEZ** (*astropy Quantity array*) – Surface brightness of exo-zodiacal light for each star in sInds in units of 1/arcsec2
- **WA** (*astropy Quantity array*) – Working angle for each star in sInds in units of arcsec
- **mode** (*dict*) – Selected observing mode
- **C_b** (*astropy Quantity array*) – Background noise electron count rate in units of 1/s (optional)
- **C_sp** (*astropy Quantity array*) – Residual speckle spatial structure (systematic error) in units of 1/s (optional)
- **TK** (*TimeKeeping object*) – Optional TimeKeeping object (default None), used to model detector degradation effects where applicable.

Returns
Achievable dMag for given integration time and working angle

Return type
dMag (ndarray)

`calc_intTime`(*TL*, *sInds*, *fZ*, *fEZ*, *dMag*, *WA*, *mode*, *TK=None*)

Finds integration times of target systems for a specific observing mode (imaging or characterization), based on Nemati 2014 (SPIE).

Parameters

- **TL** (*TargetList module*) – TargetList class object
- **sInds** (*integer ndarray*) – Integer indices of the stars of interest
- **fZ** (*astropy Quantity array*) – Surface brightness of local zodiacal light in units of 1/arcsec2
- **fEZ** (*astropy Quantity array*) – Surface brightness of exo-zodiacal light in units of 1/arcsec2
- **dMag** (*float ndarray*) – Differences in magnitude between planets and their host star
- **WA** (*astropy Quantity array*) – Working angles of the planets of interest in units of arcsec
- **mode** (*dict*) – Selected observing mode
- **TK** (*TimeKeeping object*) – Optional TimeKeeping object (default None), used to model detector degradation effects where applicable.

Returns
Integration times in units of day

Return type
intTime (astropy Quantity array)

`calc_saturation_dMag`(*TL*, *sInds*, *fZ*, *fEZ*, *WA*, *mode*, *TK=None*)

This calculates the delta magnitude for each target star that corresponds to an infinite integration time.

Parameters

- **TL** (*TargetList*) – TargetList class object
- **sInds** (*numpy.ndarray(int)*) – Integer indices of the stars of interest
- **fZ** (*Quantity(ndarray(float))*) – Surface brightness of local zodiacal light in units of 1/arcsec2
- **fEZ** (*Quantity(ndarray(float))*) – Surface brightness of exo-zodiacal light in units of 1/arcsec2
- **WA** (*Quantity(ndarray(float))*) – Working angles of the planets of interest in units of arcsec
- **mode** (*dict*) – Selected observing mode
- **TK** (*TimeKeeping*, optional) – Optional TimeKeeping object (default None), used to model detector degradation effects where applicable.

Returns
Maximum achievable dMag for each target star

Return type
ndarray(float)

dMag_per_intTime_obj(*dMag*, **args*)

Objective function for calc_dMag_per_intTime's minimize_scalar function that uses calc_intTime from Nemati and then compares the value to the true intTime value

Parameters

- **dMag** (*ndarray(float)*) – dMag being tested
- ***args** – all the other arguments that calc_intTime needs

Returns
Absolute difference between true and evaluated integration time in days.

Return type
ndarray(float)

ddMag_dt(*intTimes*, *TL*, *sInds*, *fZ*, *fEZ*, *WA*, *mode*, *C_b=None*, *C_sp=None*, *TK=None*)

Finds derivative of achievable dMag with respect to integration time

Parameters

- **intTimes** (*astropy Quantity array*) – Integration times
- **TL** (*TargetList module*) – TargetList class object
- **sInds** (*integer ndarray*) – Integer indices of the stars of interest
- **fZ** (*astropy Quantity array*) – Surface brightness of local zodiacal light for each star in sInds in units of 1/arcsec2
- **fEZ** (*astropy Quantity array*) – Surface brightness of exo-zodiacal light for each star in sInds in units of 1/arcsec2
- **WA** (*astropy Quantity array*) – Working angle for each star in sInds in units of arcsec
- **mode** (*dict*) – Selected observing mode
- **C_b** (*astropy Quantity array*) – Background noise electron count rate in units of 1/s (optional)
- **C_sp** (*astropy Quantity array*) – Residual speckle spatial structure (systematic error) in units of 1/s (optional)
- **TK** (*TimeKeeping object*) – Optional TimeKeeping object (default None), used to model detector degradation effects where applicable.

Returns
Derivative of achievable dMag with respect to integration time

Return type
ddMagdt (ndarray)

int_time_denom_obj(*dMag*, **args*)

Objective function for calc_dMag_per_intTime's calculation of the root of the denominator of calc_inTime to determine the upper bound to use for minimizing to find the correct dMag

Parameters

- **dMag** (*ndarray(float)*) – dMag being tested
- ***args** – all the other arguments that calc_intTime needs

Returns
Denominator of integration time expression

Return type
Quantity(*ndarray*(float))

populate_observingModes_extra()

Add Nemati-specific observing mode keywords

populate_scienceInstruments_extra()

Add Nemati-specific keywords to scienceInstruments

EXOSIMS.OpticalSystem.Nemati_2019 module

class EXOSIMS.OpticalSystem.Nemati_2019.**Nemati_2019**(*k_samp=0.25*, *Nlensl=5*, *lam_d=500*, *lam_c=500*, *MUF_thruput=0.91*, *ContrastScenario='CGDesignPerf'*, ***specs*)

Bases: *Nemati*

Nemati Optical System class

This class contains all variables and methods necessary to perform Optical System Module calculations in exoplanet mission simulation using the model from Nemati 2014.

Parameters

- **k_samp** (*float*) – Default coronagraphic intrinsice sampling. Only used if not set in instrument definition. Defaults to 0.25. TODO: move to starlight suppresion system.
- **Nlensl** (*float*) – Total lenslets covered by PSF core. Only used when not set in science instrument definition. Only applies for spectrometers. Defaults to 5
- **lam_d** (*float*) – Default instrument design wavelength (in nm). Only used when not set in science instrument definition. Defaults to 500
- **lam_c** (*float*) – Default instrument critical wavelength (in nm). Only used when not set in science instrument definition. Defaults to 500
- **MUF_thruput** (*float*) – Model uncertainty factor core throughput. Only used when not set in science instrument definition. Defaults to 0.91
- ****specs** – *Input Specification*

default_vals_extra2

Dictionary of input values to be filled in as defaults in the instrument, starlight supporession system and observing modes. These values are specific to this module.

Type
dict

Cp_Cb_Csp(*TL*, *sInds*, *fZ*, *fEZ*, *dMag*, *WA*, *mode*, *TK=None*, *returnExtra=False*)

Calculates electron count rates for planet signal, background noise, and speckle residuals.

Parameters

- **TL** (*TargetList module*) – TargetList class object
- **sInds** (*integer ndarray*) – Integer indices of the stars of interest
- **fZ** (*astropy Quantity array*) – Surface brightness of local zodiacal light in units of 1/arcsec2

- **fEZ** (*astropy Quantity array*) – Surface brightness of exo-zodiacal light in units of 1/arcsec2
- **dMag** (*float ndarray*) – Differences in magnitude between planets and their host star
- **WA** (*astropy Quantity array*) – Working angles of the planets of interest in units of arcsec
- **mode** (*dict*) – Selected observing mode
- **TK** (*TimeKeeping object*) – Optional TimeKeeping object (default None), used to model detector degradation effects where applicable.
- **returnExtra** (*bool*) – Optional flag, default False, set True to return additional rates for validation

Returns

C_p (astropy.units.Quantity numpy.ndarray):
Planet signal electron count rate in units of 1/s

C_b (astropy.units.Quantity numpy.ndarray):
Background noise electron count rate in units of 1/s

C_sp (astropy.units.Quantity numpy.ndarray):
1/s

Return type
tuple

get_csv_values(*csv_file*, **headers*)

This takes in a csv file and returns the values in the columns associated with the headers given as args

Parameters

- **csv_file** (*str or Path*) – location of the csv file to read
- ***headers** (*str*) – The headers that correspond to the columns of data to be returned

Returns
The values in the columns for every given header. Ordered the same way they were given as inputs

Return type
list

populate_observingModes_extra()

Add Nemati_2019-specific observing mode keywords

populate_scienceInstruments_extra()

Add Nemati_2019-specific keywords to scienceInstruments

2.28.1.5 EXOSIMS.PlanetPhysicalModel package

Submodules

EXOSIMS.PlanetPhysicalModel.Forecaster module

class EXOSIMS.PlanetPhysicalModel.Forecaster.**Forecaster**(*n_pop=4*, ***specs*)

Bases: *FortneyMarleyCahoyMix1*

Planet M-R relation model based on the FORECASTER software, Chen & Kippling 2016.

This module requires to download the fitting_parameters.h5 file from the FORECASTER GitHub repository at https://github.com/chenjj2/forecaster and add it to the PlanetPhysicalModel directory.

Parameters
****specs** – user specified values

calc_radius_from_mass(*Mp*)

Forecast the Radius distribution given the mass distribution.

Parameters
Mp (*astropy Quantity array*) – Planet mass in units of Earth mass

Returns
Planet radius in units of Earth radius

Return type
Rp (astropy Quantity array)

indicate(*M*, *trans*, *i*)

indicate which M belongs to population i given transition parameter

piece_linear(*hyper*, *M*, *prob_R*)

model: straight line

split_hyper_linear(*hyper*)

split hyper and derive c

EXOSIMS.PlanetPhysicalModel.ForecasterMod module

class EXOSIMS.PlanetPhysicalModel.ForecasterMod.**ForecasterMod**(***specs*)

Bases: *FortneyMarleyCahoyMix1*

Planet M-R relation model based on modification of the FORECASTER best-fit model (Chen & Kippling 2016) as described in Savransky et al. (2019)

This modification forces all planets below hydrogen burning to have a maximum radius of 1 R_jupiter and also adds the Saturn density as an explicit point in the model

calc_mass_from_radius(*Rp*)

Calculate planet mass from radius

Parameters
Rp (*astropy Quantity array*) – Planet radius in units of Earth radius

Returns
Planet mass in units of Earth mass

Return type
astropy Quantity array

Note: The fit is non-invertible for Jupiter radii, so all those get 1 Jupiter mass.

`calc_radius_from_mass`(*Mp*)

Calculate planet radius from mass

Parameters
Mp (*astropy Quantity array*) – Planet mass in units of Earth mass

Returns
Planet radius in units of Earth radius

Return type
astropy Quantity array

EXOSIMS.PlanetPhysicalModel.FortneyMarleyCahoyMix1 module

`class` EXOSIMS.PlanetPhysicalModel.FortneyMarleyCahoyMix1.**FortneyMarleyCahoyMix1**(***specs*)

Bases: *PlanetPhysicalModel*

Planet density models based on Fortney & Marley, albedo models based on Cahoy. Intended for use with the Kepler-like planet population modules.

Parameters
****specs** – user specified values

Attributes:

Notes: 1. The calculation of albedo is based solely on the semi-major axis and uses a uniform distribution of metallicities to interpolate albedo from the grid in Cahoy et al. 2010.

`calc_albedo_from_sma`(*a*, *prange*)

Helper function for calculating albedo.

We assume a uniform distribution of metallicities, and then interpolate the grid from Cahoy et al. 2010.

Parameters
a (*astropy Quanitity array*) – Semi-major axis values

Returns
Albedo values

Return type
p (ndarray)

`calc_mass_from_radius`(*Rp*)

Helper function for calculating mass given the radius.

The calculation is done in two steps, first covering all things that can only be ice/rock/iron, and then things that can be giants.

Parameters
Rp (*astropy Quantity array*) – Planet radius in units of Earth radius

Returns
Planet mass in units of Earth mass

Return type
Mp (astropy Quantity array)

`calc_radius_from_mass`(*Mp*)

Helper function for calculating radius given the mass.

The calculation is done in two steps, first covering all things that can only be ice/rock/iron, and then things that can be giants.

Parameters
Mp (*astropy Quantity array*) – Planet mass in units of Earth mass

Returns
Planet radius in units of Earth radius

Return type
Rp (astropy Quantity array)

2.28.1.6 EXOSIMS.PlanetPopulation package

Submodules

EXOSIMS.PlanetPopulation.AlbedoByRadius module

`class` EXOSIMS.PlanetPopulation.AlbedoByRadius.**AlbedoByRadius**(*SAG13coeffs=[[0.38, -0.19, 0.26, 0.0], [0.73, -1.18, 0.59, 3.4]]*, *SAG13starMass=1.0*, *Rprange=[0.6666666666666666, 17.0859375]*, *arange=[0.09084645, 1.45354324]*, *ps=[0.2, 0.5]*, *Rb=[1.4]*, ***specs*)

Bases: *SAG13*

Planet Population module based on SAG13 occurrence rates.

NOTE: This assigns constant albedo based on radius ranges.

`SAG13coeffs`

Coefficients used by the SAG13 broken power law. The 4 lines correspond to Gamma, alpha, beta, and the minimum radius.

Type
float 4x2 ndarray

`Gamma`

Gamma coefficients used by SAG13 broken power law.

Type
float ndarray

`alpha`

Alpha coefficients used by SAG13 broken power law.

Type
float ndarray

`beta`

Beta coefficients used by SAG13 broken power law.

Type
float ndarray

`Rplim`

Minimum radius used by SAG13 broken power law.

Type
float ndarray

`SAG13starMass`

Assumed stellar mass corresponding to the given set of coefficients.

Type
astropy Quantity

`mu`

Gravitational parameter associated with SAG13starMass.

Type
astropy Quantity

`Ca`

Constants used for sampling.

Type
float 2x1 ndarray

`ps`

Constant geometric albedo values.

Type
float nx1 ndarray

`Rb`

Planetary radius break points for albedos in earthRad.

Type
float (n-1)x1 ndarray

`Rbs`

Planetary radius break points with 0 padded on left and np.inf padded on right

Type
float (n+1)x1 ndarray

`gen_plan_params`(*n*)

Generate semi-major axis (AU), eccentricity, geometric albedo, and planetary radius (earthRad)

Semi-major axis and planetary radius are jointly distributed. Eccentricity is a Rayleigh distribution. Albedo is a constant value based on planetary radius.

Parameters
n (*integer*) – Number of samples to generate

Returns

a (astropy Quantity array):
Semi-major axis in units of AU

e (float ndarray):
Eccentricity

p (float ndarray):
Geometric albedo

Rp (astropy Quantity array):
Planetary radius in units of earthRad

Return type
tuple

`get_p_from_Rp`(*Rp*)

Generate constant albedos for radius ranges

Parameters
Rp (*astropy Quantity array*) – Planetary radius with units of earthRad

Returns
Albedo values

Return type
float ndarray

EXOSIMS.PlanetPopulation.AlbedoByRadiusDulzPlavchan module

`class` EXOSIMS.PlanetPopulation.AlbedoByRadiusDulzPlavchan.**AlbedoByRadiusDulzPlavchan**(*starMass=1.0*, *occDataPath=None*, *esigma=0.139629798140*, *ps=[0.2, 0.5]*, *Rb=[1.4]*, ***specs*)

Bases: *DulzPlavchan*

Planet Population module based on occurrence rate tables from Shannon Dulz and Peter Plavchan.

NOTE: This assigns constant albedo based on radius ranges.

`SAG13coeffs`

Coefficients used by the SAG13 broken power law. The 4 lines correspond to Gamma, alpha, beta, and the minimum radius.

Type
float 4x2 ndarray

`Gamma`

Gamma coefficients used by SAG13 broken power law.

Type
float ndarray

`alpha`

Alpha coefficients used by SAG13 broken power law.

Type
float ndarray

`beta`

Beta coefficients used by SAG13 broken power law.

Type
float ndarray

`Rplim`

Minimum radius used by SAG13 broken power law.

Type
float ndarray

`SAG13starMass`

Assumed stellar mass corresponding to the given set of coefficients.

Type
astropy Quantity

`mu`

Gravitational parameter associated with SAG13starMass.

Type
astropy Quantity

`Ca`

Constants used for sampling.

Type
float 2x1 ndarray

`ps`

Constant geometric albedo values.

Type
float nx1 ndarray

`Rb`

Planetary radius break points for albedos in earthRad.

Type
float (n-1)x1 ndarray

`Rbs`

Planetary radius break points with 0 padded on left and np.inf padded on right

Type
float (n+1)x1 ndarray

`gen_plan_params`(*n*)

Generate semi-major axis (AU), eccentricity, geometric albedo, and planetary radius (earthRad)

Semi-major axis and planetary radius are jointly distributed. Eccentricity is a Rayleigh distribution. Albedo is a constant value based on planetary radius.

Parameters
n (*integer*) – Number of samples to generate

Returns

a (astropy Quantity array):
Semi-major axis in units of AU

e (float ndarray):
Eccentricity

p (float ndarray):
Geometric albedo

Rp (astropy Quantity array):
Planetary radius in units of earthRad

Return type
tuple

`get_p_from_Rp`(*Rp*)

Generate constant albedos for radius ranges

Parameters
Rp (*astropy Quantity array*) – Planetary radius with units of earthRad

Returns
Albedo values

Return type
float ndarray

EXOSIMS.PlanetPopulation.Brown2005EarthLike module

`class` EXOSIMS.PlanetPopulation.Brown2005EarthLike.`Brown2005EarthLike`(*eta=1*, *arange=[0.6377303505401009, 1.3665650368716449]*, *erange=[0.0, 0.35]*, *constrainOrbits=False*, ***specs*)

Bases: *PlanetPopulation*

Population of Earth-Like Planets from Brown 2005 paper

This implementation is intended to enforce this population regardless of JSON inputs. The only inputs that will not be disregarded are erange and constrainOrbits.

`gen_plan_params`(*n*)

Generate semi-major axis (AU), eccentricity, geometric albedo, and planetary radius (earthRad)

Semi-major axis and eccentricity are uniformly distributed with all other parameters constant.

Parameters
n (*integer*) – Number of samples to generate

Returns

a (astropy Quantity array):
Semi-major axis in units of AU

e (float ndarray):
Eccentricity

p (float ndarray):
Geometric albedo

Rp (astropy Quantity array):
Planetary radius in units of earthRad

Return type
tuple

`gen_radius_nonorm`(*n*)

Generate planetary radius values in Earth radius. This one just generates a bunch of EarthRad

Parameters
n (*integer*) – Number of target systems. Total number of samples generated will be, on average, n*self.eta

Returns
Planet radius values in units of Earth radius

Return type
astropy Quantity array

EXOSIMS.PlanetPopulation.DulzPlavchan module

`class` EXOSIMS.PlanetPopulation.DulzPlavchan.**DulzPlavchan**(*starMass=1.0*, *occDataPath=None*, *esigma=0.13962979814050144*, ***specs*)

Bases: *PlanetPopulation*

Population based on occurrence rate tables from Shannon Dulz and Peter Plavchan.

The data comes as either Period-Radius or semi-major axis-mass pairs. If occDataPath is not specified, the nominal Period-Radius table is loaded.

Parameters
specs – user specified values

`starMass`
stellar mass in M_sun used to convert period to semi-major axis

`occDataPath`
path on local disk to occurrence rate table

`esigma`
Sigma value of Rayleigh distribution for eccentricity.

Type
float

Notes: 1. Mass/Radius and semi-major axis are specified in occurrence rate tables. User specified values will be ignored. 2. Albedo is sampled as in KeplerLike1 and KeplerLike2. 3. Eccentricity is Rayleigh distributed with user defined sigma parameter.

`MfromRp`(*Rp*)

Converts mass to radius using Chen and Kipping

Parameters
M (*astropy Quantity array*) – Planet mass in units of Earth mass

Returns
Planet radius in units of Earth radius

Return type
astropy Quantity array

RpfromM(*M*)

Converts mass to radius using Chen and Kipping

Parameters
M (*astropy Quantity array*) – Planet mass in units of Earth mass

Returns
Planet radius in units of Earth radius

Return type
Rp (astropy Quantity array)

dist_albedo(*p*)

Probability density function for albedo

Parameters
p (*float ndarray*) – Albedo value(s)

Returns
Albedo probability density

Return type
float ndarray

dist_eccen(*e*)

Probability density function for eccentricity

Parameters
e (*float ndarray*) – Eccentricity value(s)

Returns
Eccentricity probability density

Return type
float ndarray

dist_eccen_from_sma(*e*, *a*)

Probability density function for eccentricity constrained by semi-major axis, such that orbital radius always falls within the provided sma range.

This provides a Rayleigh distribution between the minimum and maximum allowable values.

Parameters

- **e** (*float ndarray*) – Eccentricity values
- **a** (*float ndarray*) – Semi-major axis value in AU. Not an astropy quantity.

Returns
Probability density of eccentricity constrained by semi-major axis

Return type
float ndarray

dist_radius(*Rp*)

Probability density function for planetary radius.

Note that this is a marginalized distribution.

Parameters
Rp (*float ndarray*) – Planetary radius value(s)

Returns
Planetary radius probability density

Return type
float ndarray

`dist_sma`(*a*)

Probability density function for semi-major axis.

Note that this is a marginalized distribution.

Parameters
a (*float ndarray*) – Semi-major axis value(s)

Returns
Semi-major axis probability density

Return type
float ndarray

`gen_albedo`(*n*)

Generate geometric albedo values

The albedo is determined by sampling the semi-major axis distribution, and then calculating the albedo from the physical model.

Parameters
n (*integer*) – Number of samples to generate

Returns
Planet albedo values

Return type
float ndarray

`gen_plan_params`(*n*)

Generate semi-major axis (AU), eccentricity, geometric albedo, and planetary radius (earthRad)

Semi-major axis and planetary radius come from the occurrence rate tables and are assumed to be log-uniformly distributed within bins. Eccentricity is a Rayleigh distribution. Albedo is dependent on the PlanetPhysicalModel but is calculated such that it is independent of other parameters.

Parameters
n (*integer*) – Number of samples to generate

Returns

a (astropy Quantity array):
Semi-major axis in units of AU

e (float ndarray):
Eccentricity

p (float ndarray):
Geometric albedo

Rp (astropy Quantity array):
Planetary radius in units of earthRad

Return type
tuple

`gen_sma_radius`(*n*)

Generates semi-major axis and planetary radius samples

Parameters
n (*int*) – number of samples

Returns

a (astropy Quantity array):
Semi-major axis samples in units of AU

Rp (astropy Quantity array):
Planetary radius samples in units of Earth radius

Return type
tuple

EXOSIMS.PlanetPopulation.EarthTwinHabZone1 module

class EXOSIMS.PlanetPopulation.EarthTwinHabZone1.**EarthTwinHabZone1**(*eta=0.1*, ***specs*)

Bases: *PlanetPopulation*

Population of Earth twins (1 R_Earth, 1 M_Eearth, 1 p_Earth) On circular Habitable zone orbits (0.7 to 1.5 AU)

Note that these values may not be overwritten by user inputs. This implementation is intended to enforce this population regardless of JSON inputs.

dist_sma(*a*)

Probability density function for uniform semi-major axis distribution in AU

Parameters
a (*float ndarray*) – Semi-major axis value(s) in AU. Not an astropy quantity.

Returns
Semi-major axis probability density

Return type
float ndarray

gen_plan_params(*n*)

Generate semi-major axis (AU), eccentricity, geometric albedo, and planetary radius (earthRad)

Semi-major axis is uniformly distributed and all other parameters are constant.

Parameters
n (*integer*) – Number of samples to generate

Returns

a (astropy Quantity array):
Semi-major axis in units of AU

e (float ndarray):
Eccentricity

p (float ndarray):
Geometric albedo

Rp (astropy Quantity array):
Planetary radius in units of earthRad

Return type
tuple

EXOSIMS.PlanetPopulation.EarthTwinHabZone1SDET module

class EXOSIMS.PlanetPopulation.EarthTwinHabZone1SDET.**EarthTwinHabZone1SDET**(*eta=0.1*, ***specs*)

Bases: *PlanetPopulation*

Population of Earth twins (1 R_Earth, 1 M_Eearth, 1 p_Earth) On circular Habitable zone orbits (0.7 to 1.5 AU)

> **Warning:** Note that these values may not be overwritten by user inputs. This implementation is intended to enforce this population regardless of the *Input Specification*.

Parameters

- **eta** (*float*) – Occurrence rate.
- **specs** (*dict*) – *Input Specification*

dist_sma(*a*)

Probability density function for uniform semi-major axis distribution in AU

Parameters
a (*float ndarray*) – Semi-major axis value(s) in AU. Not an astropy quantity.

Returns
Semi-major axis probability density

Return type
float numpy.ndarray

gen_plan_params(*n*)

Generate semi-major axis (AU), eccentricity, geometric albedo, and planetary radius (earthRad)

Semi-major axis is uniformly distributed and all other parameters are constant.

Parameters
n (*int*) – Number of samples to generate

Returns

a (astropy.units.Quantity numpy.ndarray):
Semi-major axis in units of AU

e (float numpy.ndarray):
Eccentricity

p (float numpy.ndarray):
Geometric albedo

Rp (astropy.units.Quantity numpy.ndarray):
Planetary radius in units of earthRad

Return type
tuple

EXOSIMS.PlanetPopulation.EarthTwinHabZone2 module

class EXOSIMS.PlanetPopulation.EarthTwinHabZone2.**EarthTwinHabZone2**(*eta=0.1*, *erange=[0.0, 0.9]*, *constrainOrbits=True*, ***specs*)

Bases: *EarthTwinHabZone1*

Population of Earth twins (1 R_Earth, 1 M_Eearth, 1 p_Earth) On eccentric habitable zone orbits (0.7 to 1.5 AU).

This implementation is intended to enforce this population regardless of JSON inputs. The only inputs that will not be disregarded are erange and constrainOrbits.

gen_plan_params(*n*)

Generate semi-major axis (AU), eccentricity, geometric albedo, and planetary radius (earthRad)

Semi-major axis and eccentricity are uniformly distributed with all other parameters constant.

Parameters
n (*integer*) – Number of samples to generate

Returns

a (astropy Quantity array):
Semi-major axis in units of AU

e (float ndarray):
Eccentricity

p (float ndarray):
Geometric albedo

Rp (astropy Quantity array):
Planetary radius in units of earthRad

Return type
tuple

EXOSIMS.PlanetPopulation.EarthTwinHabZone3 module

class EXOSIMS.PlanetPopulation.EarthTwinHabZone3.**EarthTwinHabZone3**(*eta=0.1*, ***specs*)

Bases: *PlanetPopulation*

Population of Earth twins (1 R_Earth, 1 M_Eearth, 1 p_Earth) On circular Habitable zone orbits (0.75 to 1.77 AU)

Note that these values may not be overwritten by user inputs. This implementation is intended to enforce this population regardless of JSON inputs.

dist_sma(*a*)

Probability density function for uniform semi-major axis distribution in AU

Parameters
a (*float ndarray*) – Semi-major axis value(s) in AU. Not an astropy quantity.

Returns
Semi-major axis probability density

Return type
f (float ndarray)

gen_sma(*n*)

Generate semi-major axis values in AU

The Earth-twin population assumes a uniform distribution between the minimum and maximum values.

Parameters
n (*integer*) – Number of samples to generate

Returns
Semi-major axis values in units of AU

Return type
a (astropy Quantity array)

EXOSIMS.PlanetPopulation.EarthTwinHabZoneSDET module

class EXOSIMS.PlanetPopulation.EarthTwinHabZoneSDET.**EarthTwinHabZoneSDET**(*eta=0.1*, ***specs*)

Bases: *PlanetPopulation*

Population of Earth twins (1 R_Earth, 1 M_Eearth, 1 p_Earth) On circular Habitable zone orbits (0.7 to 1.5 AU)

Note that these values may not be overwritten by user inputs. This implementation is intended to enforce this population regardless of JSON inputs.

dist_radius(*Rp*)

Probability density function for planetary radius in Earth radius

The prototype provides a log-uniform distribution between the minimum and maximum values.

Parameters
Rp (*float ndarray*) – Planetary radius value(s) in Earth radius. Not an astropy quantity.

Returns
Planetary radius probability density

Return type
float ndarray

dist_sma(*a*)

Probability density function for uniform semi-major axis distribution in AU

Parameters
a (*float ndarray*) – Semi-major axis value(s) in AU. Not an astropy quantity.

Returns
Semi-major axis probability density

Return type
f (float ndarray)

gen_plan_params(*n*)

Generate semi-major axis (AU), eccentricity, geometric albedo, and planetary radius (earthRad)

Semi-major axis is uniformly distributed and all other parameters are constant.

Parameters
n (*integer*) – Number of samples to generate

Returns

a (astropy.units.Quantity numpy.ndarray):
Semi-major axis in units of AU

e (float numpy.ndarray):
Eccentricity

p (float numpy.ndarray):
Geometric albedo

Rp (astropy.units.Quantity numpy.ndarray):
Planetary radius in units of earthRad

Return type
tuple

EXOSIMS.PlanetPopulation.Guimond2019 module

class EXOSIMS.PlanetPopulation.Guimond2019.**Guimond2019**(*eta=1*, *arange=[0.1, 50.0]*, *erange=[0.0, 0.999]*, *prange=[0.434, 0.434]*, *constrainOrbits=False*, ***specs*)

Bases: *PlanetPopulation*

Population of Earth-Like Planets from Brown 2005 paper

This implementation is intended to enforce this population regardless of JSON inputs. The only inputs that will not be disregarded are erange and constrainOrbits.

gen_plan_params(*n*)

Generate semi-major axis (AU), eccentricity, geometric albedo, and planetary radius (earthRad)

Semi-major axis and eccentricity are uniformly distributed with all other parameters constant.

Parameters
n (*integer*) – Number of samples to generate

Returns

a (astropy Quantity array):
Semi-major axis in units of AU

e (float ndarray):
Eccentricity

p (float ndarray):
Geometric albedo

Rp (astropy Quantity array):
Planetary radius in units of earthRad

Return type
tuple

gen_radius_nonorm(*n*)

Generate planetary radius values in Earth radius. This one just generates a bunch of EarthRad

Parameters
n (*integer*) – Number of target systems. Total number of samples generated will be, on average, n*self.eta

Returns
Planet radius values in units of Earth radius

Return type
astropy Quantity array

loguniform(*low=0.001*, *high=1*, *size=None*)

Parameters

- **float** –
- **float** – upper end of samples to generate
- **integer** – number of values to sample

Returns
of logarithmically distributed values

Return type
ndarray

EXOSIMS.PlanetPopulation.JupiterTwin module

class EXOSIMS.PlanetPopulation.JupiterTwin.**JupiterTwin**(*eta=1*, *erange=[0.0, 0.048]*, *constrainOrbits=True*, ***specs*)

Bases: *PlanetPopulation*

Population of Jupiter twins (11.209 R_Earth, 317.83 M_Eearth, 1 p_Earth) On eccentric orbits (0.7 to 1.5 AU)*5.204. Numbers pulled from nssdc.gsfc.nasa.gov/planetary/factsheet/jupiterfact.html

This implementation is intended to enforce this population regardless of JSON inputs. The only inputs that will not be disregarded are erange and constrainOrbits.

gen_plan_params(*n*)

Generate semi-major axis (AU), eccentricity, geometric albedo, and planetary radius (earthRad)

Semi-major axis and eccentricity are uniformly distributed with all other parameters constant.

Parameters
n (*integer*) – Number of samples to generate

Returns

a (astropy Quantity array):
Semi-major axis in units of AU

e (float ndarray):
Eccentricity

p (float ndarray):
Geometric albedo

Rp (astropy Quantity array):
Planetary radius in units of earthRad

Return type
tuple

EXOSIMS.PlanetPopulation.KeplerLike1 module

class EXOSIMS.PlanetPopulation.KeplerLike1.**KeplerLike1**(*smaknee=30*, *esigma=0.13962979814050144*, *prange=[0.083, 0.882]*, *Rprange=[1, 22.6]*, ***specs*)

Bases: *PlanetPopulation*

Population based on Kepler radius distribution with RV-like semi-major axis distribution with exponential decay.

Parameters
****specs** – user specified values

smaknee

Location (in AU) of semi-major axis decay point (knee). Not an astropy quantity.

Type
float

esigma

Sigma value of Rayleigh distribution for eccentricity.

Type
float

Notes: 1. The gen_mass function samples the Radius and calculates the mass from there. Any user-set mass limits are ignored. 2. The gen_albedo function samples the sma, and then calculates the albedos from there. Any user-set albedo limits are ignored. 3. The Rprange is fixed to (1,22.6) R_Earth and cannot be overwritten by user settings (the JSON input will be ignored) 4. The radius piece-wise distribution (from Fressin et al 2012) provides the normalization required to get the proper overall eta. The gen_radius method provided here normalizes in order to return exactly the number of samples requested. A second method (gen_radius_nonorm) is provided for generating the simulated universe population. The latter assumes a poisson distribution for occurences in each bin. 5. Eccentricity is assumed to be Rayleigh distributed with a user-settable sigma parameter (defaults to value from Fressin et al 2012).

dist_albedo(*p*)

Probability density function for albedo

Parameters
p (*float ndarray*) – Albedo value(s)

Returns
Albedo probability density

Return type
float ndarray

dist_eccen(*e*)

Probability density function for eccentricity

Parameters
e (*float ndarray*) – Eccentricity value(s)

Returns
Eccentricity probability density

Return type
float ndarray

`dist_eccen_from_sma`(*e*, *a*)

Probability density function for eccentricity constrained by semi-major axis, such that orbital radius always falls within the provided sma range.

This provides a Rayleigh distribution between the minimum and maximum allowable values.

Parameters

- **e** (*float ndarray*) – Eccentricity values
- **a** (*float ndarray*) – Semi-major axis value in AU. Not an astropy quantity.

Returns
Probability density of eccentricity constrained by semi-major axis

Return type
float ndarray

`dist_radius`(*Rp*)

Probability density function for planetary radius in Earth radius

Parameters
Rp (*float ndarray*) – Planetary radius value(s) in Earth radius. Not an astropy quantity.

Returns
Planetary radius probability density

Return type
float ndarray

`dist_sma`(*a*)

Probability density function for semi-major axis in AU

Parameters
a (*float ndarray*) – Semi-major axis value(s) in AU. Not an astropy quantity.

Returns
Semi-major axis probability density

Return type
float ndarray

`gen_albedo`(*n*)

Generate geometric albedo values

The albedo is determined by sampling the semi-major axis distribution, and then calculating the albedo from the physical model.

Parameters
n (*integer*) – Number of samples to generate

Returns
Planet albedo values

Return type
float ndarray

`gen_mass`(*n*)

Generate planetary mass values in Earth Mass

The mass is determined by sampling the radius and then calculating the mass from the physical model.

Parameters
n (*integer*) – Number of samples to generate

Returns
Planet mass values in units of Earth mass

Return type
astropy Quantity array

`gen_plan_params`(*n*)

Generate semi-major axis (AU), eccentricity, geometric albedo, and planetary radius (earthRad)

Semi-major axis is distributed RV like with exponential decay. Eccentricity is a Rayleigh distribution. Albedo is dependent on the PlanetPhysicalModel but is calculated such that it is independent of other parameters. Planetary radius comes from the Kepler observations.

Parameters
n (*integer*) – Number of samples to generate

Returns

a (astropy Quantity array):
Semi-major axis in units of AU

e (float ndarray):
Eccentricity

p (float ndarray):
Geometric albedo

Rp (astropy Quantity array):
Planetary radius in units of earthRad

Return type
tuple

`gen_radius`(*n*)

Generate planetary radius values in Earth radius

Samples a radius distribution defined as log-uniform in each of 9 radius bins with fixed occurrence rates.

Parameters
n (*integer*) – Number of samples to generate

Returns
Planet radius values in units of Earth radius

Return type
astropy Quantity array

`gen_radius_nonorm`(*n*)

Generate planetary radius values in Earth radius.

Samples a radius distribution defined as log-uniform in each of 9 radius bins with fixed occurrence rates. The rates in the bins determine the overall occurrence rates of all planets.

Parameters
n (*integer*) – Number of target systems. Total number of samples generated will be, on average, n*self.eta

Returns
Planet radius values in units of Earth radius

Return type
astropy Quantity array

gen_sma(*n*)

Generate semi-major axis values in AU

Samples a power law distribution with exponential turn-off determined by class attribute smaknee

Parameters
n (*integer*) – Number of samples to generate

Returns
Semi-major axis values in units of AU

Return type
astropy Quantity array

EXOSIMS.PlanetPopulation.KeplerLike2 module

class EXOSIMS.PlanetPopulation.KeplerLike2.**KeplerLike2**(*smaknee=30*, *esigma=0.25*, ***specs*)

Bases: *KeplerLike1*

Population based on Kepler radius distribution with RV-like semi-major axis distribution with exponential decay.

NOTE: This is an exact clone of KeplerLike1, but uses (approximate) inverse transform sampling instead of simple rejection sampling for performance improvements.

Parameters
****specs** – user specified values

smaknee

Location (in AU) of semi-major axis decay point (knee). Not an astropy quantity.

Type
float

esigma

Sigma value of Rayleigh distribution for eccentricity.

Type
float

Notes: 1. The gen_mass function samples the Radius and calculates the mass from there. Any user-set mass limits are ignored. 2. The gen_albedo function samples the sma, and then calculates the albedos from there. Any user-set albedo limits are ignored. 3. The Rprange is fixed to (1,22.6) R_Earth and cannot be overwritten by user settings (the JSON input will be ignored) 4. The radius piece-wise distribution provides the normalization required to get the proper overall eta. The gen_radius method provided here normalizes in order to return exactly the number of samples requested. A second method (gen_radius_nonorm) is provided for generating the simulated universe population. The latter assumes a poisson distribution for occurences in each bin. 5. Eccentricity is assumed to be Rayleigh distributed with a user-settable sigma parameter (defaults to 0.25).

gen_sma(*n*)

Generate semi-major axis values in AU

Samples a power law distribution with exponential turn-off determined by class attribute smaknee

Parameters
n (*integer*) – Number of samples to generate

Returns
Semi-major axis in units of AU

Return type
a (astropy Quantity array)

EXOSIMS.PlanetPopulation.KnownRVPlanets module

class EXOSIMS.PlanetPopulation.KnownRVPlanets.**KnownRVPlanets**(*smaknee=30*, *esigma=0.25*, *rvplanetfilepath=None*, *planetfile='planets_2019.05.31_11.18.02.votable'*, ***specs*)

Bases: *KeplerLike1*

Population consisting only of known RV planets. Eccentricity and sma distributions are taken from KeplerLike1 (Rayleigh and power law with exponential decay, respectively). Mass is sampled from power law and radius is assumed to be calculated from mass via the physical model.

The data file read in by this class also provides all of the information about the target stars, and so no StarCatalog object is needed (only the KnownRvPlanetsTargetList implementation).

To download a new copy of the data file:

1. Navigate to the IPAC exoplanet archive at http://exoplanetarchive.ipac.caltech.edu/cgi-bin/TblView/nph-tblView?app=ExoTbls&config=planets
2. Type 'radial' (minus quotes) in the 'Discovery Method' search box and hit enter.
3. In the 'Download Table' menu select 'VOTable Format', 'Download all Columns' and 'Download Currently Filtered Rows'.
4. In the 'Download Table' menu click 'Download Table'.

Parameters
****specs** – user specified values

smaknee
Location (in AU) of semi-major axis decay point (knee). Not an astropy quantity.

Type
float

esigma
Sigma value of Rayleigh distribution for eccentricity.

Type
float

rvplanetfilepath
Full path to RV planet votable file from IPAC. If None, assumes default file in PlanetPopulation directory of EXOSIMS.

Type
string

period
Orbital period in units of day. Error in perioderr.

Type
astropy Quantity array

planetfile

Name of input file to use

Type
str

tper

Periastron time in units of jd. Error in tpererr.

Type
astropy Time

Notes:

gen_mass(*n*)

Generate planetary mass values in Earth mass

The mass is determined by sampling the RV mass distribution from Cumming et al. 2010

Parameters
n (*integer*) – Number of samples to generate

Returns
Planet mass values in units of Earth mass

Return type
Mp (astropy Quantity array)

gen_radius(*n*)

Generate planetary radius values in Earth radius

Samples the mass distribution and then converts to radius using the physical model.

Parameters
n (*integer*) – Number of samples to generate

Returns
Planet radius values in units of Earth radius

Return type
Rp (astropy Quantity array)

EXOSIMS.PlanetPopulation.SAG13 module

class EXOSIMS.PlanetPopulation.SAG13.**SAG13**(*SAG13coeffs=[[0.38, -0.19, 0.26, 0.0], [0.73, -1.18, 0.59, 3.4]]*, *SAG13starMass=1.0*, *Rprange=[0.6666666666666666, 17.0859375]*, *arange=[0.09084645, 1.45354324]*, ***specs*)

Bases: *KeplerLike2*

Planet Population module based on SAG13 occurrence rates.

This is the current working model based on averaging multiple studies. These do not yet represent official scientific values.

SAG13coeffs

Coefficients used by the SAG13 broken power law. The 4 lines correspond to Gamma, alpha, beta, and the minimum radius.

Type
float 4x2 ndarray

Gamma

Gamma coefficients used by SAG13 broken power law.

Type
float ndarray

alpha

Alpha coefficients used by SAG13 broken power law.

Type
float ndarray

beta

Beta coefficients used by SAG13 broken power law.

Type
float ndarray

Rplim

Minimum radius used by SAG13 broken power law.

Type
float ndarray

SAG13starMass

Assumed stellar mass corresponding to the given set of coefficients.

Type
astropy Quantity

mu

Gravitational parameter associated with SAG13starMass.

Type
astropy Quantity

Ca

Constants used for sampling.

Type
float 2x1 ndarray

dist_radius(*Rp*)

Marginalized probability density function for planetary radius in Earth radius.

Parameters
Rp (*float ndarray*) – Planetary radius value(s) in Earth radius. Not an astropy quantity.

Returns
Planetary radius probability density

Return type
float ndarray

dist_sma(*a*)

Marginalized probability density function for semi-major axis in AU.

Parameters
a (*float ndarray*) – Semi-major axis value(s) in AU. Not an astropy quantity.

Returns
Semi-major axis probability density

Return type
float ndarray

`dist_sma_given_radius`(*a*, *beta*, *m*, *C*, *smaknee*)

Conditional probability density function of semi-major axis given planetary radius.

Parameters

- **a** (*float ndarray*) – Semi-major axis value(s) in AU. Not an astropy quantity
- **beta** (*float*) – Exponent for distribution
- **m** (*float*) – Gravitational parameter (AU3/year2)
- **C** (*float*) – Normalization for distribution
- **smaknee** (*float*) – Coefficient for decay

Returns
Probability density

Return type
float ndarray

`dist_sma_radius`(*a*, *R*)

Joint probability density function for semi-major axis (AU) and planetary radius in Earth radius.

This method performs a change of variables on the SAG13 broken power law (originally in planetary radius and period).

Parameters

- **a** (*float ndarray*) – Semi-major axis values in AU. Not an astropy quantity
- **R** (*float ndarray*) – Planetary radius values in Earth radius. Not an astropy quantity

Returns
Joint (semi-major axis and planetary radius) probability density matrix of shape (len(R),len(a))

Return type
float ndarray

`gen_plan_params`(*n*)

Generate semi-major axis (AU), eccentricity, geometric albedo, and planetary radius (earthRad)

Semi-major axis and planetary radius are jointly distributed. Eccentricity is a Rayleigh distribution. Albedo is dependent on the PlanetPhysicalModel but is calculated such that it is independent of other parameters.

Parameters
n (*integer*) – Number of samples to generate

Returns

a (astropy Quantity array):
Semi-major axis in units of AU

e (float ndarray):
Eccentricity

p (float ndarray):
Geometric albedo

Rp (astropy Quantity array):
Planetary radius in units of earthRad

Return type
tuple

`gen_radius_sma`(*n*)

Generate radius values in earth radius and semi-major axis values in AU.

This method is called by gen_radius and gen_sma.

Parameters
n (*integer*) – Number of samples to generate

Returns

Rp (astropy Quantity array):
Planet radius values in units of Earth radius

a (astropy Quantity array):
Semi-major axis values in units of AU

Return type
tuple

EXOSIMS.PlanetPopulation.SolarSystem module

`class` EXOSIMS.PlanetPopulation.SolarSystem.**SolarSystem**(*prange=[0.1, 0.7]*, *Rprange=[0.01, 30.0]*, ***specs*)

Bases: *PlanetPopulation*

Population of Earth-Like Planets from Brown 2005 paper

This implementation is intended to enforce this population regardless of JSON inputs. The only inputs that will not be disregarded are erange and constrainOrbits.

`gen_plan_params`(*nPlans*)

Values taken From mallama2018PlantProperties Seidlemenn 1992 :param float: nPlans, the number of planets

2.28.1.7 EXOSIMS.PostProcessing package

Submodules

EXOSIMS.PostProcessing.PostProcessing2 module

`class` EXOSIMS.PostProcessing.PostProcessing2.**PostProcessing2**(***specs*)

Bases: *PostProcessing*

PostProcessing2 class

Updated PostProcessing det_occur function that utilized BackgroundSource module GalaxiesFaintStars to calculate FA probability.

`det_occur`(*SNR*, *mode*, *TL*, *sInd*, *intTime*)

Determines if a detection has occurred and returns booleans

This method returns two booleans where True gives the case.

Parameters

- **SNR** (*float ndarray*) – signal-to-noise ratio of the planets around the selected target
- **mode** (*dict*) – Selected observing mode
- **TL** (*TargetList module*) – TargetList class object
- **sInd** (*integer*) – Index of the star being observed
- **intTime** (*astropy Quantity*) – Selected star integration time for detection

Returns

FA (boolean):
False alarm (false positive) boolean.

MD (boolean ndarray):
Missed detection (false negative) boolean with the size of number of planets around the target.

Return type
tuple

Note: This implementation assumes the dark hole is set by intCutoff_dMag. Alternatively, the true integration depth could be calculated from the integration time.

2.28.1.8 EXOSIMS.Prototypes package

Submodules

EXOSIMS.Prototypes.BackgroundSources module

class EXOSIMS.Prototypes.BackgroundSources.**BackgroundSources**(*cachedir=None*, ***specs*)

Bases: object

BackgroundSources Prototype

Parameters

- **cachedir** (*str, optional*) – Full path to cachedir. If None (default) use default (see *Cache Directory*)
- ****specs** – *Input Specification*

cachedir

Path to the EXOSIMS cache directory (see *Cache Directory*)

Type
str

_outspec

Output Specification

Type
dict

dNbackground(*coords*, *intDepths*)

Returns background source number densities

Parameters

- **coords** (*SkyCoord*) – SkyCoord object containing right ascension, declination, and distance to star of the planets of interest in units of deg, deg and pc
- **intDepths** (*ndarray(float)*) – Integration depths equal to the planet magnitude (Vmag+dMag), i.e. the V magnitude of the dark hole to be produced for each target. Must be of same length as coords.

Returns
dN: Number densities of background sources for given targets in units of 1/arcmin2. Same length as inputs.

Return type
Quantity(*ndarray*(float))

EXOSIMS.Prototypes.Completeness module

class EXOSIMS.Prototypes.Completeness.**Completeness**(*minComp=0.1*, *cachedir=None*, ***specs*)

Bases: object

Completeness Prototype

Parameters

- **minComp** (*float*) – Minimum completeness for target filtering. Defaults to 0.1.
- **cachedir** (*str, optional*) – Full path to cachedir. If None (default) use default (see *Cache Directory*)
- ****specs** – *Input Specification*

_outspec
Output Specification

Type
dict

cachedir
Path to the EXOSIMS cache directory (see *Cache Directory*)

Type
str

minComp
Minimum completeness value for inclusion in target list

Type
float

PlanetPhysicalModel
Planet physical model object

Type
PlanetPhysicalModel

PlanetPopulation
Planet population object

Type
PlanetPopulation

updates

Dynamic completeness updates array for revisists.

Type
numpy.ndarray

comp_calc(*smin*, *smax*, *dMag*)

Calculates completeness for given minimum and maximum separations and dMag.

Parameters

- **smin** (*ndarray(float)*) – Minimum separation(s) in AU
- **smax** (*ndarray(float)*) – Maximum separation(s) in AU
- **dMag** (*ndarray(float)*) – Difference in brightness magnitude

Returns
Completeness values

Return type
ndarray(float)

> **Warning:** The prototype implementation does not perform any real completeness calculations. To be used when you need a completeness object but do not care about the actual values.

comp_per_intTime(*intTimes*, *TL*, *sInds*, *fZ*, *fEZ*, *WA*, *mode*, *C_b=None*, *C_sp=None*, *TK=None*)

Calculates completeness values per integration time

Note: Prototype does no calculations and always returns the same value

Parameters

- **intTimes** (*Quantity(ndarray(float))*) – Integration times
- **TL** (*TargetList*) – TargetList object
- **sInds** (*ndarray(int)*) – Integer indices of the stars of interest
- **fZ** (*Quantity(ndarray(float))*) – Surface brightness of local zodiacal light in units of 1/arcsec2
- **fEZ** (*Quantity(ndarray(float))*) – Surface brightness of exo-zodiacal light in units of 1/arcsec2
- **WA** (*Quantity(ndarray(float))*) – Working angle of the planet of interest in units of arcsec
- **mode** (*dict*) – Selected observing mode
- **C_b** (*Quantity(ndarray(float)), optional*) – Background noise electron count rate in units of 1/s
- **C_sp** (*Quantity(ndarray(float)), optional*) – Residual speckle spatial structure (systematic error) in units of 1/s

Returns
Completeness values

Return type
ndarray(float)

completeness_setup()

Preform any preliminary calculations needed for this flavor of completeness

For the Prototype, this is just a dummy function for later overloading

completeness_update(*TL*, *sInds*, *visits*, *dt*)

Updates completeness value for stars previously observed

Parameters

- **TL** (*TargetList*) – TargetList class object
- **sInds** (*ndarray(int)*) – Indices of stars to update
- **visits** (*ndarray(int)*) – Number of visits for each star
- **dt** (*Quantity(ndarray(float))*) – Time since previous observation

Returns
Completeness values for each star

Return type
ndarray(float)

dcomp_dt(*intTimes*, *TL*, *sInds*, *fZ*, *fEZ*, *WA*, *mode*, *C_b=None*, *C_sp=None*, *TK=None*)

Calculates derivative of completeness with respect to integration time

Note: Prototype does no calculations and always returns the same value

Parameters

- **intTimes** (*Quantity(ndarray(float))*) – Integration times
- **TL** (*TargetList*) – TargetList class object
- **sInds** (*ndarray(int)*) – Integer indices of the stars of interest
- **fZ** (*Quantity(ndarray(float))*) – Surface brightness of local zodiacal light in units of 1/arcsec2
- **fEZ** (*Quantity(ndarray(float))*) – Surface brightness of exo-zodiacal light in units of 1/arcsec2
- **WA** (*Quantity(ndarray(float))*) – Working angle of the planet of interest in units of arcsec
- **mode** (*dict*) – Selected observing mode

Returns
Derivative of completeness with respect to integration time (units 1/time)

Return type
Quantity(*ndarray*(float))

gen_update(*TL*)

Generates any information necessary for dynamic completeness calculations (completeness at successive observations of a star in the target list)

Parameters
TL (*TargetList*) – TargetList object

Returns
None

generate_cache_names(***specs*)

Generate unique filenames for cached products

revise_updates(*ind*)

Keeps completeness update values only for targets remaining in target list during filtering (called from TargetList.filter_target_list)

Parameters
ind (*ndarray(int)*) – array of indices to keep

target_completeness(*TL*)

Generates completeness values for target stars

This method is called from TargetList __init__ method.

Parameters
TL (*TargetList*) – TargetList object

Returns
Completeness values for each target star

Return type
ndarray(float)

Warning: The prototype implementation does not perform any real completeness calculations. To be used when you need a completeness object but do not care about the actual values.

EXOSIMS.Prototypes.Observatory module

class EXOSIMS.Prototypes.Observatory.**Observatory**(*SRP=True*, *koAngles_SolarPanel=[0, 180]*, *ko_dtStep=1*, *settlingTime=1*, *thrust=450*, *slewIsp=4160.0*, *scMass=6000.0*, *slewMass=0.0*, *skMass=0.0*, *twotanks=False*, *skEff=0.7098*, *slewEff=1.0*, *dryMass=3400.0*, *coMass=5800.0*, *occulterSep=55000.0*, *skIsp=220.0*, *defburnPortion=0.05*, *constTOF=14*, *maxdVpct=0.02*, *spkpath=None*, *checkKeepoutEnd=True*, *forceStaticEphem=False*, *occ_dtmin=0.055*, *occ_dtmax=61.0*, *sk_Tmin=0.0*, *sk_Tmax=365.0*, *non_lambertian_coefficient_front=0.038*, *non_lambertian_coefficient_back=0.004*, *specular_reflection_factor=0.975*, *nreflection_coefficient=0.999*, *emission_coefficient_front=0.8*, *emission_coefficient_back=0.2*, *allowRefueling=False*, *external_fuel_mass=0*, *cachedir=None*, ***specs*)

Bases: object

Observatory Prototype

Parameters

- **SRP** (*bool*) – Toggle solar radiation pressure. Defaults True.

- **koAngles_SolarPanel** (*list(float)*) – [Min, Max] keepout angles (in degrees) due to solar panels. Defaults to [0,180].
- **ko_dtStep** (*float*) – Step size to use when calculating keepout maps (in days). Defaults to 1.
- **settlingTime** (*float*) – Observatory settling time after retargeting (in days). Defaults to 1. This time is added to every observation and counts againts the total integration time allocation.
- **thrust** (*float*) – Slew thrust mangitude (in mN). Defaults to 450 mN.
- **slewIsp** (*float*) – Slew specific impulse (in seconds). Defaults to 4160 s.
- **scMass** (*float*) – Maneuvering spacecraft initial wet mass (in kg). Nominally this is the starshade, but may also be the observatory if the starshade is kept on the stable orbit. Defaults to 6000 kg.

  Warning: If `twotanks` is true, this input will be ignored and attribute `scMass` will be initially set to the sum of `slewMass` and `skMass`.

- **slewMass** (*float*) – Initial fuel mass of slewing propulsion system (in kg). Defaults to 0. Only used if twotanks is True.
- **skMass** (*float*) – Initial fuel mass of stationkeeping propulsion system (in kg). Defaults to 0. Only used if twotanks is True.
- **twotanks** (*bool*) – Determines whether stationkeeping and slewing propulsion systems use separate tanks. If False, it is assumed that all onboard fuel is fungible. Defaults False.
- **skEff** (*float*) – Stationkeeping propulsion system efficiency. Must be between 0 and 1. Defaults to 0.7098 (approximately 45 deg cosine losses).
- **slewEff** (*float*) – Slewing propulsion system efficiency. Must be between 0 and 1. Defaults to 1.
- **dryMass** (*float*) – Maneuvering spacecraft dry mass (in kg). Defaults to 3400 kg. Must be smaller than scMass.
- **coMass** (*float*) – Non-manuevering spacecraft (nominally the observatory) initial wet mass (in kg). Defaults to 5800 kg.
- **occulterSep** (*float*) – Initial occulter separation (in km). Defaults to 55000.
- **skIsp** (*float*) – Stationkeeping propulsion system specific impulse (in seconds). Defaults to 220 s.
- **defburnPortion** (*float*) – Default burn portion for simple model slews. Must be between 0 and 1. Defaults to 0.05.
- **constTOF** (*float*) – Constant time of flight value (in days). Defaults to 14. DEPRECATED
- **maxdVpct** (*float*) – Maximum delta V percentage allowed for any maneuver. Must be between 0 and 1. Defaults to 0.02.
- **spkpath** (*str, optional*) – Path to SPK file on disk. If not set, defaults to de432s.bsp in *Downloads Directory*.
- **checkKeepoutEnd** (*bool*) – Check keepout conditions at end of observation. Defaults True. TODO: Move to SurveySimulation

- **forceStaticEphem** (*bool*) – Use static ephemerides for solar system objects instead of jplephem. Defaults False.
- **occ_dtmin** (*float*) – Minimum slew time (in days). Defaults to 0.055
- **occ_dtmax** (*float*) – Maximum slew time (in days). Defaults to 61
- **sk_Tmin** (*float*) – Minimum time after mission start to compute stationkeeping (in days). Defaults to 0.
- **sk_Tmax** (*float*) – Maximum time after mission start to compute stationkeeping (in days). Defaults to 365
- **non_lambertian_coefficient_front** (*float*) – Non-Lambertion reflectivity coefficient of front face of manuevering spacecraft. Used for SRP calculations. Defaults to 0.038.
- **non_lambertian_coefficient_back** (*float*) – Non-Lambertion reflectivity coefficient of back face of manuevering spacecraft. Used for SRP calculations. Defaults to 0.004.
- **specular_reflection_factor** (*float*) – Specular reflectivity of maneuvering spacecraft. Used for SRP calculations. Defaults to 0.975.
- **nreflection_coefficient** (*float*) – non-specular reflectivity of maneuvering spacecraft. Used for SRP calculations. Defaults to 0.999
- **emission_coefficient_front** (*float*) – Emission coefficient of front face of maneuvering spacecraft. Used for SRP calculations. Defaults to 0.8
- **emission_coefficient_back** (*float*) – Emission coefficient of rear face of maneuvering spacecraft. Used for SRP calculations. Defaults to 0.2
- **allowRefueling** (*bool*) – Fuel tanks can be topped off when they reach empty from external fuel source whose capacity is defined by the `external_fuel_mass` input. Defaults False.
- **external_fuel_mass** (*float*) – Initial mass of external fuel supply (in kg). Ignored if `allowRefueling` is False. Defaults to 0.
- **cachedir** (*str, optional*) – Full path to cachedir. If None (default) use default (see *Cache Directory*)
- ****specs** – *Input Specification*

_outspec

Output Specification

Type
dict

ao

Thurst acceleration (current thrust/spacecraft mass). Acceleration units.

Type
astropy.units.quantity.Quantity

cachedir

Path to the EXOSIMS cache directory (see *Cache Directory*)

Type
str

checkKeepoutEnd

Toggle checking of keepout at end of observations (as well as the beginning). TODO: need to depricate in favor of continuous visibility.

Type
bool

coMass

Non-manuevering spacecraft (nominally the observatory) wet mass. Mass units.

Type
astropy.units.quantity.Quantity

constTOF

Constant time of flight for single occulter slew. DEPRECATED

Type
astropy.units.quantity.Quantity

defburnPortion

Default burn portion for simple slew model.

Type
float

dryMass

Maneuvering spacecraft dry mass. Mass units.

Type
astropy.units.quantity.Quantity

dVmax

Maximum single-slew allowable delta V (as determined by maxdVpct input. Units of velocity.

Type
astropy.units.quantity.Quantity

dVtot

Total possible slew delta V as determined by ideal rocket equation applied to the initial fuel mass.

Type
astropy.units.quantity.Quantity

emission_coefficient_back

Emission coefficient of back face of maneuvering spacecraft. Used for SRP calculations.

Type
float

emission_coefficient_front

Emission coefficient of front face of maneuvering spacecraft. Used for SRP calculations.

Type
float

flowRate

Slew ropulsion system mass flow rate. Units: mass/time

Type
astropy.units.quantity.Quantity

forceStaticEphem

Use static ephemerides for solar system objects instead of jplephem.

Type
bool

havejplephem

jplephem module installed and SPK available.

Type
bool

kernel

jplephem kernel used for ephemeris calculations of solar system bodies

Type
jplephem.spk.SPK

ko_dtStep

Step size to use when calculating keepout maps. Time units.

Type
astropy.units.quantity.Quantity

koAngles_SolarPanel

[Min, Max] keepout angles due to solar panels.

Type
astropy.units.quantity.Quantity

maxdVpct

Maximum delta V percentage allowed for any maneuver.

Type
float

maxFuelMass

Total capacity of all fuel. This parameter is constant and should never be modified externally.

Type
astropy.units.quantity.Quantity

non_lambertian_coefficient_back

Non-Lambertion reflectivity coefficient of back face of manuevering spacecraft. Used for SRP calculations.

Type
float

non_lambertian_coefficient_front

Non-Lambertion reflectivity coefficient of front face of manuevering spacecraft. Used for SRP calculations.

Type
float

nreflection_coefficient

Non-specular reflectivity of maneuvering spacecraft. Used for SRP calculations.

Type
float

occ_dtmax

Maximum allowable slew time

Type
astropy.units.quantity.Quantity

occ_dtmin

Minimum allowable slew time.

Type
astropy.units.quantity.Quantity

occulterSep

Current occulter separation distance.

Type
astropy.units.quantity.Quantity

scMass

Current maneuvering spacecraft mass.

Type
astropy.units.quantity.Quantity

settlingTime

Observatory settling time after every retargeting.

Type
astropy.units.quantity.Quantity

sk_Tmax

Maximum time after mission start to compute stationkeeping

Type
astropy.units.quantity.Quantity

sk_Tmin

Minimum time after mission start to compute stationkeeping

Type
astropy.units.quantity.Quantity

skEff

Stationkeeping propulsion system efficience.

Type
float

skIsp

Stationkeeping propulsion system specific impulse. Time units.

Type
astropy.units.quantity.Quantity

skMass

Stationkeeping propulsion system fuel mass.

Type
astropy.units.quantity.Quantity

skMaxFuelMass

Total capacity of stationkeeping fuel tank. This parameter is constant and should never be modified externally. Set to 0 if allowRefueling is False.

Type
astropy.units.quantity.Quantity

slewEff

Slew propulsion system efficiencey.

Type
float

slewIsp

Slew propulsion system specific impulse. Time units.

Type
astropy.units.quantity.Quantity

slewMass

Slew propulsion system fuel mass.

Type
astropy.units.quantity.Quantity

slesMaxFuelMass

Total capacity of slewing fuel tank. This parameter is constant and should never be modified externally. Set to 0 if allowRefueling is False.

Type
astropy.units.quantity.Quantity

specular_reflection_factor

Specular reflectivity of maneuvering spacecraft. Used for SRP calculations.

Type
float

spkpath

Full path to SPK file used by jplephem.

Type
str

SRP

Toggles whether solar radiation pressure is included in calculations.

Type
bool

thrust

Slew propulsion system thrust. Force units.

Type
astropy.units.quantity.Quantity

twotanks

Toggles whether stationkeeping and slewing fuel are bookkept separately. If false, all fuel is fungible.

Type
bool

Note: For finding positions of solar system bodies, this routine will attempt to use the jplephem module and a local SPK file on disk. The module can be installed via pip or from source. The default SPK file (which the code attempts to automatically download) can be downloaded manually from: http://naif.jpl.nasa.gov/pub/

naif/generic_kernels/spk/planets/de432s.bsp and should be placed in the *Downloads Directory* (or another path, specified by the `spkpath` input).

---

class SolarEph(*a*, *e*, *I*, *O*, *w*, *lM*)

Bases: `object`

Solar system ephemerides class

This class takes the constants in Appendix D.4 of Vallado as inputs and stores them for use in defining solar system ephemerides at a given time.

Parameters

- **a** (*list*) – semimajor axis list (in AU)
- **e** (*list*) – eccentricity list
- **I** (*list*) – inclination list
- **O** (*list*) – right ascension of the ascending node list
- **w** (*list*) – longitude of periapsis list
- **lM** (*list*) – mean longitude list

Each of these lists has a maximum of 4 elements. The values in these lists are used to propagate the solar system planetary ephemerides for a specific solar system planet.

a

list of semimajor axis (in AU)

Type
list

e

list of eccentricity

Type
list

I

list of inclination

Type
list

O

list of right ascension of the ascending node

Type
list

w

list of longitude of periapsis

Type
list

lM

list of mean longitude values

Type
list

Each of these lists has a maximum of 4 elements. The values in these lists are used to propagate the solar system planetary ephemerides for a specific solar system planet.

calculate_dV(*TL*, *old_sInd*, *sInds*, *sd*, *slewTimes*, *tmpCurrentTimeAbs*)

Finds the change in velocity needed to transfer to a new star line of sight

This method sums the total delta-V needed to transfer from one star line of sight to another. It determines the change in velocity to move from one station-keeping orbit to a transfer orbit at the current time, then from the transfer orbit to the next station-keeping orbit at currentTime + dt. Station-keeping orbits are modeled as discrete boundary value problems. This method can handle multiple indeces for the next target stars and calculates the dVs of each trajectory from the same starting star.

Parameters

- **TL** (*TargetList*) – TargetList class object
- **old_sInd** (*int*) – Index of the current star
- **sInds** (*ndarray(int)*) – Integer index of the next star(s) of interest
- **slewTimes** (*Time(ndarray)*) – Slew times.
- **tmpCurrentTimeAbs** (*Time*) – Current absolute mission time in MJD

Returns
State vectors in rotating frame in normalized units (nx6)

Return type
ndarray(float)

calculate_observableTimes(*TL*, *sInds*, *currentTime*, *koMaps*, *koTimes*, *mode*)

Returns the next window of time during which targets are observable

This method returns a nx2 ndarray of times for every star given in the target list. The two entries for every star are the next times (after current time) when the star exits and enters keepout (i.e. the start and end times of the next window of observability).

Parameters

- **TL** (*TargetList*) – TargetList class object
- **sInds** (*numpy.ndarray(int)*) – Integer indices of the stars of interest
- **currentTime** (*Time*) – Current absolute mission time in MJD
- **koMaps** (*dict*) – Keepout values for n stars throughout time range of length m, key names being the system names specified in mode. True is a target unobstructed and observable, and False is a target unobservable due to obstructions in the keepout zone.
- **koTimes** (*Time*) – Absolute MJD mission times from start to end in steps of 1 d
- **mode** (*dict*) – Selected observing mode

Returns
Start and end times of next observability time window in absolute time MJD. n is length of sInds

Return type
Time

calculate_slewTimes(*TL*, *old_sInd*, *sInds*, *sd*, *obsTimes*, *currentTime*)

Finds slew times and separation angles between target stars

This method determines the slew times of an occulter spacecraft needed to transfer from one star's line of sight to all others in a given target list.

Parameters

- **TL** (*TargetList*) – TargetList class object
- **old_sInd** (*int*) – Integer index of the most recently observed star
- **sInds** (*ndarray(int)*) – Integer indices of the star of interest
- **sd** (*Quantity*) – Angular separation between stars in rad
- **obsTimes** (*Time(ndarray)*) – Observation times for targets.
- **currentTime** (*astropy Time*) – Current absolute mission time in MJD

Returns
Time to transfer to new star line of sight in units of days

Return type
Quantity

cent(*currentTime*)

Finds time in Julian centuries since J2000 epoch

This quantity is needed for many algorithms from Vallado 2013.

Parameters
currentTime (*Time*) – Current absolute mission time in MJD

Returns
time in Julian centuries since the J2000 epoch

Return type
ndarray(float)

distForces(*TL*, *sInd*, *currentTime*)

Finds lateral and axial disturbance forces on an occulter

Parameters

- **TL** (*TargetList*) – TargetList class object
- **sInd** (*int*) – Integer index of the star of interest
- **currentTime** (*Time*) – Current absolute mission time in MJD

Returns

Quantity:
dF_lateral: Lateral disturbance force in units of N

Quantity:
dF_axial: Axial disturbance force in units of N

Return type
tuple

eclip2equat(*r_eclip*, *currentTime*)

Rotates heliocentric coordinates from ecliptic to equatorial frame.

Parameters

- **r_eclip** (*Quantity(ndarray(float))*) – Positions vector in heliocentric ecliptic frame in units of AU
- **currentTime** (*astropy.time.Time*) – Current absolute mission time in MJD

Returns
Positions vector in heliocentric equatorial frame in units of AU. nx3

Return type
Quantity(*ndarray*(float))

equat2eclip(*r_equat*, *currentTime*, *rotsign=1*)

Rotates heliocentric coordinates from equatorial to ecliptic frame.

Parameters

- **r_equat** (*Quantity(ndarray(float))*) – Positions vector in heliocentric equatorial frame in units of AU. nx3
- **currentTime** (*Time*) – Current absolute mission time in MJD
- **rotsign** (*int*) – Optional flag, default 1, set -1 to reverse the rotation

Returns
Positions vector in heliocentric ecliptic frame in units of AU. nx3

Return type
Quantity(*ndarray*(float))

find_nextObsWindow(*TL*, *sInds*, *currentTimes*, *koMap*, *koTimes*)

Method used by calculate_observableTimes for finding next observable windows

This method returns a nx2 ndarray of times for every star given in the target list. The two entries for every star are the next times (after current time) when the star exits and enters keepout (i.e. the start and end times of the next window of observability).

Parameters

- **TL** (*TargetList*) – TargetList class object
- **sInds** (*ndarray(int)*) – Integer indices of the stars of interest
- **currentTimes** (*Time*) – Current absolute mission time in MJD same length as sInds
- **koMap** (*ndarray(int)*) – Keepout values for n stars throughout time range of length m (mxn)
- **koTimes** (*astropy.time.Time*) – Absolute MJD mission times from start to end in steps of 1 d
- **mode** (*dict*) – Selected observing mode

Returns
Start and end times of next observability time window in MJD

Return type
astropy.time.Time(*ndarray*)

generate_koMap(*TL*, *missionStart*, *missionFinishAbs*, *koangles*)

Creates keepout map for all targets throughout mission lifetime.

This method returns a binary map showing when all stars in the given target list are in or out of the keepout zone (i.e. when they are not observable) from mission start to mission finish.

Parameters

- **TL** (*TargetList*) – TargetList class object
- **missionStart** (*Time*) – Absolute start of mission time in MJD
- **missionFinishAbs** (*Time*) – Absolute end of mission time in MJD

- **koangles** (*Quantity(ndarray(float))*) – s x 4 x 2 array where s is the number of starlight suppression systems as defined in the Optical System. Each of the remaining 4 x 2 arrays are system specific koAngles for the Sun, Moon, Earth, and small bodies (4), each with a minimum and maximum value (2) in units of deg.

Returns

koMap (~numpy.ndarray(bool)):
True is a target unobstructed and observable, and False is a target unobservable due to obstructions in the keepout zone.

koTimes (~astropy.time.Time):
Absolute MJD mission times from start to end in steps of 1 d

Return type
tuple

keepout(*TL*, *sInds*, *currentTime*, *koangles*, *returnExtra=False*)

Finds keepout Boolean values for stars of interest.

This method returns the keepout Boolean values for stars of interest, where True is an observable star.

Parameters

- **TL** (*TargetList*) – TargetList class object
- **sInds** (*ndarray(int)*) – Integer indices of the stars of interest
- **currentTime** (*Time*) – Current absolute mission time in MJD MAY ONLY BE ONE VALUE OR DUPLICATES OF THE SAME VALUE
- **koangles** (*Quantity(ndarray(float))*) – s x 4 x 2 array where s is the number of starlight suppression systems as defined in the Optical System. Each of the remaining 4 x 2 arrays are system specific koAngles for the Sun, Moon, Earth, and small bodies (4), each with a minimum and maximum value (2) in units of deg.
- **returnExtra** (*bool*) – Optional flag, default False, set True to return additional information.

Returns

kogood (~numpy.ndarray(bool)):
kogood s x n x m array of boolean values. True is a target unobstructed and observable, and False is a target unobservable due to obstructions in the keepout zone.

r_body (~astropy.units.Quantity(~numpy.ndarray(float))):
Only returned if returnExtra is True 11 x m x 3 array where m is len(currentTime) of heliocentric equatorial Cartesian elements of the Sun, Moon, Earth and Mercury->Pluto

r_targ (~astropy.units.Quantity(~numpy.ndarray(float))):
Only returned if returnExtra is True m x n x 3 array where m is len(currentTime) or 1 if staticStars is true in TargetList of heliocentric equatorial Cartesian coords of target and n is the len(sInds)

culprit (numpy.ndarray(int)):
Only returned if returnExtra is True s x n x m x 12 array of boolean integer values identifying which body is responsible for keepout (when equal to 1). m is number of targets and n is len(currentTime). Last dimension is ordered same as r_body, with an extra line for solar panels being the culprit

koangleArray (~astropy.units.Quantity(~numpy.ndarray(float))):
Only returned if returnExtra is True s x 11 x 2 element array of minimum and maximum keepouts used for each body. Same ordering as r_body.

Return type
tuple or *ndarray*(bool)

keplerplanet(*currentTime*, *bodyname*, *eclip=False*)

Finds solar system body positions vector in heliocentric equatorial (default) or ecliptic frame for current time (MJD).

This method uses algorithms 2 and 10 from Vallado 2013 to find heliocentric equatorial position vectors for solar system objects.

Parameters

- **currentTime** (*Time*) – Current absolute mission time in MJD
- **bodyname** (*str*) – Solar system object name
- **eclip** (*bool*) – Boolean used to switch to heliocentric ecliptic frame. Defaults to False, corresponding to heliocentric equatorial frame.

Returns
Solar system body positions in heliocentric equatorial (default) or ecliptic frame in units of AU

Return type
Quantity(*ndarray*(float))

Note: Use eclip=True to get ecliptic coordinates.

---

log_occulterResults(*DRM*, *slewTimes*, *sInd*, *sd*, *dV*)

Updates the given DRM to include occulter values and results

Parameters

- **DRM** (*dict*) – Design Reference Mission, contains the results of one complete observation (detection and characterization)
- **slewTimes** (*astropy.units.Quantity*) – Time to transfer to new star line of sight in units of days
- **sInd** (*int*) – Integer index of the star of interest
- **sd** (*astropy.units.Quantity*) – Angular separation between stars in rad
- **dV** (*astropy.units.Quantity*) – Delta-V used to transfer to new star line of sight in units of m/s

Returns
Design Reference Mission dictionary, contains the results of one complete observation (detection and characterization)

Return type
dict

mass_dec(*dF_lateral*, *t_int*)

Returns mass_used and deltaV

The values returned by this method are used to decrement spacecraft mass for station-keeping.

Parameters

- **dF_lateral** (*astropy.units.Quantity*) – Lateral disturbance force in units of N
- **t_int** (*astropy.units.Quantity*) – Integration time in units of day

Returns

intMdot (astropy.units.Quantity):
Mass flow rate in units of kg/s

mass_used (astropy.units.Quantity):
Mass used in station-keeping units of kg

deltaV (astropy.units.Quantity):
Change in velocity required for station-keeping in units of km/s

Return type
tuple

`mass_dec_sk`(*TL*, *sInd*, *currentTime*, *t_int*)

Returns mass_used, deltaV and disturbance forces

This method calculates all values needed to decrement spacecraft mass for station-keeping.

Parameters

- **TL** (*TargetList*) – TargetList class object
- **sInd** (*int*) – Integer index of the star of interest
- **currentTime** (*astropy.time.Time*) – Current absolute mission time in MJD
- **t_int** (*astropy.units.Quantity*) – Integration time in units of day

Returns

dF_lateral (astropy.units.Quantity):
Lateral disturbance force in units of N

dF_axial (astropy.units.Quantity):
Axial disturbance force in units of N

intMdot (astropy.units.Quantity):
Mass flow rate in units of kg/s

mass_used (astropy.units.Quantity):
Mass used in station-keeping units of kg

deltaV (astropy.units.Quantity):
Change in velocity required for station-keeping in units of km/s

Return type
tuple

`moon_earth`(*currentTime*)

Finds geocentric equatorial positions vector for Earth's moon

This method uses Algorithm 31 from Vallado 2013 to find the geocentric equatorial positions vector for Earth's moon.

Parameters
currentTime (*Time*) – Current absolute mission time in MJD

Returns
Geocentric equatorial position vector in units of AU

Return type
Quantity(*ndarray*(float))

orbit(*currentTime*, *eclip=False*)

Finds observatory orbit positions vector in heliocentric equatorial (default) or ecliptic frame for current time (MJD).

This method returns the telescope geosynchronous circular orbit position vector.

Parameters

- **currentTime** (*Time*) – Current absolute mission time in MJD
- **eclip** (*bool*) – Boolean used to switch to heliocentric ecliptic frame. Defaults to False, corresponding to heliocentric equatorial frame.

Returns
Observatory orbit positions vector in heliocentric equatorial (default) or ecliptic frame in units of AU. nx3

Return type
Quantity(*ndarray*(float))

Note: Use eclip=True to get ecliptic coordinates.

---

propeph(*x*, *TDB*)

Propagates an ephemeris from Vallado 2013 to current time.

Parameters

- **x** (*list*) – ephemeride list (maximum of 4 elements)
- **TDB** (*float*) – time in Julian centuries since the J2000 epoch

Returns
ephemerides value at current time

Return type
numpy.darray(float)

refuel_tank(*TK*, *tank=None*)

Attempt to refuel a fuel tank and report status

Parameters

- **TK** (*TimeKeeping*) – TimeKeeping object. Not used in prototype but an input for any implementations that wish to do time-aware operations.
- **tank** (*str, optional*) – Either 'sk' or 'slew' when twotanks is True. Otherwise, None. Defaults None

Returns
True represents successful refeuling. False means refueling is not possible for selected tank.

Return type
bool

rot(*th*, *axis*)

Finds the rotation matrix of angle th about the axis value

Parameters

- **th** (*float*) – Rotation angle in radians
- **axis** (*int*) – Integer value denoting rotation axis (1,2, or 3)

Returns
Rotation matrix

Return type
ndarray(float)

solarSystem_body_position(*currentTime*, *bodyname*, *eclip=False*)

Finds solar system body positions vector in heliocentric equatorial (default) or ecliptic frame for current time (MJD).

This passes all arguments to one of spk_body or keplerplanet, depending on the value of self.havejplephem.

Parameters

- **currentTime** (*Time*) – Current absolute mission time in MJD
- **bodyname** (*str*) – Solar system object name
- **eclip** (*bool*) – Boolean used to switch to heliocentric ecliptic frame. Defaults to False, corresponding to heliocentric equatorial frame.

Returns
Solar system body positions in heliocentric equatorial (default) or ecliptic frame in units of AU. nx3

Return type
Quantity(*ndarray*(float))

Note: Use eclip=True to get ecliptic coordinates.

---

spk_body(*currentTime*, *bodyname*, *eclip=False*)

Finds solar system body positions vector in heliocentric equatorial (default) or ecliptic frame for current time (MJD).

This method uses spice kernel from NAIF to find heliocentric equatorial position vectors for solar system objects.

Parameters

- **currentTime** (*Time*) – Current absolute mission time in MJD
- **bodyname** (*str*) – Solar system object name
- **eclip** (*bool*) – Boolean used to switch to heliocentric ecliptic frame. Defaults to False, corresponding to heliocentric equatorial frame.

Returns
Solar system body positions in heliocentric equatorial (default) or ecliptic frame in units of AU. nx3

Return type
Quantity(*ndarray*(float))

Note: Use eclip=True to get ecliptic coordinates.

star_angularSep(*TL*, *old_sInd*, *sInds*, *currentTime*)

Finds angular separation from old star to given list of stars

This method returns the angular separation from the last observed star to all others on the given list at the currentTime.

Parameters

- **TL** (*TargetList*) – TargetList class object
- **old_sInd** (*int*) – Integer index of the last star of interest
- **sInds** (*ndarray(int)*) – Integer indices of the stars of interest
- **currentTime** (*Time*) – Current absolute mission time in MJD

Returns
Angular separation between two target stars

Return type
float

EXOSIMS.Prototypes.OpticalSystem module

class EXOSIMS.Prototypes.OpticalSystem.**OpticalSystem**(*obscurFac=0.1*, *shapeFac=0.7853981633974483*, *pupilDiam=4*, *intCutoff=50*, *scienceInstruments=[{'name': 'imager'}]*, *QE=0.9*, *optics=0.5*, *FoV=10*, *pixelNumber=1000*, *pixelSize=1e-05*, *pixelScale=0.02*, *sread=1e-06*, *idark=0.0001*, *texp=100*, *Rs=50*, *lenslSamp=2*, *starlightSuppressionSystems=[{'name': 'coronagraph'}]*, *lam=500*, *BW=0.2*, *occ_trans=0.2*, *core_thruput=0.1*, *core_contrast=1e-10*, *contrast_floor=None*, *core_platescale=None*, *core_platescale_units=None*, *input_angle_units='arcsec'*, *ohTime=1*, *observingModes=None*, *SNR=5*, *timeMultiplier=1.0*, *IWA=0.1*, *OWA=inf*, *stabilityFact=1*, *cachedir=None*, *koAngles_Sun=[0, 180]*, *koAngles_Earth=[0, 180]*, *koAngles_Moon=[0, 180]*, *koAngles_Small=[0, 180]*, *binaryleakfilepath=None*, *texp_flag=False*, *bandpass_model='box'*, *bandpass_step=0.1*, *use_core_thruput_for_ez=False*, *csv_angsep_colname='r_as'*, ***specs*)

Bases: object

OpticalSystem Prototype

Parameters

- **obscurFac** (*float*) – Obscuration factor (fraction of primary mirror area obscured by secondary and spiders). Defaults to 0.1. Must be between 0 and 1. See *pupilArea* attribute definition.
- **shapeFac** (*float*) – Shape Factor. Determines the ellipticity of the primary mirror. Defaults to np.pi/4 (circular aperture). See pupilArea attribute definition.
- **pupilDiam** (*float*) – Primary mirror major diameter (in meters). Defaults to 4.
- **intCutoff** (*float*) – Integration time cutoff (in days). Determines the maximum time that is allowed per integration, and is used to limit integration target $\Delta\mathrm{mag}$. Defaults to 50.

- **scienceInstruments** (*list(dict)*) – List of dicts defining all science instruments. Minimally must contain one science instrument. Each dictionary must minimally contain a `name` keyword, which must be unique to each instrument and must include the substring `imager` (for imaging devices) or `spectro` (for spectrometers). By default, this keyword is set to `[{'name': 'imager'}]`, creating a single imaging science instrument. Additional parameters are filled in with default values set by the keywords below. For more details on science instrument definitions see *OpticalSystem*.
- **QE** (*float*) – Default quantum efficiency. Only used when not set in science instrument definition. Defaults to 0.9
- **optics** (*float*) – Total attenuation due to science instrument optics. This is the net attenuation due to all optics in the science instrument path after the primary mirror, excluding any starlight suppression system (i.e., coronagraph) optics. Only used when not set in science instrument definition. Defaults to 0.5
- **FoV** (*float*) – Default instrument half-field of view (in arcseconds). Only used when not set in science instrument definition. Defaults to 10
- **pixelNumber** (*float*) – Default number of pixels across the detector. Only used when not set in science instrument definition. Defaults to 1000.
- **pixelSize** (*float*) – Default pixel pitch (nominal distance between adjacent pixel centers, in meters). Only used when not set in science instrument definition. Defaults to 1e-5
- **pixelScale** (*float*) – Default pixel scale (instantaneous field of view of each pixel, in arcseconds). Only used when not set in science instrument definition. Defaults to 0.02.
- **sread** (*float*) – Default read noise (in electrons/pixel/read). Only used when not set in science instrument definition. Defaults to 1e-6
- **idark** (*float*) – Default dark current (in electrons/pixel/s). Only used when not set in science instrument definition. Defaults to 1e-4
- **texp** (*float*) – Default single exposure time (in s). Only used when not set in science instrument definition. Defaults to 100
- **Rs** (*float*) – Default spectral resolving power. Only used when not set in science instrument definition. Only applies to spetrometers. Defaults to 50.
- **lenslSamp** (*float*) – Default lenslet sampling (number of pixels per lenslet rows or columns). Only used when not set in science instrument definition. Defaults to 2
- **starlightSuppressionSystems** (*list(dict)*) – List of dicts defining all starlight suppression systems. Minimally must contain one system. Each dictionary must minimally contain a `name` keyword, which must be unique to each system. By default, this keyword is set to `[{'name': 'coronagraph'}]`, creating a single coronagraphic starlight suppression system. Additional parameters are filled in with default values set by the keywords below. For more details on starlight suppression system definitions see *OpticalSystem*.
- **lam** (*float*) – Default central wavelength of starlight suppression system (in nm). Only used when not set in starlight suppression system definition. Defaults to 500
- **BW** (*float*) – Default fractional bandwidth. Only used when not set in starlight suppression system definition. Defaults to 0.2
- **occ_trans** (*float*) – Default coronagraphic transmission. Only used when not set in starlight suppression system definition. Defaults to 0.2
- **core_thruput** (*float*) – Default core throughput. Only used when not set in starlight suppression system definition. Defaults to 0.1

- **core_contrast** (*float*) – Default core contrast. Only used when not set in starlight suppression system definition. Defaults to 1e-10
- **contrast_floor** (*float, optional*) – Default contrast floor. Only used when not set in starlight suppression system definition. If not None, sets absolute contrast floor. Defaults to None
- **core_platescale** (*float, optional*) – Default core platescale. Only used when not set in starlight suppression system definition. Defaults to None. Units determiend by input_angle_units.
- **input_angle_units** (*str, optional*) – Default angle units of all starlightSuppressionSystems-related inputs (as applicable). This includes all CSV input tables or FITS input tables without a UNIT keyword in the header. Only used when not set in starlight suppression system definition. None, 'unitless' or 'LAMBDA/D' are all interepreted as
$lambda/D$ units. Otherwise must be a string that is parsable as an astropy angle unit. Defaults to 'arcsec'.
- **ohTime** (*float*) – Default overhead time (in days). Only used when not set in starlight suppression system definition. Time is added to every observation (on top of observatory settling time). Defaults to 1
- **observingModes** (*list(dict), optional*) – List of dicts defining observing modes. These are essentially combinations of instruments and starlight suppression systems, identified by their names in keywords instName and systName, respectively. One mode must be identified as the default detection mode (by setting keyword detectionMode to True in the mode definition. If None (default) a single observing mode is generated combining the first instrument in scienceInstruments with the first starlight suppression system in starlightSuppressionSystems, and is marked as the detection mode. Additional parameters are filled in with default values set by the keywords below. For more details on mode definitions see *OpticalSystem*.
- **SNR** (*float*) – Default target signal to noise ratio. Only used when not set in observing mode definition. Defaults to 5
- **timeMultiplier** (*float*) – Default integration time multiplier. Only used when not set in observing mode definition. Every integration time calculated for an observing mode is scaled by this factor. For example, if an observing mode requires two rolls per observation (i.e., if it covers only 180 degrees of the field), then this quantity should be set to 2 for that mode. However, in some cases (i.e., spectroscopic followup) it may not be necessary to integrate on the full field, in which case this quantity could be set to 1. Defaults to 1
- **IWA** (*float*) – Default *IWA* (in input_angle_units). Only used when not set in starlight suppression system definition. Defaults to 0.1
- **OWA** (*float*) – Default *OWA* (in input_angle_units). Only used when not set in starlight suppression system definition. Defaults to numpy.Inf
- **stabilityFact** (*float*) – Stability factor. Defaults to 1
- **cachedir** (*str, optional*) – Full path to cachedir. If None (default) use default (see *Cache Directory*)
- **koAngles_Sun** (*list(float)*) – Default [Min, Max] keepout angles for Sun. Only used when not set in starlight suppression system definition. Defaults to [0,180]
- **koAngles_Earth** (*list(float)*) – Default [Min, Max] keepout angles for Earth. Only used when not set in starlight suppression system definition. Defaults to [0,180]

- **koAngles_Moon** (*list(float)*) – Default [Min, Max] keepout angles for the moon. Only used when not set in starlight suppression system definition. Defaults to [0,180]
- **koAngles_Small** (*list(float)*) – Default [Min, Max] keepout angles for all other bodies. Only used when not set in starlight suppression system definition. Defaults to [0,180],
- **binaryleakfilepath** (*str, optional*) – If set, full path to binary leak definition file. Defaults to None
- **texp_flag** (*bool*) – Toggle use of planet shot noise value for frame exposure time (overrides instrument texp value). Defaults to False.
- **bandpass_model** (*str*) – Default model to use for mode bandpasses. Must be one of 'gaussian' or 'box' (case insensitive). Only used if not set in mode definition. Defaults to box.
- **bandpass_step** (*float*) – Default step size (in nm) to use when generating Box-model bandpasses. Only used if not set in mode definition. Defaults to 0.1.
- **use_core_thruput_for_ez** (*bool*) – If True, compute exozodi contribution using core_thruput. If False (default) use occ_trans
- **csv_angsep_colname** (*str*) – Default column name to use for the angular separation column for CSV data. Only used when not set in starlight suppression system definition. Defaults to r_as (matching the default input_angle_units). These two inputs should be updated together.
- ****specs** – *Input Specification*

_outspec

Output Specification

Type
dict

allowed_observingMode_kws

List of allowed keywords in observingMode dictionaries

Type
list

allowed_scienceInstrument_kws

List of allowed keywords in scienceInstrument dictionaries

Type
list

allowed_starlightSuppressionSystem_kws

List of allowed keywords in starlightSuppressionSystem dictionaries

Type
list

cachedir

Path to the EXOSIMS cache directory (see *Cache Directory*)

Type
str

default_vals

All inputs not assigned to object attributes are considered to be default values to be used for filling in information in the optical system definition, and are copied into this dictionary for storage.

Type
dict

`haveOcculter`

One or more starlight suppresion systems are starshade-based

Type
bool

`intCutoff`

Maximum allowable continuous integration time. Time units.

Type
astropy.units.quantity.Quantity

`IWA`

Minimum inner working angle.

Type
astropy.units.quantity.Quantity

`obscurFac`

Obscuration factor (fraction of primary mirror area obscured by secondary and spiders).

Type
float

`observingModes`

List of dicts defining observing modes. These are essentially combinations of instruments and starlight suppression systems, identified by their names in keywords `instName` and `systName`, respectively. One mode must be identified as the default detection mode (by setting keyword `detectionMode` to True in the mode definition. If None (default) a single observing mode is generated combining the first instrument in `scienceInstruments` with the first starlight suppression system in `starlightSuppressionSystems`, and is marked as the detection mode. Additional parameters are filled in with default values set by the keywords below. For more details on mode definitions see *OpticalSystem*.

Type
list

`OWA`

Maximum outer working angle.

Type
astropy.units.quantity.Quantity

`pupilArea`

Total effective pupil area:

$$A = (1 - F_o)F_s D^2$$

where F_o is the obscuration factor, F_s is the shape factor, and D is the pupil diameter.

Type
astropy.units.quantity.Quantity

`pupilDiam`

Pupil major diameter. Length units.

Type
astropy.units.quantity.Quantity

scienceInstruments

List of dicts defining all science instruments. Minimally must contain one science instrument. Each dictionary must minimally contain a `name` keyword, which must be unique to each instrument and must include the substring `imager` (for imaging devices) or `spectro` (for spectrometers). By default, this keyword is set to `[{'name': 'imager'}]`, creating a single imaging science instrument. Additional parameters are filled in with default values set by the keywords below. For more details on science instrument definitions see *OpticalSystem*.

Type
list

shapeFac

Primary mirror shape factor.

Type
float

stabilityFact

Telescope stability factor.

Type
float

starlightSuppressionSystems

List of dicts defining all starlight suppression systems. Minimally must contain one system. Each dictionary must minimally contain a `name` keyword, which must be unique to each system. By default, this keyword is set to `[{'name': 'coronagraph'}]`, creating a single coronagraphic starlight suppression system. Additional parameters are filled in with default values set by the keywords below. For more details on starlight suppression system definitions see *OpticalSystem*.

Type
list

texp_flag

Toggle use of planet shot noise value for frame exposure time (overriides instrument texp value).

Type
bool

use_core_thruput_for_ez

Toggle use of core_thruput (instead of occ_trans) in computing exozodi flux.

Type
bool

Cp_Cb_Csp(*TL*, *sInds*, *fZ*, *fEZ*, *dMag*, *WA*, *mode*, *returnExtra=False*, *TK=None*)

Calculates electron count rates for planet signal, background noise, and speckle residuals.

Parameters

- **TL** (*TargetList*) – TargetList class object
- **sInds** (*ndarray(int)*) – Integer indices of the stars of interest
- **fZ** (*Quantity(ndarray(float))*) – Surface brightness of local zodiacal light in units of 1/arcsec2
- **fEZ** (*Quantity(ndarray(float))*) – Surface brightness of exo-zodiacal light in units of 1/arcsec2
- **dMag** (*ndarray(float)*) – Differences in magnitude between planets and their host star

- **WA** (*Quantity(ndarray(float))*) – Working angles of the planets of interest in units of arcsec
- **mode** (*dict*) – Selected observing mode
- **returnExtra** (*bool*) – Optional flag, default False, set True to return additional rates for validation
- **TK** (*TimeKeeping*, optional) – Optional TimeKeeping object (default None), used to model detector degradation effects where applicable.

Returns

C_p (~astropy.units.Quantity(~numpy.ndarray(float))):
Planet signal electron count rate in units of 1/s

C_b (~astropy.units.Quantity(~numpy.ndarray(float))):
Background noise electron count rate in units of 1/s

C_sp (~astropy.units.Quantity(~numpy.ndarray(float))):
Residual speckle spatial structure (systematic error) in units of 1/s

Return type
tuple

Cp_Cb_Csp_helper(*TL*, *sInds*, *fZ*, *fEZ*, *dMag*, *WA*, *mode*)

Helper method for Cp_Cb_Csp that performs lots of common computations :param TL: TargetList class object :type TL: *TargetList* :param sInds: Integer indices of the stars of interest :type sInds: ~numpy.ndarray(int) :param fZ: Surface brightness of local zodiacal light in units of 1/arcsec2 :type fZ: ~astropy.units.Quantity(~numpy.ndarray(float)) :param fEZ: Surface brightness of exo-zodiacal light in units of 1/arcsec2 :type fEZ: ~astropy.units.Quantity(~numpy.ndarray(float)) :param dMag: Differences in magnitude between planets and their host star :type dMag: ~numpy.ndarray(float) :param WA: Working angles of the planets of interest in units of arcsec :type WA: ~astropy.units.Quantity(~numpy.ndarray(float)) :param mode: Selected observing mode :type mode: dict

Returns

C_star (~astropy.units.Quantity(~numpy.ndarray(float))):
Non-coronagraphic star count rate (1/s)

C_p0 (~astropy.units.Quantity(~numpy.ndarray(float))):
Planet count rate (1/s)

C_sr (~astropy.units.Quantity(~numpy.ndarray(float))):
Starlight residual count rate (1/s)

C_z (~astropy.units.Quantity(~numpy.ndarray(float))):
Local zodi count rate (1/s)

C_ez (~astropy.units.Quantity(~numpy.ndarray(float))):
Exozodi count rate (1/s)

C_dc (~astropy.units.Quantity(~numpy.ndarray(float))):
Dark current count rate (1/s)

C_bl (~astropy.units.Quantity(~numpy.ndarray(float))):
Background leak count rate (1/s)'

Npix (float):
Number of pixels in photometric aperture

Return type
tuple

calc_dMag_per_intTime(*intTimes*, *TL*, *sInds*, *fZ*, *fEZ*, *WA*, *mode*, *C_b=None*, *C_sp=None*, *TK=None*)

Finds achievable planet delta magnitude for one integration time per star in the input list at one working angle.

Parameters

- **intTimes** (*Quantity(ndarray(float))*) – Integration times in units of day
- **TL** (*TargetList*) – TargetList class object
- **sInds** (*numpy.ndarray(int)*) – Integer indices of the stars of interest
- **fZ** (*Quantity(ndarray(float))*) – Surface brightness of local zodiacal light in units of 1/arcsec2
- **fEZ** (*Quantity(ndarray(float))*) – Surface brightness of exo-zodiacal light in units of 1/arcsec2
- **WA** (*Quantity(ndarray(float))*) – Working angles of the planets of interest in units of arcsec
- **mode** (*dict*) – Selected observing mode
- **C_b** (*Quantity(ndarray(float))*) – Background noise electron count rate in units of 1/s (optional)
- **C_sp** (*Quantity(ndarray(float))*) – Residual speckle spatial structure (systematic error) in units of 1/s (optional)
- **TK** (*TimeKeeping*, optional) – Optional TimeKeeping object (default None), used to model detector degradation effects where applicable.

Returns

Achievable dMag for given integration time and working angle

Return type

numpy.ndarray(float)

Warning: The prototype implementation assumes the exact same integration time model as the other prototype methods (specifically Cp_Cb_Csp and calc_intTime). If either of these is overloaded, and, in particular, if C_b and/or C_sp are not modeled as independent of C_p, then the analytical approach used here will *not* work and must be replaced with numerical inversion.

calc_intTime(*TL*, *sInds*, *fZ*, *fEZ*, *dMag*, *WA*, *mode*, *TK=None*)

Finds integration time to reach a given dMag at a particular WA with given local and exozodi values for specific targets and for a specific observing mode.

Parameters

- **TL** (*TargetList*) – TargetList class object
- **sInds** (*numpy.ndarray(int)*) – Integer indices of the stars of interest
- **fZ** (*Quantity(ndarray(float))*) – Surface brightness of local zodiacal light in units of 1/arcsec2
- **fEZ** (*Quantity(ndarray(float))*) – Surface brightness of exo-zodiacal light in units of 1/arcsec2
- **dMag** (*numpy.ndarray(int)numpy.ndarray(float)*) – Differences in magnitude between planets and their host star

- **WA** (*Quantity(ndarray(float))*) – Working angles of the planets of interest in units of arcsec
- **mode** (*dict*) – Selected observing mode
- **TK** (*TimeKeeping*, optional) – Optional TimeKeeping object (default None), used to model detector degradation effects where applicable.

Returns
Integration times

Return type
Quantity(*ndarray*(float))

Note: All infeasible integration times are returned as NaN values

calc_saturation_dMag(*TL*, *sInds*, *fZ*, *fEZ*, *WA*, *mode*, *TK=None*)

This calculates the delta magnitude for each target star that corresponds to an infinite integration time.

Parameters

- **TL** (*TargetList*) – TargetList class object
- **sInds** (*numpy.ndarray(int)*) – Integer indices of the stars of interest
- **fZ** (*Quantity(ndarray(float))*) – Surface brightness of local zodiacal light in units of 1/arcsec2
- **fEZ** (*Quantity(ndarray(float))*) – Surface brightness of exo-zodiacal light in units of 1/arcsec2
- **WA** (*Quantity(ndarray(float))*) – Working angles of the planets of interest in units of arcsec
- **mode** (*dict*) – Selected observing mode
- **TK** (*TimeKeeping*, optional) – Optional TimeKeeping object (default None), used to model detector degradation effects where applicable.

Returns
Saturation (maximum achievable) dMag for each target star

Return type
ndarray(float)

ddMag_dt(*intTimes*, *TL*, *sInds*, *fZ*, *fEZ*, *WA*, *mode*, *C_b=None*, *C_sp=None*, *TK=None*)

Finds derivative of achievable dMag with respect to integration time.

Parameters

- **intTimes** (*Quantity(ndarray(float))*) – Integration times in units of day
- **TL** (*TargetList*) – TargetList class object
- **sInds** (*numpy.ndarray(int)*) – Integer indices of the stars of interest
- **fZ** (*Quantity(ndarray(float))*) – Surface brightness of local zodiacal light in units of 1/arcsec2
- **fEZ** (*Quantity(ndarray(float))*) – Surface brightness of exo-zodiacal light in units of 1/arcsec2

- **WA** (*Quantity(ndarray(float))*) – Working angles of the planets of interest in units of arcsec
- **mode** (*dict*) – Selected observing mode
- **C_b** (*Quantity(ndarray(float))*) – Background noise electron count rate in units of 1/s (optional)
- **C_sp** (*Quantity(ndarray(float))*) – Residual speckle spatial structure (systematic error) in units of 1/s (optional)
- **TK** (*TimeKeeping*, optional) – Optional TimeKeeping object (default None), used to model detector degradation effects where applicable.

Returns
Derivative of achievable dMag with respect to integration time in units of 1/s

Return type
Quantity(*ndarray*(float))

genObsModeHex()

Generate a unique hash for every observing mode to be used in downstream identification and caching.

The hash will be based on the _outspec entries for the obsmode, its science instrument and its starlight suppression system.

get_angle_unit_from_header(*hdr*, *syst*)

Helper method. Extract angle unit from header, if it exists.

Parameters

- **hdr** (*astropy.io.fits.header.Header or list*) – FITS header for data or header row from CSV
- **syst** (*dict*) – Dictionary containing the parameters of one starlight suppression system

Returns
The angle unit.

Return type
astropy.units.Unit

get_core_mean_intensity(*syst*)

Load and process core_mean_intensity data

Parameters
syst (*dict*) – Dictionary containing the parameters of one starlight suppression system

Returns
Updated dictionary of starlight suppression system parameters

Return type
dict

get_coro_param(*syst*, *param_name*, *fill=0.0*, *expected_ndim=None*, *expected_first_dim=None*, *min_val=None*, *max_val=None*)

For a given starlightSuppressionSystem, this method loads an input parameter from a table (fits or csv file) or a scalar value. It then creates a callable lambda function, which depends on the wavelength of the system and the angular separation of the observed planet.

Parameters

- **syst** (*dict*) – Dictionary containing the parameters of one starlight suppression system

- **param_name** (*str*) – Name of the parameter that must be loaded
- **fill** (*float*) – Fill value for working angles outside of the input array definition
- **expected_ndim** (*int, optional*) – Expected number of dimensions. Only checked if not None. Defaults None.
- **expected_first_dim** (*int, optional*) – Expected size of first dimension of data. Only checked if not None. Defaults None
- **min_val** (*float, optional*) – Minimum allowed value of parameter. Defaults to None (no check).
- **max_val** (*float, optional*) – Maximum allowed value of paramter. Defaults to None (no check).

Returns
Updated dictionary of starlight suppression system parameters

Return type
dict

Note: The created lambda function handles the specified wavelength by rescaling the specified working angle by a factor syst['lam']/mode['lam']

Note: If the input parameter is taken from a table, the IWA and OWA of that system are constrained by the limits of the allowed WA on that table.

---

get_param_data(*ipth*, *left_col_name=None*, *param_name=None*, *expected_ndim=None*, *expected_first_dim=None*)

Gets the data from a file, used primarily to create interpolants for coronagraph parameters

Parameters

- **ipth** (*str*) – String to file location, will also work with any other path object
- **left_col_name** (*str,optional*) – For CSV files only. String representing the column containing the independent parameter to be extracted. This is for use in the case where the CSV file contains multiple columns and only two need to be returned. Defaults None.
- **param_name** (*str, optional*) – For CSV files only. String representing the column containing the dependent parameter to be extracted. This is for use in the case where the CSV file contains multiple columns and only two need to be returned. Defaults None.
- **expected_ndim** (*int, optional*) – Expected number of dimensions. Only checked if not None. Defaults None.
- **expected_first_dim** (*int, optional*) – Expected size of first dimension of data. Only checked if not None. Defaults None

Returns

dat (~numpy.ndarray):
Data array

hdr (list or astropy.io.fits.header.Header):
Data header. For CVS files this is a list of column header strings.

Return type
tuple

Note: CSV files *must* have a single header row

populate_observingModes(*observingModes*)

Helper method to parse input observingMode dictionaries and assign default values, as needed. Also creates the allowed_observingMode_kws attribute.

Parameters
observingModes (*list*) – List of observingMode dicts.

populate_observingModes_extra()

Additional setup for observing modes This is intended for overloading in downstream implementations and is intentionally left blank in the prototype.

populate_scienceInstruments(*scienceInstruments*)

Helper method to parse input scienceInstrument dictionaries and assign default values, as needed. Also creates the allowed_scienceInstrument_kws attribute.

Parameters
scienceInstruments (*list*) – List of scienceInstrument dicts.

populate_scienceInstruments_extra()

Additional setup for science instruments. This is intended for overloading in downstream implementations and is intentionally left blank in the prototype.

populate_starlightSuppressionSystems(*starlightSuppressionSystems*)

Helper method to parse input starlightSuppressionSystem dictionaries and assign default values, as needed. Also creates the allowed_starlightSuppressionSystem_kws attribute.

Parameters
starlightSuppressionSystems (*list*) – List of starlightSuppressionSystem dicts.

populate_starlightSuppressionSystems_extra()

Additional setup for starlight suppression systems. This is intended for overloading in downstream implementations and is intentionally left blank in the prototype.

update_syst_WAs(*syst*, *WA0*, *param_name*)

Helper method. Check system IWA/OWA and update from table data, as needed. Alternatively, set from defaults.

Parameters

- **syst** (*dict*) – Dictionary containing the parameters of one starlight suppression system
- **WA0** (*ndarray, optional*) – Array of angles from table data. Must be in arcseconds. If None, then just set from defaults.
- **param_name** (*str, optional*) – Name of parameter the table data belongs to. Must be set if WA is set.

Returns
Updated dictionary of starlight suppression system parameters

Return type
dict

EXOSIMS.Prototypes.PlanetPhysicalModel module

class EXOSIMS.Prototypes.PlanetPhysicalModel.**PlanetPhysicalModel**(*cachedir=None*, *whichPlanetPhaseFunction='lambert'*, ***specs*)

Bases: `object`

PlanetPhysicalModel Prototype

Parameters

- **whichPlanetPhaseFunction** (*str*) – Name of phase function to use. See *PlanetPhysicalModel*.
- ****specs** – *Input Specification*

_outspec

Output Specification

Type
dict

cachedir

Path to the EXOSIMS cache directory (see *Cache Directory*)

Type
str

whichPlanetPhaseFunction

Name of phase function to use.

Type
str

calc_Phi(*beta*, *phiIndex=None*)

Calculate the phase function. Prototype method uses the Lambert phase function from Sobolev 1975.

Parameters
beta (*Quantity(ndarray(float))*) – Planet phase angles at which the phase function is to be calculated, in units of rad

Returns
Planet phase function

Return type
ndarray(float)

calc_Teff(*starL*, *d*, *p*)

Calcluates the effective planet temperature given the stellar luminosity, planet albedo and star-planet distance.

This calculation represents a basic balckbody power balance, and does not take into account the actual emmisivity of the planet, or any non-equilibrium effects or temperature variations over the surface.

Note: The input albedo is taken to be the bond albedo, as required by the equilibrium calculation. For an isotropic scatterer (Lambert phase function) the Bond albedo is 1.5 times the geometric albedo. However, the Bond albedo must be strictly defined between 0 and 1, and an albedo of 1 produces a zero effective temperature.

Parameters

- **starL** (*ndarray(float)*) – Stellar luminosities in units of solar luminosity.

- **d** (*Quantity(ndarray(float))*) – Star-planet distances
- **p** (*ndarray(float)*) – Planet albedos

Returns
Planet effective temperature in degrees

Return type
Quantity(*ndarray*(float))

calc_albedo_from_sma(*a*, *prange=[0.367, 0.367]*)

Helper function for calculating albedo given the semi-major axis. The prototype provides only a dummy function that always returns the same value of 0.367.

Parameters
a (*Quantity(ndarray(float))*) – Semi-major axis values

Returns
Albedo values

Return type
ndarray(float)

calc_beta(*Phi*)

Calculates the Phase angle based on the assumed planet phase function

Parameters
Phi (*float*) – Phase angle function value ranging from 0 to 1

Returns
Phase angle from 0 rad to pi rad

Return type
beta (float)

calc_mass_from_radius(*Rp*)

Helper function for calculating mass given the radius.

Parameters
Rp (*Quantity(ndarray(float))*) – Planet radius in units of Earth radius

Returns
Planet mass in units of Earth mass

Return type
Quantity(*ndarray*(float))

calc_radius_from_mass(*Mp*)

Helper function for calculating radius given the mass.

Prototype provides only a dummy function that assumes a density of water.

Parameters
Mp (*astropy.units.Quantity(numpy.ndarray(float))*) – Planet mass in units of Earth mass

Returns
Planet radius in units of Earth radius

Return type
Quantity(*ndarray*(float))

EXOSIMS.Prototypes.PlanetPopulation module

class EXOSIMS.Prototypes.PlanetPopulation.**PlanetPopulation**(*arange=[0.1, 100.0]*, *erange=[0.01, 0.99]*, *Irange=[0.0, 180.0]*, *Orange=[0.0, 360.0]*, *wrange=[0.0, 360.0]*, *prange=[0.1, 0.6]*, *Rprange=[1.0, 30.0]*, *Mprange=[1.0, 4131.0]*, *scaleOrbits=False*, *constrainOrbits=False*, *eta=0.1*, *cachedir=None*, ***specs*)

Bases: object

PlanetPopulation Prototype

Parameters

- **arange** (*list(float)*) – [Min, Max] semi-major axis (in AU). Defaults to [0.1,100.]
- **erange** (*list(float)*) – [Min, Max] eccentricity. Defaults to [0.01,0.99]
- **Irange** (*list(float)*) – [Min, Max] inclination (in degrees). Defaults to [0.,180.]
- **Orange** (*list(float)*) – [Min, Max] longitude of the ascending node (in degrees). Defaults to [0.,360.]
- **wrange** (*list(float)*) – [Min, Max] argument of periapsis. Defaults to [0.,360.]
- **prange** (*list(float)*) – [Min, Max] geometric albedo. Defaults to [0.1,0.6]
- **Rprange** (*list(float)*) – [Min, Max] planet radius (in Earth radii). Defaults to [1.,30.]
- **Mprange** (*list(float)*) – [Min, Max] planet mass (in Earth masses). Defaults to [1.,4131.]
- **scaleOrbits** (*bool*) – Scale orbits by $\sqrt{L}$ where L is the stellar luminosity. This has the effect of matching insolation distnaces and preserving the habitable zone of the population. Defaults to False.
- **constrainOrbits** (*bool*) – Do not allow orbits where orbital radius can exceed the arange limits. Defaults to False
- **eta** (*float*) – Overall occurrence rate of the population. The expected number of planets per target star. Must be strictly positive, but may be greater than 1 (if more than 1 planet is expected per star, on average). Defaults to 0.1.
- **cachedir** (*str, optional*) – Full path to cachedir. If None (default) use default (see *Cache Directory*)
- ****specs** – *Input Specification*

_outspec

Output Specification

Type
dict

arange

[Min, Max] semi-major axis

Type
astropy.units.quantity.Quantity

cachedir

Path to the EXOSIMS cache directory (see *Cache Directory*)

Type
str

constrainOrbits

Do not allow orbits where orbital radius can exceed the `arange` limits.

Type
bool

erange

[Min, Max] eccentricity.

Type
numpy.ndarray

eta

Overall occurrence rate of the population. The expected number of planets per target star. Must be strictly positive, but may be greater than 1 (if more than 1 planet is expected per star, on average).

Type
float

Irange

[Min, Max] inclination

Type
astropy.units.quantity.Quantity

Mprange

[Min, Max] planet mass

Type
astropy.units.quantity.Quantity

Orange

[Min, Max] longitude of the ascending node

Type
astropy.units.quantity.Quantity

pfromRp

Albedo is dependent on planetary radius

Type
bool

PlanetPhysicalModel

Planet physical model object

Type
PlanetPhysicalModel

prange

[Min, Max] geometric albedo.

Type
numpy.ndarray

Rprange

[Min, Max] planet radius

Type
astropy.units.quantity.Quantity

rrange

[Min, Max] orbital radius

Type
astropy.units.quantity.Quantity

scaleOrbits

Scale orbits by $\sqrt{L}$ where L is the stellar luminosity. This has the effect of matching insolation distnaces and preserving the habitable zone of the population.

Type
bool

wrange

[Min, Max] argument of periapsis.

Type
astropy.units.quantity.Quantity

checkranges(*var*, *name*)

Helper function provides asserts on all 2 element lists of ranges

Parameters

- **var** (*list*) – 2-element list
- **name** (*str*) – Variable name

Returns
Sorted input variable

Return type
list

Raises AssertionError on test fail.

dist_albedo(*p*)

Probability density function for albedo

The prototype provides a uniform distribution between the minimum and maximum values.

Parameters
p (*ndarray(float)*) – Albedo value(s)

Returns
Albedo probability density

Return type
ndarray(float)

dist_eccen(*e*)

Probability density function for eccentricity

The prototype provides a uniform distribution between the minimum and maximum values.

Parameters
e (*ndarray(float)*) – Eccentricity value(s)

Returns
Eccentricity probability density

Return type
ndarray(float)

dist_eccen_from_sma(*e*, *a*)

Probability density function for eccentricity constrained by semi-major axis, such that orbital radius always falls within the provided sma range.

The prototype provides a uniform distribution between the minimum and maximum allowable values.

Parameters

- **e** (*ndarray(float)*) – Eccentricity values
- **a** (*ndarray(float)*) – Semi-major axis value in AU. Not an astropy quantity.

Returns
Probability density of eccentricity constrained by semi-major axis

Return type
ndarray(float)

dist_mass(*Mp*)

Probability density function for planetary mass in Earth mass

The prototype provides an unbounded power law distribution. Note that this should really be a function of a density model and the radius distribution for all implementations that use it.

Parameters
Mp (*ndarray(float)*) – Planetary mass value(s) in Earth mass. Not an astropy quantity.

Returns
Planetary mass probability density

Return type
ndarray(float)

dist_radius(*Rp*)

Probability density function for planetary radius in Earth radius

The prototype provides a log-uniform distribution between the minimum and maximum values.

Parameters
Rp (*ndarray(float)*) – Planetary radius value(s) in Earth radius. Not an astropy quantity.

Returns
Planetary radius probability density

Return type
ndarray(float)

dist_sma(*a*)

Probability density function for semi-major axis in AU

The prototype provides a log-uniform distribution between the minimum and maximum values.

Parameters
a (*ndarray(float)*) – Semi-major axis value(s) in AU. Not an astropy quantity.

Returns
Semi-major axis probability density

Return type
ndarray(float)

gen_angles(*n*, *commonSystemPlane=False*, *commonSystemPlaneParams=None*)

Generate inclination, longitude of the ascending node, and argument of periapse in degrees

The prototype generates inclination as sinusoidally distributed and longitude of the ascending node and argument of periapse as uniformly distributed.

Parameters

- **n** (*int*) – Number of samples to generate
- **commonSystemPlane** (*bool*) – Generate delta inclinations from common orbital plane rather than fully independent inclinations and Omegas. Defaults False. If True, commonSystemPlaneParams must be supplied.
- **commonSystemPlaneParams** (*None or list*) –

  4 element list of [I mean, I standard deviation,
  O mean, O standard deviation]

  in units of degrees, describing the distribution of inclinations and Omegas relative to a common orbital plane. Ignored if commonSystemPlane is False.

Returns

I (~astropy.units.Quantity(~numpy.ndarray(float))):
Inclination in units of degrees OR deviation in inclination (deg)

O (~astropy.units.Quantity(~numpy.ndarray(float))):
Longitude of the ascending node (deg)

w (~astropy.units.Quantity(~numpy.ndarray(float))):
Argument of periapsis (deg)

Return type
tuple

gen_input_check(*n*)

Helper function checks that input is integer, casts to int, is >= 0

Parameters
n (*float*) – An integer to validate

Returns
The input integer as an integer

Return type
int

Raises AssertionError on test fail.

gen_mass(*n*)

Generate planetary mass values in units of Earth mass.

The prototype provides a log-uniform distribution between the minimum and maximum values.

Parameters
n (*int*) – Number of samples to generate

Returns
Planet mass values in units of Earth mass.

Return type
Quantity(*ndarray*(float))

`gen_plan_params`(*n*)

Generate semi-major axis (AU), eccentricity, geometric albedo, and planetary radius (earthRad)

The prototype generates semi-major axis and planetary radius with log-uniform distributions and eccentricity and geometric albedo with uniform distributions.

Parameters
n (*int*) – Number of samples to generate

Returns

a (~astropy.units.Quantity(~numpy.ndarray(float))):
Semi-major axis in units of AU

e (~numpy.ndarray(float)):
Eccentricity

p (~numpy.ndarray(float)):
Geometric albedo

Rp (~astropy.units.Quantity(~numpy.ndarray(float))):
Planetary radius in units of earthRad

Return type
tuple

EXOSIMS.Prototypes.PostProcessing module

`class EXOSIMS.Prototypes.PostProcessing.PostProcessing`(*FAP=3e-07*, *MDP=0.001*, *ppFact=1.0*, *ppFact_char=1.0*, *FAdMag0=15*, *cachedir=None*, ***specs*)

Bases: `object`

PostProcessing Prototype

Parameters

- **FAP** (*float*) – False Alarm Probability. See [Kasdin2006]. Defaults to 3e-7
- **MDP** (*float*) – Missed Detection Probability. See [Kasdin2006]. Defaults to 1e-3
- **ppFact** (*float or str*) – Post-processing contrast factor, between 0 and 1. Either a scalar float for constant gain, or a string with the full path to a FITS file containing a two-column array for separation-dependent gain, where the first column contains the angular separation in units of arcsec. Defaults to 1.0
- **ppFact_char** (*float or str*) – Same as ppFact, but for spectral characterization. Defaults to 1.0
- **FAdMag0** (*float or str*) – Minimum delta magnitude that can be obtained by a false alarm: either a scalar for constant dMag, or a string with the full path to a FITS file containing a two-column array for separation-dependent dMag, where the first column contains the angular separation in units of arcsec. Defaults to 15
- **cachedir** (*str, optional*) – Full path to cachedir. If None (default) use default (see *Cache Directory*)
- ****specs** – *Input Specification*

_outspec

Output Specification

Type
dict

BackgroundSources

BackgroundSources object

Type
BackgroundSources

cachedir

Path to the EXOSIMS cache directory (see *Cache Directory*)

Type
str

FAP

False Alarm Probability. See [Kasdin2006].

Type
float

MDP

Missed Detection Probability. See [Kasdin2006].

Type
float

ppFact

Post-processing contrast factor, between 0 and 1, parametrized by angular separation

Type
callable

ppFact_char

Same as ppFact, but for characterization

Type
callable

FAdMag0

Minimum delta magnitude that can be obtained by a false alarm parametrized by angular separation

Type
callable

det_occur(*SNR*, *mode*, *TL*, *sInd*, *intTime*)

Determines if a detection has occurred

Parameters

- **SNR** (*ndarray(float)*) – signal-to-noise ratio of the planets around the selected target
- **mode** (*dict*) – Selected observing mode
- **TL** (*TargetList*) – TargetList class object
- **sInd** (*int*) – Index of the star being observed
- **intTime** (*Quantity(float)*) – Selected star integration time for detection

Returns

FA (bool):
False alarm (false positive) boolean.

MD (ndarray (bool)):
Missed detection (false negative) boolean with the size of number of planets around the target.

Return type
tuple

Note: The prototype implemenation does not consider background sources in calculating false positives, however, the unused TargetList, integration time and star index inputs are part of the interface to allow another implementation to do this.

EXOSIMS.Prototypes.SimulatedUniverse module

class EXOSIMS.Prototypes.SimulatedUniverse.**SimulatedUniverse**(*fixedPlanPerStar=None*, *Min=None*, *cachedir=None*, *lucky_planets=False*, *commonSystemPlane=False*, *commonSystemPlaneParams=[0, 2.25, 0, 2.25]*, ***specs*)

Bases: object

SimulatedUniverse Prototype

Parameters

- **fixedPlanPerStar** (*int, optional*) – If set, every system will have the same number of planets. Defaults to None
- **Min** (*float, optional*) – Initial mean anomaly for all planets. If set, every planet has the same mean anomaly at mission start. Defaults to None
- **cachedir** (*str, optional*) – Full path to cachedir. If None (default) use default (see *Cache Directory*)
- **lucky_planets** (*bool*) – Used downstream in survey simulation. If True, planets are observed at optimal times. Defaults to False
- **commonSystemPlane** (*bool*) – Planet inclinations are sampled as normally distributed about a common system plane. Defaults to False
- **commonSystemPlaneParams** (*list(float)*) –

  [inclination mean, inclination standard deviation, Omega mean,
  Omega standard deviation] defining the normal distribution of

  inclinations and longitudes of the ascending node about a common system plane in units of degrees. Ignored if commonSystemPlane is False. Defaults to [0 2.25, 0, 2.25], where the standard deviation is approximately the standard deviation of solar system planet inclinations.

- ****specs** – *Input Specification*

_outspec

Output Specification

Type
dict

a

Planet semi-major axis (length units)

Type
astropy.units.quantity.Quantity

BackgroundSources

BackgroundSources object

Type
BackgroundSources

cachedir

Path to the EXOSIMS cache directory (see *Cache Directory*)

Type
str

commonSystemPlaneParams

2 element list of [mean, standard deviation] in units of degrees, describing the distribution of inclinations relative to a common orbital plane. Ignored if commonSystemPlane is False.

Type
list

commonSystemPlane

If False, planet inclinations are independently drawn for all planets, including those in the same target system. If True, inclinations will be drawn from a normal distribution defined by commonSystemPlaneParams and added to a single inclination value drawn for each system.

Type
bool

Completeness

Completeness object

Type
Completeness

d

Current orbital radius magnitude (length units)

Type
astropy.units.quantity.Quantity

dMag

Current planet $\Delta\mathrm{mag}$

Type
numpy.ndarray

e

Planet eccentricity

Type
numpy.ndarray

fEZ

Surface brightness of exozodiacal light in units of 1/arcsec2

Type
astropy.units.quantity.Quantity

fixedPlanPerStar

If set, every system has the same number of planets, given by this attribute

Type
int or None

I

Planet inclinations (angle units)

Type
astropy.units.quantity.Quantity

lucky_planets

If True, planets are observed at optimal times.

Type
bool

M0

Initial planet mean anomaly (at mission start time).

Type
astropy.units.quantity.Quantity

Min

Input constant initial mean anomaly. If none, initial mean anomaly is randomly distributed from a uniform distribution in [0, 360] degrees.

Type
float or None

Mp

Planet mass.

Type
astropy.units.quantity.Quantity

nPlans

Number of planets in all target systems.

Type
int

O

Planet longitude of the ascending node (angle units)

Type
astropy.units.quantity.Quantity

OpticalSystem

Optical System object

Type
OpticalSystem

p

Planet geometric albedo

Type
numpy.ndarray

phi

Current value of planet phase function.

Type
numpy.ndarray

phiIndex

Intended for use with input 'whichPlanetPhaseFunction'='realSolarSystemPhaseFunc' When None, the default is the phi_lambert function, otherwise it is Solar System Phase Functions

Type
numpy.ndarray

plan2star

Index of host star or each planet. Indexes attributes of TargetsList.

Type
numpy.ndarray

planet_atts

List of planet attributes

Type
list

PlanetPhysicalModel

Planet physical model object.

Type
PlanetPhysicalModel

PlanetPopulation

Planet population object.

Type
PlanetPopulation

PostProcessing

Postprocessing object.

Type
PostProcessing

r

Current planet orbital radius (3xnPlans). Length units.

Type
astropy.units.quantity.Quantity

Rp

Planet radius (length units).

Type
astropy.units.quantity.Quantity

s

Current planet projected separation. Length units.

Type
astropy.units.quantity.Quantity

sInds

Indices of stars with planets. Equivalent to unique entries of `plan2star`.

Type
numpy.ndarray

TargetList

Target list object.

Type
TargetList

v

Current orbital velocity vector (3xnPlans). Velocity units.

Type
astropy.units.quantity.Quantity

w

Planet argument of periapsis.

Type
astropy.units.quantity.Quantity

WA

Current planet angular separation (angle units)

Type
astropy.units.quantity.Quantity

ZodiacalLight

Zodiacal light object.

Type
ZodiacalLight

Note: When generating planets, *PlanetPopulation* attribute `eta` is treated as the rate parameter of a Poisson distribution. Each target's number of planets is a Poisson random variable sampled with $\lambda \equiv \eta$.

Warning: All attributes described as 'current' are updated only when planets are observed. As such, during mission simulations, these values for different planets correspond to different times (bookkept in the survey simulation object).

dump_system_params(*sInd=None*)

Create a dictionary of time-dependant planet properties for a specific target

Parameters
sInd (*int*) – Index of the target system of interest. Default value (None) will return an empty dictionary with the selected parameters and their units.

Returns
Dictionary of time-dependant planet properties

Return type
dict

`dump_systems()`

Create a dictionary of planetary properties for archiving use.

Parameters
None –

Returns
Dictionary of planetary properties

Return type
dict

`gen_M0()`

Set initial mean anomaly for each planet

`gen_physical_properties(**specs)`

Generates the planetary systems' physical properties.

Populates arrays of the orbital elements, albedos, masses and radii of all planets, and generates indices that map from planet to parent star.

Parameters
****specs** – *Input Specification*

Returns
None

`init_systems()`

Finds initial time-dependant parameters. Assigns each planet an initial position, velocity, planet-star distance, apparent separation, phase function, surface brightness of exo-zodiacal light, delta magnitude, and working angle.

This method makes use of the systems' physical properties (masses, distances) and their orbital elements (a, e, I, O, w, M0).

`load_systems(systems)`

Load a dictionary of planetary properties (nominally created by dump_systems)

Parameters

- **systems** (*dict*) – Dictionary of planetary properties corresponding to the output of dump_systems.
- **Returns** – None

Note: If keyword `systemInclination` is present in the dictionary, it is assumed that it was generated with `commonSystemPlane` set to True. Similarly, if keyword `starnEZ` is present, it is assumed that `ZodiacalLight.commonSystemfEZ` should be true.

Warning: This method assumes that the exact same targetlist is being used as in the object that generated the systems dictionary. If this assumption is violated unexpected results may occur.

propag_system(*sInd*, *dt*)

Propagates planet time-dependant parameters: position, velocity, planet-star distance, apparent separation, phase function, surface brightness of exo-zodiacal light, delta magnitude, and working angle.

This method uses the Kepler state transition matrix to propagate a planet's state (position and velocity vectors) forward in time using the Kepler state transition matrix.

Parameters

- **sInd** (*int*) – Index of the target system of interest
- **dt** (*Quantity(float)*) – Time increment in units of day, for planet position propagation

Returns
None

revise_planets_list(*pInds*)

Replaces Simulated Universe planet attributes with filtered values, and updates the number of planets.

Parameters
pInds (*ndarray(int)*) – Planet indices to keep

Returns
None

Warning: Throws AssertionError if all planets are removed

revise_stars_list(*sInds*)

Revises the TargetList with filtered values, and updates the planets list accordingly.

Parameters
sInds (*ndarray(int)*) – Star indices to keep

Returns
None

set_planet_phase(*beta=1.5707963267948966*)

Positions all planets at input star-planet-observer phase angle where possible. For systems where the input phase angle is not achieved, planets are positioned at quadrature (phase angle of 90 deg).

The position found here is not unique. The desired phase angle will be achieved at two points on the planet's orbit (for non-face on orbits).

Parameters
beta (*float*) – star-planet-observer phase angle in radians.

setup_system_planes()

Helper function that augments the system planes if commonSystemPlane is true

Parameters
None –

Returns
None

EXOSIMS.Prototypes.StarCatalog module

class EXOSIMS.Prototypes.StarCatalog.**StarCatalog**(*ntargs=1*, *cachedir=None*, *VmagFill=0.1*, ***specs*)

Bases: object

StarCatalog Prototype

Parameters

- **ntargs** (*int*) – Number of stars in catalog. Defaults to 1.
- **cachedir** (*str, optional*) – Full path to cachedir. If None (default) use default (see *Cache Directory*)
- **VmagFill** (*float*) – Fill value for V magnitudes. Defaults to 0.1. Must be set to non-zero value or TargetList will fail to build.
- ****specs** – *Input Specification*

catalog_atts

All star catalog attributes that were copied in

Type
list

ntargs

Number of stars

Type
int

Name

Star names

Type
ndarray(str)

Spec

Star spectral types

Type
ndarray(str)

Umag

U magnitude

Type
ndarray(float)

Bmag

B magnitude

Type
ndarray(float)

Vmag

V magnitude

Type
ndarray(float)

Rmag

R magnitude

Type
ndarray(float)

Imag

I magnitude

Type
ndarray(float)

Jmag

J magnitude

Type
ndarray(float)

Hmag

H magnitude

Type
ndarray(float)

Kmag

K magnitude

Type
ndarray(float)

BV

B-V Johnson magnitude

Type
ndarray(float)

MV

Absolute V magnitude

Type
ndarray(float)

BC

Bolometric correction

Type
ndarray(float)

L

Stellar luminosity in Solar luminosities

Type
ndarray(float)

Binary_Cut

Boolean where True is a binary star with companion closer than 10 arcsec

Type
ndarray(bool)

dist

Distance to star in units of pc

Type
Quantity(*ndarray*(float))

parx

Parallax in units of mas

Type
Quantity(*ndarray*(float))

coords

SkyCoord object (ICRS frame) containing right ascension, declination, and distance to star in units of deg, deg, and pc

Type
astropy.coordinates.SkyCoord

pmra

Proper motion in right ascension in units of mas/year

Type
Quantity(*ndarray*(float))

pmdec

Proper motion in declination in units of mas/year

Type
Quantity(*ndarray*(float))

rv

Radial velocity in units of km/s

Type
Quantity(*ndarray*(float))

cachedir

Path to cache directory

Type
str

Note: The prototype will generate empty arrays for all attributes of size ntargs. These can then be either filled in or overwritten by inheriting implementations.

EXOSIMS.Prototypes.SurveyEnsemble module

class EXOSIMS.Prototypes.SurveyEnsemble.**SurveyEnsemble**(*cachedir=None*, ***specs*)

Bases: object

SurveyEnsemble Prototype

Parameters

- **cachedir** (*str, optional*) – Full path to cachedir. If None (default) use default (see *Cache Directory*)

- ****specs** – *Input Specification*

_outspec

Output Specification

Type
dict

cachedir

Path to the EXOSIMS cache directory (see *Cache Directory*)

Type
str

Important: The prototype implementation provides no parallelization. See *SurveyEnsemble* for more info.

---

run_ensemble(*sim*, *nb_run_sim*, *run_one=None*, *genNewPlanets=True*, *rewindPlanets=True*, *kwargs={}*)

Execute simulation ensemble

Parameters

- **sim** (*EXOSIMS.MissionSim*) – MissionSim object
- **nb_run_sim** (*int*) – number of simulations to run
- **run_one** (*callable*) – method to call for each simulation
- **genNewPlanets** (*bool*) – Generate new planets each for simulation. Defaults True.
- **rewindPlanets** (*bool*) – Reset planets to initial mean anomaly for each simulation. Defaults True
- **kwargs** (*dict*) – Keyword arguments to pass onwards (not used in prototype)

Returns
List of dictionaries of mission results

Return type
list(dict)

run_one(*SS*, *genNewPlanets=True*, *rewindPlanets=True*)

Parameters

- **SS** (*SurveySimulation*) – SurveySimulation object
- **genNewPlanets** (*bool*) – Generate new planets each for simulation. Defaults True.
- **rewindPlanets** (*bool*) – Reset planets to initial mean anomaly for each simulation. Defaults True

Returns
Mission results

Return type
list(dict)

EXOSIMS.Prototypes.SurveySimulation module

class EXOSIMS.Prototypes.SurveySimulation.**SurveySimulation**(*scriptfile=None*, *ntFlux=1*, *nVisitsMax=5*, *charMargin=0.15*, *dt_max=1.0*, *record_counts_path=None*, *nokoMap=False*, *nofZ=False*, *cachedir=None*, *defaultAddExoplanetObsTime=True*, *find_known_RV=False*, *include_known_RV=None*, *make_debug_bird_plots=False*, *debug_plot_path=None*, ***specs*)

Bases: object

SurveySimulation Prototype

Parameters

- **scriptfile** (*str, optional*) – JSON script file. If not set, assumes that dictionary has been passed through specs. Defaults to None ntFlux (int): Number of intervals to split integration into for computing total SNR. When greater than 1, SNR is effectively computed as a Reimann sum. Defaults to 1
- **nVisitsMax** (*int*) – Maximum number of observations (in detection mode) per star. Defaults to 5
- **charMargin** (*float*) – Integration time margin for characterization. Defaults to 0.15
- **dt_max** (*float*) – Maximum time for revisit window (in days). Defaults to 1.
- **record_counts_path** (*str, optional*) – If set, write out photon count info to file specified by this keyword. Defaults to None.
- **nokoMap** (*bool*) – Skip generating keepout map. Only useful if you're not planning on actually running a mission simulation. Defaults to False.
- **nofZ** (*bool*) – Skip precomputing zodical light minima. Only useful if you're not planning on actually running a mission simulation. Defaults to False.
- **cachedir** (*str, optional*) – Full path to cachedir. If None (default) use default (see *Cache Directory*)
- **defaultAddExoplanetObsTime** (*bool*) – If True, time advancement when no targets are observable will add to exoplanetObsTime (i.e., wasting time is counted against you). Defaults to True
- **find_known_RV** (*bool*) – Identify stars with known planets. Defaults to False
- **include_known_RV** (*str, optional*) – Path to file including known planets to include. Defaults to None
- **make_debug_bird_plots** (*bool, optional*) – If True, makes completeness bird plots for every observation that are saved in the cache directory
- **debug_plot_path** (*str, optional*) – Path to save the debug plots in, must be set if make_debug_bird_plots is True
- ****specs** – *Input Specification*

_outspec

Output Specification

Type
dict

absTimefZmin

Absolute time of local zodi minima

Type
astropy.time.core.Time

BackgroundSources

BackgroundSources object

Type
BackgroundSources

cachedir

Path to the EXOSIMS cache directory (see *Cache Directory*)

Type
str

cachefname

Base filename for cache files.

Type
str

charMargin

Integration time margin for characterization.

Type
float

Completeness

Completeness object

Type
Completeness

count_lines

Photon counts. Only used when `record_counts_path` is set

Type
list

defaultAddExoplanetObsTime

If True, time advancement when no targets are observable will add to exoplanetObsTime (i.e., wasting time is counted against you).

Type
bool

DRM

The mission simulation. List of observation dictionaries.

Type
list

dt_max

Maximum time for revisit window.

Type
astropy.units.quantity.Quantity

find_known_RV

Identify stars with known planets.

Type
bool

fullSpectra

Array of booleans indicating whether a planet's spectrum has been fully observed.

Type
numpy.ndarray(bool)

fZmins

Dictionary of local zodi minimum value candidates for each observing mode

Type
dict

fZtypes

Dictionary of type of local zodi minimum candidates for each observing mode

Type
dict

include_known_RV

Path to file including known planets to include.

Type
str, optional

intTimeFilterInds

Indices of targets where integration times fall below cutoff value

Type
numpy.ndarray(ind)

intTimesIntTimeFilter

Default integration times for pre-filtering targets.

Type
astropy.units.quantity.Quantity

known_earths

Indices of Earth-like planets

Type
numpy.ndarray

known_rocky

Indices of rocky planets

Type
list

known_stars

Stars with known planets

Type
list

koMaps

Keepout Maps

Type
dict

koTimes

Times corresponding to keepout map array entries.

Type
astropy.time.core.Time

lastDetected

ntarg x 4. For each target, contains 4 lists with planets' detected status (boolean), exozodi brightness (in units of 1/arcsec2), delta magnitude, and working angles (in units of arcsec)

Type
numpy.ndarray

lastObsTimes

Contains the last observation start time for future completeness update in units of day

Type
astropy.units.quantity.Quantity

logger

Logger object

Type
logging.Logger

modules

Modules dictionary.

Type
dict

ntFlux

Number of intervals to split integration into for computing total SNR. When greater than 1, SNR is effectively computed as a Reimann sum.

Type
int

nVisitsMax

Maximum number of observations (in detection mode) per star.

Type
int

Observatory

Observatory object.

Type
Observatory

`OpticalSystem`
Optical system object.

Type
OpticalSystem

`partialSpectra`
Array of booleans indicating whether a planet's spectrum has been partially observed.

Type
numpy.ndarray

`PlanetPhysicalModel`
Planet pysical model object

Type
PlanetPhysicalModel

`PlanetPopulation`
Planet population object

Type
PlanetPopulation

`PostProcessing`
Postprocessing object

Type
PostProcessing

`propagTimes`
Contains the current time at each target system.

Type
astropy.units.quantity.Quantity

`record_counts_path`
If set, write out photon count info to file specified by this keyword.

Type
str, optional

`seed`
Seed for random number generation

Type
int

`SimulatedUniverse`
Simulated universe object

Type
SimulatedUniverse

`StarCatalog`
Star catalog object (only if `keepStarCatalog` input is True.

Type
StarCatalog

starExtended

TBD

Type
numpy.ndarray

starRevisit

ntargs x 2. Contains indices of targets to revisit and revisit times of these targets in units of day

Type
numpy.ndarray

starVisits

ntargs x 1. Contains the number of times each target was visited

Type
numpy.ndarray

TargetList

TargetList object.

Type
TargetList

TimeKeeping

Timekeeping object

Type
TimeKeeping

valfZmin

Minimum local zodi for each target.

Type
astropy.units.quantity.Quantity

ZodiacalLight

Zodiacal light object.

Type
ZodiacalLight

arbitrary_time_advancement(*dt*)

Handles fully dynamically scheduled case where OBduration is infinite and missionPortion is less than 1.

Parameters
dt (*Quantity*) – Total amount of time, including all overheads and extras used for the previous observation.

Returns
None

calc_signal_noise(*sInd*, *pInds*, *t_int*, *mode*, *fZ=None*, *fEZ=None*, *dMag=None*, *WA=None*)

Calculates the signal and noise fluxes for a given time interval. Called by observation_detection and observation_characterization methods in the SurveySimulation module.

Parameters

- **sInd** (*int*) – Integer index of the star of interest
- **t_int** (*Quantity(ndarray(float))*) – Integration time interval in units of day

- **pInds** (*int*) – Integer indices of the planets of interest
- **mode** (*dict*) – Selected observing mode (from OpticalSystem)
- **fZ** (*Quantity(ndarray(float))*) – Surface brightness of local zodiacal light in units of 1/arcsec2
- **fEZ** (*Quantity(ndarray(float))*) – Surface brightness of exo-zodiacal light in units of 1/arcsec2
- **dMag** (*ndarray(float)*) – Differences in magnitude between planets and their host star
- **WA** (*Quantity(ndarray(float))*) – Working angles of the planets of interest in units of arcsec

Returns

Signal (float):
Counts of signal

Noise (float):
Counts of background noise variance

Return type
tuple

calc_targ_intTime(*sInds*, *startTimes*, *mode*)

Helper method for next_target to aid in overloading for alternative implementations.

Given a subset of targets, calculate their integration times given the start of observation time.

Prototype just calculates integration times for fixed contrast depth.

Parameters

- **sInds** (*ndarray(int)*) – Indices of available targets
- **startTimes** (*astropy quantity array*) – absolute start times of observations. must be of the same size as sInds
- **mode** (*dict*) – Selected observing mode for detection

Returns
Integration times for detection same dimension as sInds

Return type
Quantity(*ndarray*(float))

Note: next_target filter will discard targets with zero integration times.

chooseOcculterSlewTimes(*sInds*, *slewTimes*, *dV*, *intTimes*, *charTimes*)

Selects the best slew time for each star

This method searches through an array of permissible slew times for each star and chooses the best slew time for the occulter based on maximizing possible characterization time for that particular star (as a default).

Parameters

- **sInds** (*ndarray(int)*) – Indices of available targets
- **slewTimes** (*astropy quantity array*) – slew times to all stars (must be indexed by sInds)

- **dV** (*Quantity(ndarray(float))*) – Delta-V used to transfer to new star line of sight in unis of m/s
- **intTimes** (*Quantity(ndarray(float))*) – Integration times for detection in units of day
- **charTimes** (*Quantity(ndarray(float))*) – Time left over after integration which could be used for characterization in units of day

Returns

sInds (int):
Indeces of next target star

slewTimes (astropy.units.Quantity(numpy.ndarray(float))):
slew times to all stars (must be indexed by sInds)

intTimes (astropy.units.Quantity(numpy.ndarray(float))):
Integration times for detection in units of day

dV (astropy.units.Quantity(numpy.ndarray(float))):
Delta-V used to transfer to new star line of sight in unis of m/s

Return type
tuple

choose_next_target(*old_sInd*, *sInds*, *slewTimes*, *intTimes*)

Helper method for method next_target to simplify alternative implementations.

Given a subset of targets (pre-filtered by method next_target or some other means), select the best next one. The prototype uses completeness as the sole heuristic.

Parameters

- **old_sInd** (*int*) – Index of the previous target star
- **sInds** (*ndarray(int)*) – Indices of available targets
- **slewTimes** (*Quantity(ndarray(float))*) – slew times to all stars (must be indexed by sInds)
- **intTimes** (*Quantity(ndarray(float))*) – Integration times for detection in units of day

Returns

sInd (int):
Index of next target star

waitTime (Quantity):
Some strategic amount of time to wait in case an occulter slew is desired (default is None)

Return type
tuple

filterOcculterSlews(*sInds*, *slewTimes*, *obsTimeArray*, *intTimeArray*, *mode*)

Filters occulter slews that have already been calculated/selected.

Used by the refineOcculterSlews method when slew times have been selected a priori. This method filters out slews that are not within desired observing blocks, the maximum allowed integration time, and are outside of future keepouts.

Parameters

- **sInds** (*ndarray(int)*) – Indices of available targets

- **slewTimes** (*Quantity(ndarray(float))*) – slew times to all stars (must be indexed by sInds)
- **obsTimeArray** (*Quantity(ndarray(float))*) – Array of times during which a star is out of keepout, has shape nx50 where n is the number of stars in sInds. Unit of days
- **intTimeArray** (*Quantity(ndarray(float))*) – Array of integration times for each time in obsTimeArray, has shape nx2 where n is the number of stars in sInds. Unit of days
- **mode** (*dict*) – Selected observing mode for detection

Returns

sInds (int):
Indeces of next target star

intTimes (astropy.units.Quantity(numpy.ndarray(float))):
Integration times for detection in units of day

slewTimes (astropy.units.Quantity(numpy.ndarray(float))):
slew times to all stars (must be indexed by sInds)

Return type
tuple

findAllowableOcculterSlews(*sInds*, *old_sInd*, *sd*, *slewTimes*, *obsTimeArray*, *intTimeArray*, *mode*)

Finds an array of allowable slew times for each star

Used by the refineOcculterSlews method when slew times have NOT been selected a priori. This method creates nx50 arrays (where the row corresponds to a specific star and the column corresponds to a future point in time relative to currentTime).

These arrays are initially zero but are populated with the corresponding values (slews, intTimes, etc) if slewing to that time point (i.e. beginning an observation) would lead to a successful observation. A "successful observation" is defined by certain conditions relating to keepout and the komap, observing blocks, mission lifetime, and some constraints on the dVmap calculation in SotoStarshade. Each star will likely have a range of slewTimes that would lead to a successful observation – another method is then called to select the best of these slewTimes.

Parameters

- **sInds** (*ndarray(int)*) – Indices of available targets
- **old_sInd** (*int*) – Index of the previous target star
- **sd** (*Quantity(ndarray(float))*) – Angular separation between stars in rad
- **slewTimes** (*astropy quantity array*) – slew times to all stars (must be indexed by sInds)
- **obsTimeArray** (*Quantity(ndarray(float))*) – Array of times during which a star is out of keepout, has shape nx50 where n is the number of stars in sInds
- **intTimeArray** (*Quantity(ndarray(float))*) – Array of integration times for each time in obsTimeArray, has shape nx50 where n is the number of stars in sInds
- **mode** (*dict*) – Selected observing mode for detection

Returns

sInds (numpy.ndarray(int)):
Indices of next target star

slewTimes (astropy.units.Quantity(numpy.ndarray(float))):
slew times to all stars (must be indexed by sInds)

intTimes (astropy.units.Quantity(numpy.ndarray(float))):
Integration times for detection in units of day

dV (astropy.units.Quantity(numpy.ndarray(float))):
Delta-V used to transfer to new star line of sight in unis of m/s

Return type
tuple

`find_known_plans`()

Find and return list of known RV stars and list of stars with earthlike planets based on info from David Plavchan dated 12/24/2018

`genOutSpec`(*tofile: Optional[str] = None*, *starting_outspec: Optional[Dict[str, Any]] = None*, *modnames: bool = False*) → Dict[str, Any]

Join all _outspec dicts from all modules into one output dict and optionally write out to JSON file on disk.

Parameters

- **tofile** (*str, optional*) – Name of the file containing all output specifications (outspecs). Defaults to None.
- **starting_outspec** (*dict, optional*) – Initial outspec (from MissionSim). Defaults to None.
- **modnames** (*bool*) – If True, populate outspec dictionary with the module it originated from, instead of the actual value of the keyword. Defaults False.

Returns
Dictionary containing the full *Input Specification*, including all filled-in default values. Combination of all individual module _outspec attributes.

Return type
dict

`generateHashfName`(*specs*)

Generate cached file Hashname

Requires a .XXX appended to end of hashname for each individual use case

Parameters
specs (*dict*) – *Input Specification*

Returns
Unique indentifier string for cahce products from this set of modules and inputs

Return type
str

`initializeStorageArrays`()

Initialize all storage arrays based on # of stars and targets

`is_earthlike`(*plan_inds*, *sInd*)

Is the planet earthlike?

`next_target`(*old_sInd*, *mode*)

Finds index of next target star and calculates its integration time.

This method chooses the next target star index based on which stars are available, their integration time, and maximum completeness. Returns None if no target could be found.

Parameters

- **old_sInd** (*int*) – Index of the previous target star
- **mode** (*dict*) – Selected observing mode for detection

Returns

DRM (dict):
Design Reference Mission, contains the results of one complete observation (detection and characterization)

sInd (int):
Index of next target star. Defaults to None.

intTime (astropy.units.Quantity):
Selected star integration time for detection in units of day. Defaults to None.

waitTime (astropy.units.Quantity):
a strategically advantageous amount of time to wait in the case of an occulter for slew times

Return type
tuple

observation_characterization(*sInd*, *mode*)

Finds if characterizations are possible and relevant information

Parameters

- **sInd** (*int*) – Integer index of the star of interest
- **mode** (*dict*) – Selected observing mode for characterization

Returns

characterized (list(int)):
Characterization status for each planet orbiting the observed target star including False Alarm if any, where 1 is full spectrum, -1 partial spectrum, and 0 not characterized

fZ (astropy.units.Quantity(numpy.ndarray(float))):
Surface brightness of local zodiacal light in units of 1/arcsec2

systemParams (dict):
Dictionary of time-dependant planet properties averaged over the duration of the integration

SNR (numpy.ndarray(float)):
Characterization signal-to-noise ratio of the observable planets. Defaults to None.

intTime (astropy.units.Quantity(numpy.ndarray(float))):
Selected star characterization time in units of day. Defaults to None.

Return type
tuple

observation_detection(*sInd*, *intTime*, *mode*)

Determines SNR and detection status for a given integration time for detection. Also updates the lastDetected and starRevisit lists.

Parameters

- **sInd** (*int*) – Integer index of the star of interest
- **intTime** (*Quantity(ndarray(float))*) – Selected star integration time for detection in units of day. Defaults to None.
- **mode** (*dict*) – Selected observing mode for detection

Returns

detected (numpy.ndarray(int)):
Detection status for each planet orbiting the observed target star: 1 is detection, 0 missed detection, -1 below IWA, and -2 beyond OWA

fZ (astropy.units.Quantity(numpy.ndarray(float))):
Surface brightness of local zodiacal light in units of 1/arcsec2

systemParams (dict):
Dictionary of time-dependant planet properties averaged over the duration of the integration

SNR (numpy.darray(float)):
Detection signal-to-noise ratio of the observable planets

FA (bool):
False alarm (false positive) boolean

Return type
tuple

`refineOcculterSlews`(*old_sInd*, *sInds*, *slewTimes*, *obsTimes*, *sd*, *mode*)

Refines/filters/chooses occulter slews based on time constraints

Refines the selection of occulter slew times by filtering based on mission time constraints and selecting the best slew time for each star. This method calls on other occulter methods within SurveySimulation depending on how slew times were calculated prior to calling this function (i.e. depending on which implementation of the Observatory module is used).

Parameters

- **old_sInd** (*int*) – Index of the previous target star
- **sInds** (*ndarray(int)*) – Indices of available targets
- **slewTimes** (*astropy quantity array*) – slew times to all stars (must be indexed by sInds)
- **obsTimes** (*Quantity(ndarray(float))*) – A binary array with TargetList.nStars rows and (missionFinishAbs-missionStart)/dt columns where dt is 1 day by default. A value of 1 indicates the star is in keepout (and therefore cannot be observed). A value of 0 indicates the star is not in keepout and may be observed.
- **sd** (*Quantity(ndarray(float))*) – Angular separation between stars in rad
- **mode** (*dict*) – Selected observing mode for detection

Returns

sInds (int):
Indeces of next target star

slewTimes (astropy.units.Quantity(numpy.ndarray(float))):
slew times to all stars (must be indexed by sInds)

intTimes (astropy.units.Quantity(numpy.ndarray(float))):
Integration times for detection in units of day

dV (astropy.units.Quantity(numpy.ndarray(float))):
Delta-V used to transfer to new star line of sight in unis of m/s

Return type
tuple

reset_sim(*genNewPlanets=True*, *rewindPlanets=True*, *seed=None*)

Performs a full reset of the current simulation.

This will reinitialize the TimeKeeping, Observatory, and SurveySimulation objects with their own outspecs.

Parameters

- **genNewPlanets** (*bool*) – Generate all new planets based on the original input specification. If False, then the original planets will remain. Setting to True forces `rewindPlanets` to be True as well. Defaults True.
- **rewindPlanets** (*bool*) – Reset the current set of planets to their original orbital phases. If both genNewPlanets and rewindPlanet are False, the original planets will be retained and will not be rewound to their initial starting locations (i.e., all systems will remain at the times they were at the end of the last run, thereby effectively randomizing planet phases. Defaults True.
- **seed** (*int, optional*) – Random seed to use for all random number generation. If None (default) a new random seed will be generated when re-initializing the SurveySimulation.

revisitFilter(*sInds*, *tmpCurrentTimeNorm*)

Helper method for Overloading Revisit Filtering

Parameters

- **sInds** (*ndarray(int)*) – Indices of stars still in observation list
- **tmpCurrentTimeNorm** (*astropy.units.Quantity*) – The simulation time after overhead was added in MJD form

Returns
indices of stars still in observation list

Return type
ndarray(int)

run_sim()

Performs the survey simulation

scheduleRevisit(*sInd*, *smin*, *det*, *pInds*)

A Helper Method for scheduling revisits after observation detection

Updates self.starRevisit attribute only

Parameters

- **sInd** (*int*) – sInd of the star just detected
- **smin** (*Quantity*) – minimum separation of the planet to star of planet just detected
- **det** (*ndarray(bool)*) – Detection status of all planets in target system
- **pInds** (*ndarray(int)*) – Indices of planets in the target system

Returns
None

update_occulter_mass(*DRM*, *sInd*, *t_int*, *skMode*)

Updates the occulter wet mass in the Observatory module, and stores all the occulter related values in the DRM array.

Parameters

- **DRM** (*dict*) – Design Reference Mission, contains the results of one complete observation (detection and characterization)
- **sInd** (*int*) – Integer index of the star of interest
- **t_int** (*Quantity(ndarray(float))*) – Selected star integration time (for detection or characterization) in units of day
- **skMode** (*str*) – Station keeping observing mode type (‘det’ or ‘char’)

Returns
Design Reference Mission dictionary, contains the results of one complete observation

Return type
dict

EXOSIMS.Prototypes.SurveySimulation.**array_encoder**(*obj*)

Encodes numpy arrays, astropy Times, and astropy Quantities, into JSON.

Called from json.dump for types that it does not already know how to represent, like astropy Quantity’s, numpy arrays, etc. The json.dump() method encodes types like ints, strings, and lists itself, so this code does not see these types. Likewise, this routine can and does return such objects, which is OK as long as they unpack recursively into types for which encoding is known

Parameters

- **obj** (*Any*) – Object to encode.
- **Returns** –

Any:
Encoded object

EXOSIMS.Prototypes.TargetList module

class EXOSIMS.Prototypes.TargetList.**TargetList**(*missionStart=60634*, *staticStars=True*, *keepStarCatalog=False*, *fillPhotometry=False*, *fillMissingBandMags=False*, *explainFiltering=False*, *filterBinaries=True*, *cachedir=None*, *filter_for_char=False*, *earths_only=False*, *getKnownPlanets=False*, *int_WA=None*, *int_dMag=25*, *scaleWAdMag=False*, *popStars=None*, *cherryPickStars=None*, *skipSaturationCalcs=True*, ***specs*)

Bases: object

TargetList Prototype

Instantiation of an object of this class requires the instantiation of the following class objects:

- StarCatalog
- OpticalSystem
- ZodiacalLight
- Completeness
- PostProcessing

Parameters

- **missionStart** (*float*) – Mission start time (MJD)
- **staticStars** (*bool*) – Do not apply proper motions to stars during simulation. Defaults True.
- **keepStarCatalog** (*bool*) – Retain the StarCatalog object as an attribute after the target list is populated (defaults False)
- **fillPhotometry** (*bool*) – Attempt to fill in missing photometric data for targets from available values (primarily spectral type and luminosity). Defaults False.
- **fillMissingBandMags** (*bool*) – If `fillPhotometry` is True, also fill in missing band magnitudes. Ignored if `fillPhotometry` is False. Defaults False.

  Warning: This can be used for generating more complete target data for other uses but is *not* recommended for use in integration time calculations.

- **explainFiltering** (*bool*) – Print informational messages at each target list filtering step. Defaults False.
- **filterBinaries** (*bool*) – Remove all binary stars or stars with known close companions from target list. Defaults True.
- **cachedir** (*str or None*) – Full path to cache directory. If None, use default.
- **filter_for_char** (*bool*) – Use spectroscopy observation mode (rather than the default detection mode) for all calculations. Defaults False.
- **earths_only** (*bool*) – Used in upstream modules. Alias for filter_for_char. Defaults False.
- **getKnownPlanets** (*bool*) – Retrieve information about which stars have known planets along with all known (to the NASA Exoplanet Archive) target aliases. Defaults False.

  Warning: This can take a *very* long time for large star catalogs if starting from scratch. For this reason, the cache comes pre-loaded with all entries coresponding to EXOCAT1.

- **int_WA** (*float or numpy.ndarray or None*) – Working angle (arcsec) at which to caluclate integration times for default observations. If input is scalar, all targets will get the same value. If input is an array, it must match the size of the input catalog. If None, a default value of halway between the inner and outer working angle of the default observing mode will be used. If the OWA is infinite, twice the IWA is used.
- **int_dMag** (*float or numpy.ndarray*) –
  $Delta$
  $textrmmag$ to assume when calculating integration time for default observations. If input is scalar, all targets will get the same value. If input is an array, it must match the size of the input catalog. Defaults to 25
- **scaleWAdMag** (*bool*) – If True, rescale int_dMag and int_WA for all stars based on luminosity and to ensure that WA is within the IWA/OWA. Defaults False.
- **popStars** (*list, optional*) – Remove given stars (by exact name matching) from target list. Defaults None.
- **cherryPickStars** (*list, optional*) – Before doing any other filtering, filter out all stars from input star catalog *excep* for the ones in this list (by exact name matching). Defaults None (do not initially filter out any stars from the star catalog).

- **skipSaturationCalcs** (*bool*) – If True, do not perform any new calculations for saturation dMag and saturation completeness (cached values will still be loaded if found on disk). The saturation_dMag and saturation_comp will all be set to NaN if this keyword is set True and no cached values are found. No cache will be written in that case. Defaults False.
- ****specs** – *Input Specification*

_outspec

Output Specification

Type
dict

BackgroundSources

BackgroundSources object

Type
BackgroundSources

BC

Bolometric correction (V band)

Type
numpy.ndarray

Binary_Cut

Boolean - True is target has close companion.

Type
numpy.ndarray

blackbody_spectra

Storage array for blackbody spectra (populated as needed)

Type
numpy.ndarray

Bmag

B band magniutde

Type
numpy.ndarray

BV

B-V color

Type
numpy.ndarray

cachedir

Path to the EXOSIMS cache directory (see *Cache Directory*)

Type
str

calc_char_int_comp

Boolean flagged by `filter_for_char` or `earths_only`

Type
bool

`catalog_atts`
All star catalog attributes that were copied in

Type
list

`cherryPickStars`
List of star names to keep from input star catalog (all others are filtered out prior to any other filtering).

Type
list

`Completeness`
Completeness object

Type
Completeness

`coords`
Target coordinates

Type
astropy.coordinates.sky_coordinate.SkyCoord

`diameter`
Stellar diameter in angular units.

Type
astropy.units.quantity.Quantity

`dist`
Target distances

Type
astropy.units.quantity.Quantity

`earths_only`
Used in upstream modules. Alias for filter_for_char.

Type
bool

`explainFiltering`
Print informational messages at each target list filtering step.

Type
bool

`fillPhotometry`
Attempt to fill in missing target photometric values using interpolants of tabulated values for the stellar type. See MeanStars documentation for more details.

Type
bool

`fillMissingBandMags`
If `self.fillPhotometry` is True, also fill in missing band magnitudes. Ignored if `self.fillPhotometry` is False.

Type
bool

`filter_for_char`
Use spectroscopy observation mode (rather than the default detection mode) for all calculations.

Type
bool

`filterBinaries`
Remove all binary stars or stars with known close companions from target list.

Type
bool

`filter_mode`
OpticalSystem observingMode dictionary. The observingMode used for target filtering. Either the detection mode (default) or first characterization mode (if `filter_for_char` is True).

Type
dict

`getKnownPlanets`
Grab the list of known planets and target aliases from the NASA Exoplanet Archive

Type
bool

`hasKnownPlanet`
bool array indicating whether a target has known planets. Only populated if attribute `getKnownPlanets` is True. Otherwise all entries are False.

Type
numpy.ndarray

`Hmag`
H band magnitudes

Type
numpy.ndarray

`Imag`
I band magnitudes

Type
numpy.ndarray

`int_comp`
Completeness value for each target star for default observation WA and $\Delta textrmmag$.

Type
numpy.ndarray

`int_dMag`
$\Delta textrmmag$ used for default observation integration time calculation

Type
numpy.ndarray

`int_tmin`
Integration times corresponding to *int_dMag* with global minimum local zodi contribution.

Type
astropy.units.quantity.Quantity

int_WA

Working angle used for integration time calculation (angle)

Type
astropy.units.quantity.Quantity

intCutoff_comp

Completeness values of all targets corresponding to the cutoff integration time set in the optical system.

Type
numpy.ndarray

intCutoff_dMag

$\Delta textrmmag$ of all targets corresponding to the cutoff integration time set in the optical system.

Type
numpy.ndarray

Jmag

J band magnitudes

Type
numpy.ndarray

keepStarCatalog

Keep star catalog object as attribute after TargetList is built.

Type
bool

Kmag

K band mangitudes

Type
numpy.ndarray

L

Luminosities in solar luminosities (linear scale!)

Type
numpy.ndarray

ms

MeanStars object

Type
MeanStars.MeanStars.MeanStars

MsEst

'approximate' stellar masses

Type
astropy.units.quantity.Quantity

MsTrue

'true' stellar masses

Type
astropy.units.quantity.Quantity

MV

Absolute V band magnitudes

Type
numpy.ndarray

Name

Target names (str array)

Type
numpy.ndarray

nStars

Number of stars currently in target list

Type
int

systemOmega

Base longitude of the ascending node for target system orbital planes

Type
astropy.units.quantity.Quantity

OpticalSystem

OpticalSystem object

Type
OpticalSystem

parx

Parallaxes

Type
astropy.units.quantity.Quantity

PlanetPhysicalModel

PlanetPhysicalModel object

Type
PlanetPhysicalModel

PlanetPopulation

PlanetPopulation object

Type
PlanetPopulation

pmdec

Proper motion in declination

Type
astropy.units.quantity.Quantity

pmra

Proper motion in right ascension

Type
astropy.units.quantity.Quantity

popStars

List of target names that were removed from target list

Type
list, optional

PostProcessing

PostProcessing object

Type
PostProcessing

required_catalog_atts

Catalog attributes that may not be missing or nan

Type
list

Rmag

R band magnitudes

Type
numpy.ndarray

rv

Radial velocities

Type
astropy.units.quantity.Quantity

saturation_comp

Maximum possible completeness values of all targets.

Type
numpy.ndarray

saturation_dMag

$\Delta textrmmag$ at which completness stops increasing for all targets.

Type
numpy.ndarray

scaleWAdMag

Rescale int_dMag and int_WA for all stars based on luminosity and to ensure that WA is within the IWA/OWA.

Type
bool

skipSaturationCalcs

If True (default), saturation dMag and saturation completeness are not computed. If cached values exist, they will be loaded, otherwise saturation_dMag and saturation_comp will all be set to NaN. No new cache files will be written for these values.

Type
bool

Spec

Spectral type strings. Will be strictly in G0V format.

Type
numpy.ndarray

specdict

Dictionary of numerical mappings for spectral classes (O = 0, M = 6).

Type
dict

spectral_catalog_index

Dictionary mapping spectral type strings (keys) to template spectra files on disk (values).

Type
dict

spectral_catalog_types

nx4 ndarray (n is the number of template spectra avaiable). First three columns are spectral class (str), subclass (int), and luinosity class (str). The fourth column is a spectral class numeric representation, equaling `specdict[specclass]*10 + subclass`.

Type
numpy.ndarray

spectral_class

nStars x 4 array. Same column definitions as `spectral_catalog_types` but evaluated for the target stars rather than the template spectra.

Type
numpy.ndarray

standard_bands

Dictionary mapping UVBRIJHK (keys are single characters) to `synphot.spectrum.SpectralElement` objects of the filter profiles.

Type
dict

standard_bands_deltaLam

Effective bandpasses of the profiles in *standard_bands*.

Type
astropy.units.quantity.Quantity

standard_bands_lam

Effective central wavelengths of the profiles in *standard_bands*.

Type
astropy.units.quantity.Quantity

standard_bands_letters

Concatenation of the keys of *standard_bands*.

Type
str

star_fluxes

Internal storage of pre-computed star flux values that is populated each time a flux is requested for a particular target. Keyed by observing mode hex attribute.

Type
dict

staticStars

Do not apply proper motions to stars. Stars always at mission start time positions.

Type
bool

systemInclination

Inclinations of target system orbital planes

Type
astropy.units.quantity.Quantity

Teff

Stellar effective temperature.

Type
astropy.units.Quantity

template_spectra

Dictionary of template spectra objects (populated as needed).

Type
dict

Umag

U band magnitudes

Type
numpy.ndarray

Vmag

V band magnitudes

Type
numpy.ndarray

ZodiacalLight

ZodiacalLight object

Type
ZodiacalLight

binary_filter()

Removes stars which have attribute Binary_Cut == True

calc_EEID(*sInds*, *arcsec=False*)

Finds the earth equivalent insolation distance (EEID)

Parameters
sInds (*ndarray(int)*) – Indices of the stars of interest

arcsec (bool):
If True returns result arcseconds instead of AU

Returns
limit of HZ in AU or arcseconds

Return type
Quantity(*ndarray*(float))

calc_HZ(*sInds*, *S*, *A*, *B*, *C*, *arcsec=False*)

finds the inner or outer edge of the habitable zone

This method uses the empirical fit from Kaltinegger et al (2018) and references therein, https://arxiv.org/pdf/1903.11539.pdf

Parameters

- **sInds** (*ndarray(int)*) – Indices of the stars of interest
- **S** (*float*) – Constant
- **A** (*float*) – Constant
- **B** (*float*) – Constant
- **C** (*float*) – Constant
- **arcsec** (*bool*) – If True returns result arcseconds instead of AU

Returns
limit of HZ in AU or arcseconds

Return type
Quantity(*ndarray*(float))

calc_HZ_inner(*sInds*, *S_inner=1.7665*, *A_inner=0.00013351*, *B_inner=3.1515e-09*, *C_inner=-3.3488e-12*, ***kwargs*)

Convenience function to find the inner edge of the habitable zone using the emperical approach in calc_HZ().

Default contstants: Recent Venus limit Inner edge" , Kaltinegger et al 2018, Table 1.

calc_HZ_outer(*sInds*, *S_outer=0.324*, *A_outer=5.3221e-05*, *B_outer=1.4288e-09*, *C_outer=-1.1049e-12*, ***kwargs*)

Convenience function to find the inner edge of the habitable zone using the emperical approach in calc_HZ().

The default outer limit constants are the Early Mars outer limit, Kaltinegger et al (2018) Table 1.

calc_IWA_AU(*sInds*, ***kwargs*)

Convenience function to find the separation from the star of the IWA

Parameters
sInds (*ndarray(int)*) – Indices of the stars of interest

Returns
separation from the star of the IWA in AU

Return type
Quantity array

calc_saturation_and_intCutoff_vals()

Calculates the saturation and integration cutoff time dMag and completeness values, saves them as attributes, refines the dMag used to calculate integration times so it does not exceed the integration cutoff time dMag, and handles any orbit scaling necessary

completeness_filter()

Includes stars if completeness is larger than the minimum value

`dump_catalog()`

Creates a dictionary of stellar properties for archiving use.

Parameters
`None` –

Returns
Dictionary of star catalog properties

Return type
dict

`fgk_filter()`

Includes only F, G, K spectral type stars in Target List

`fillPhotometryVals()`

Attempts to determine the spectral class and luminosity class of each target star. If `self.fillPhotometry` is True, attempts to reconstruct any missing spectral types from other avaialable information, and then fill missing L, B, V and BC information from spectral type.

Uses the MeanStars object for regexps and data filling. This explicitly treats all stars as dwarfs. TODO: Update to use spectra for other luminosity classes.

The data is from: "A Modern Mean Dwarf Stellar Color and Effective Temperature Sequence" http://www.pas.rochester.edu/~emamajek/EEM_dwarf_UBVIJHK_colors_Teff.txt Eric Mamajek (JPL/Caltech, University of Rochester)

See MeanStars documentation for futher details.

TODO: only use MeanStars for dwarfs. Otherwise use spectra.

`filter_target_list(**specs)`

This function is responsible for filtering by any required metrics.

The prototype implementation removes the following stars:

- Stars with NAN values in their parameters
- Binary stars
- Systems with planets inside the OpticalSystem fundamental IWA
- Systems where minimum integration time is longer than OpticalSystem cutoff
- Systems not meeting the Completeness threshold

Additional filters can be provided in specific TargetList implementations.

`gen_Omegas(Orange)`

Randomly Generate longitude of the ascending node of target system orbital planes for all stars in the target list

Parameters
`Orange` (*ndarray(float)*) – The range to generate Omegas over

Returns
System Omegas

Return type
Quantity(*ndarray*(float))

gen_inclinations(*Irange*)

Randomly Generate Inclination of Target System Orbital Plane for all stars in the target list

Parameters
Irange (*ndarray(float)*) – the range to generate inclinations over

Returns
System inclinations

Return type
Quantity(*ndarray*(float))

get_template_spectrum(*spec*)

Helper method for loading/retrieving spectra from the spectral catalog

Parameters
spec (*str*) – Spectral type string. Must be a keys in self.spectral_catalog_index

Returns
Template pectrum from file.

Return type
synphot.SourceSpectrum

life_expectancy_filter()

Removes stars from Target List which have BV < 0.3

load_spectral_catalog()

Helper method for generating a cache of available template spectra and loading them as attributes

Creates the following attributes:

1. spectral_catalog_index: A dictionary of spectral types (keys) and the associated spectra files on disk (values)
2. spectral_catalog_types: An nx4 ndarray (n is the number of teplate spectra avaiable). First three columns are spectral class (str), subclass (int), and luinosity class (str). The fourth column is a spectral class numeric representation, equaling specdict[specclass]*10 + subclass.

load_standard_bands()

Helper method that defines standard photometric bandpasses

This method defines the following class attributes:

1. standard_bands_letters: String with standard band letters (nominally UVBRI)
2. standard_bands: A dictionary (key of band letter) whose values are synphot SpectralElement objects for that bandpass.
3. standard_bands_lam: An array of band central wavelengths (same order as standard_bands_letters
4. standard_bands_deltaLam: An array of band bandwidths (same order as standard_bands_letters.

main_sequence_filter()

Removes stars from Target List which are not main sequence

max_dmag_filter()

Includes stars if maximum delta mag is in the allowed orbital range

Removed from prototype filters. Prototype is already calling the completeness_filter with self.intCutoff_dMag

nan_filter()

Filters out targets where required values are nan

outside_IWA_filter()

Includes stars with planets with orbits outside of the IWA

populate_target_list(***specs*)

This function is responsible for populating values from the star catalog into the target list attributes and enforcing attribute requirements.

Parameters
****specs** – *Input Specification*

queryNEAsystems()

Queries NASA Exoplanet Archive system alias service to check for stars in the target list that have known planets.

Note: These queries take a *long* time, so rather than caching individual target lists, we'll keep everything we query and initially seed from a starting list that's part of the repository.

---

radiusFromMass(*sInds*)

Estimates the star radius based on its mass Table 2, ZAMS models pg321 STELLAR MASS-LUMINOSITY AND MASS-RADIUS RELATIONS OSMAN DEMIRCAN and GOKSEL KAHRAMAN 1991

Parameters
sInds (*ndarray(int)*) – star indices

Returns
Star radius estimates

Return type
Quantity(*ndarray*(float))

revise_lists(*sInds*)

Replaces Target List catalog attributes with filtered values, and updates the number of target stars.

Parameters
sInds (*ndarray(int)*) – Integer indices of the stars to retain

set_catalog_attributes()

Hepler method that sets possible and required catalog attributes.

Sets attributes:

catalog_atts (list):
Attributes to try to copy from star catalog. Missing ones will be ignored and removed from this list.

required_catalog_atts(list):
Attributes that cannot be missing or nan.

Note: This is a separate method primarily for downstream implementations that wish to modify the catalog attributes. Overloaded methods can first call this method to get the base values, or overwrite them entirely.

---

starFlux(*sInds*, *mode*)

Return the total spectral flux of the requested stars for the given observing mode. Caches results internally for faster access in subsequent calls.

Parameters

- **sInds** (*ndarray(int)*) – Indices of the stars of interest
- **mode** (*dict*) – Observing mode dictionary (see *OpticalSystem*)

Returns
Spectral fluxes in units of ph/m**2/s.

Return type
Quantity(*ndarray*(float))

starprop(*sInds*, *currentTime*, *eclip=False*)

Finds target star positions vector in heliocentric equatorial (default) or ecliptic frame for current time (MJD).

This method uses ICRS coordinates which is approximately the same as equatorial coordinates.

Parameters

- **sInds** (*ndarray(int)*) – Integer indices of the stars of interest
- **currentTime** (*Time*) – Current absolute mission time in MJD
- **eclip** (*bool*) – Boolean used to switch to heliocentric ecliptic frame. Defaults to False, corresponding to heliocentric equatorial frame.

Returns
Target star positions vector in heliocentric equatorial (default) or ecliptic frame in units of pc. Will return an m x n x 3 array where m is size of currentTime, n is size of sInds. If either m or n is 1, will return n x 3 or m x 3.

Return type
Quantity(*ndarray*(float))

Note: Use eclip=True to get ecliptic coordinates.

stellar_Teff()

Calculate the effective stellar temperature based on B-V color.

This method uses the empirical fit from [Ballesteros2012] doi:10.1209/0295-5075/97/34008

Updates/creates attribute `Teff`, as needed.

stellar_diameter()

Populates target list with approximate stellar diameters

Stellar radii are computed using the BV target colors according to the model from [Boyajian2014]. This model has a standard deviation error of 7.8%.

Updates/creates attribute `diameter`, as needed.

stellar_mass()

Populates target list with 'true' and 'approximate' stellar masses

Approximate stellar masses are calculated from absolute magnitudes using the model from [Henry1993]. "True" masses are generated by a uniformly distributed, 7%-mean error term to the apprxoimate masses.

All values are in units of solar mass.

Function called by reset sim.

vis_mag_filter(*Vmagcrit*)

Includes stars which are below the maximum apparent visual magnitude

Parameters
Vmagcrit (*float*) – maximum apparent visual magnitude

zero_lum_filter()

Filter Target Stars with 0 luminosity

EXOSIMS.Prototypes.TimeKeeping module

class EXOSIMS.Prototypes.TimeKeeping.**TimeKeeping**(*missionStart=60634*, *missionLife=0.1*, *missionPortion=1*, *OBduration=inf*, *missionSchedule=None*, *cachedir=None*, ***specs*)

Bases: object

TimeKeeping Prototype

This class keeps track of the current mission elapsed time for exoplanet mission simulation. It is initialized with a mission duration, and throughout the simulation, it allocates temporal intervals for observations. Eventually, all available time has been allocated, and the mission is over. Time is allocated in contiguous windows of size "duration". If a requested interval does not fit in the current window, we move to the next one.

Parameters

- **missionStart** (*float*) – Mission start date in MJD. Defaults to 60634 (11-20-2024)
- **missionLife** (*float*) – Mission duration (in years). Defaults to 0.1
- **missionPortion** (*float*) – Fraction of mission devoted to exoplanet imaging science. Must be between 0 and 1. Defaults to 1
- **OBduration** (*float*) – Observing block length (in days). If infinite, do not define observing blocks. Defaults to np.inf
- **missionSchedule** (*str, optional*) – Full path to mission schedule file stored on disk. Defaults None.
- **cachedir** (*str, optional*) – Full path to cachedir. If None (default) use default (see *Cache Directory*)
- ****specs** – *Input Specification*

_outspec

Output Specification

Type
dict

cachedir

Path to the EXOSIMS cache directory (see *Cache Directory*)

Type
str

currentTimeAbs

Current absolute mission time in MJD

Type
astropy.time.core.Time

`currentTimeNorm`

Current mission time minus mission start time.

Type
astropy.units.quantity.Quantity

`exoplanetObsTime`

How much time has been used so far on exoplanet science.

Type
astropy.units.quantity.Quantity

`missionFinishAbs`

Mission end time in MJD

Type
astropy.time.core.Time

`missionLife`

Total mission duration

Type
astropy.units.quantity.Quantity

`missionPortion`

Fraction of mission devoted to exoplanet science

Type
float

`missionStart`

Start time of mission in MJD

Type
astropy.time.core.Time

`OBduration`

Observing block length

Type
astropy.units.quantity.Quantity

`OBendTimes`

Array containing the normalized end times of each observing block throughout the mission

Type
astropy.units.quantity.Quantity

`OBnumber`

Index of the current observing block (OB). Each observing block has a duration, a start time, an end time, and can host one or multiple observations

Type
int

`OBstartTimes`

Array containing the normalized start times of each observing block throughout the mission

Type
astropy.units.quantity.Quantity

advanceToAbsTime(*tAbs*, *addExoplanetObsTime=True*)

Advances the current mission time to tAbs. Updates attributes currentTimeNorma dn currentTimeAbs

Parameters

- **tAbs** (*Time*) – The absolute mission time to advance currentTimeAbs to. MUST HAVE scale='tai'
- **addExoplanetObsTime** (*bool*) – A flag indicating whether to add advanced time to exoplanetObsTime or not

Returns

A bool indicating whether the operation was successful or not

Return type

bool

advancetToStartOfNextOB()

Advances to Start of Next Observation Block This method is called in the allocate_time() method of the TimeKeeping class object, when the allocated time requires moving outside of the current OB. If no OB duration was specified, a new Observing Block is created for each observation in the SurveySimulation module. Updates attributes OBnumber, currentTimeNorm and currentTimeAbs.

allocate_time(*dt*, *addExoplanetObsTime=True*)

Allocate a temporal block of width dt

Advance the mission time by dt units. Updates attributes currentTimeNorm and currentTimeAbs

Parameters

- **dt** (*Quantity*) – Temporal block allocated in units of days
- **addExoplanetObsTime** (*bool*) – Indicates the allocated time is for the primary instrument (True) or some other instrument (False) By default this function assumes all allocated time is attributed to the primary instrument (is True)

Returns

a flag indicating the time allocation was successful or not successful

Return type

bool

get_ObsDetectionMaxIntTime(*Obs*, *mode*, *currentTimeNorm=None*, *OBnumber=None*)

Tells you the maximum Detection Observation Integration Time you can pass into observation_detection(X,intTime,X)

Parameters

- **Obs** (*Observatory*) – Observatory object
- **mode** (*dict*) – Selected observing mode for detection

Returns

maxIntTimeOBendTime (astropy.units.Quantity):
The maximum integration time bounded by Observation Block end Time

maxIntTimeExoplanetObsTime (astropy.units.Quantity):
The maximum integration time bounded by exoplanetObsTime

maxIntTimeMissionLife (astropy.units.Quantity):
The maximum integration time bounded by MissionLife

Return type
tuple

init_OB(*missionSchedule*, *OBduration*)

Initializes mission Observing Blocks from file or missionDuration, missionLife, and missionPortion. Updates attributes OBstartTimes, OBendTimes, and OBnumber

Parameters

- **missionSchedule** (*str*) – A string containing the missionSchedule file (or "None").
- **OBduration** (*Quantity*) – Observing block length

Returns
None

mission_is_over(*OS*, *Obs*, *mode*)

Are the mission time, or any other mission resources, exhausted?

Parameters

- **OS** (*OpticalSystem*) – Optical System object
- **Obs** (*Observatory*) – Observatory object
- **mode** (*dict*) – Selected observing mode for detection (uses only overhead time)

Returns
True if the mission time or fuel are used up, else False.

Return type
bool

EXOSIMS.Prototypes.ZodiacalLight module

class EXOSIMS.Prototypes.ZodiacalLight.**ZodiacalLight**(*magZ=23*, *magEZ=22*, *varEZ=0*, *cachedir=None*, *commonSystemfEZ=False*, ***specs*)

Bases: object

ZodiacalLight Prototype

Parameters

- **magZ** (*float*) – Local zodi brightness (magnitudes per square arcsecond). Defaults to 23
- **magEZ** (*float*) – Exozodi brightness (mangitudes per square arcsecond). Defaults to 22
- **varEZ** (*float*) – Variance of exozodi brightness. If non-zero treat as the variance of a log-normal distribution. If zero, do not randomly distribute exozodi brightnesses. Defaults to 0
- **cachedir** (*str, optional*) – Full path to cachedir. If None (default) use default (see *Cache Directory*)
- **commonSystemfEZ** (*bool*) – Assume same zodi for planets in the same system. Defaults to False. TODO: Move to SimulatedUniverse
- ****specs** – *Input Specification*

_outspec

Output Specification

Type
dict

cachedir

Path to the EXOSIMS cache directory (see *Cache Directory*)

Type
str

commonSystemfEZ

Assume same zodi for planets in the same system.

Type
bool

fEZ0

Default surface brightness of exo-zodiacal light in units of 1/arcsec2

Type
astropy.units.quantity.Quantity

fZ0

Default surface brightness of zodiacal light in units of 1/arcsec2

Type
astropy.units.quantity.Quantity

fZMap

For each starlight suppression system (dict key), holds an array of the surface brightness of zodiacal light in units of 1/arcsec2 for each star over 1 year at discrete points defined by resolution

Type
dict

fZTimes

Absolute MJD mission times from start to end

Type
Time(*ndarray*(float))

global_min

The global minimum zodiacal light value

Type
float

magEZ

1 exo-zodi brightness magnitude (per arcsec2)

Type
float

magZ

1 zodi brightness magnitude (per arcsec2)

Type
float

varEZ

Variance of exozodi brightness. If non-zero treat as the variance of a log-normal distribution. If zero, do not randomly distribute exozodi brightnesses.

Type
float

zodi_Blam

Local zodi table data (W/m2/sr/um)

Type
numpy.ndarray

zodi_lam

Local zodi table data wavelengths (micrometers)

Type
numpy.ndarray

calcfZmax(*sInds*, *Obs*, *TL*, *TK*, *mode*, *hashname*, *koTimes=None*)

Finds the maximum zodiacal light values for each star over an entire orbit of the sun not including keeoput angles.

Parameters

- **sInds** (*ndarray(int)*) – the star indicies we would like fZmax and fZmaxInds returned for
- **Obs** (*Observatory*) – Observatory class object
- **TL** (*TargetList*) – TargetList class object
- **TK** (*TimeKeeping*) – TimeKeeping object
- **mode** (*dict*) – Selected observing mode
- **hashname** (*str*) – hashname describing the files specific to the current json script
- **koTimes** (*Time(ndarray(float)), optional*) – Absolute MJD mission times from start to end in steps of 1 d

Returns

valfZmax[sInds] (~astropy.units.Quantity(~numpy.ndarray(float))):
the maximum fZ (for the prototype, these all have the same value) with units 1/arcsec**2

absTimefZmax[sInds] (astropy.time.Time):
returns the absolute Time the maximum fZ occurs (for the prototype, these all have the same value)

Return type
tuple

calcfZmin(*sInds*, *Obs*, *TL*, *TK*, *mode*, *hashname*, *koMap=None*, *koTimes=None*)

Finds the minimum zodiacal light values for each star over an entire orbit of the sun not including keeoput angles.

Parameters

- **sInds** (*ndarray(int)*) – the star indicies we would like fZmins and fZtypes returned for
- **Obs** (*Observatory*) – Observatory class object
- **TL** (*TargetList*) – TargetList class object

- **TK** (*TimeKeeping*) – TimeKeeping object
- **mode** (*dict*) – Selected observing mode
- **hashname** (*str*) – hashname describing the files specific to the current json script
- **koMap** (*ndarray(bool), optional*) – True is a target unobstructed and observable, and False is a target unobservable due to obstructions in the keepout zone.
- **koTimes** (*Time(ndarray(float)), optional*) – Absolute MJD mission times from start to end, in steps of 1 d as default

Returns

fZmins[n, TL.nStars] (~astropy.units.Quantity(~numpy.ndarray(float))):
fZMap, but only fZmin candidates remain. All other values are set to the maximum floating number. Units are 1/arcsec2

fZtypes [n, TL.nStars] (~numpy.ndarray(float)):
ndarray of flags for fZmin types that map to fZmins 0 - entering KO 1 - exiting KO 2 - local minimum max float - not a fZmin candidate

Return type
tuple

extractfZmin(*fZmins*, *sInds*, *koTimes=None*)

Extract the global fZminimum from fZmins

Parameters

- **fZmins[n** (*Quantity(ndarray(float))*) – fZMap, but only fZmin candidates remain. All other values are set to the maximum floating number. Units are 1/arcsec2.
- **TL.nStars]** (*Quantity(ndarray(float))*) – fZMap, but only fZmin candidates remain. All other values are set to the maximum floating number. Units are 1/arcsec2.
- **sInds** (*ndarray(int)*) – the star indicies we would like valfZmin and absTimefZmin returned for
- **koTimes** (*Time(ndarray(float)), optional*) – Absolute MJD mission times from start to end, in steps of 1 d as default

Returns

valfZmin[sInds] (~astropy.units.Quantity(~numpy.ndarray(float))):
the minimum fZ (for the prototype, these all have the same value) with units 1/arcsec**2

absTimefZmin[sInds] (astropy.time.Time):
returns the absolute Time the maximum fZ occurs (for the prototype, these all have the same value)

Return type
tuple

fEZ(*MV*, *I*, *d*, *alpha=2*, *tau=1*)

Returns surface brightness of exo-zodiacal light

Parameters

- **MV** (*ndarray(int)*) – Absolute magnitude of the star (in the V band)
- **I** (*Quantity(ndarray(float))*) – Inclination of the planets of interest in units of deg
- **d** (*Quantity(ndarray(float))*) – nx3 Distance to star of the planets of interest in units of AU

- **alpha** (*float*) – power applied to radial distribution, default=2
- **tau** (*float*) – disk morphology dependent throughput correction factor, default =1

Returns
Surface brightness of exo-zodiacal light in units of 1/arcsec2

Return type
Quantity(*ndarray*(float))

fZ(*Obs*, *TL*, *sInds*, *currentTimeAbs*, *mode*)

Returns surface brightness of local zodiacal light

Parameters

- **Obs** (*Observatory*) – Observatory class object
- **TL** (*TargetList*) – TargetList class object
- **sInds** (*ndarray(int)*) – Integer indices of the stars of interest
- **currentTimeAbs** (*Time*) – absolute time to evaluate fZ for
- **mode** (*dict*) – Selected observing mode

Returns
Surface brightness of zodiacal light in units of 1/arcsec2

Return type
Quantity(*ndarray*(float))

gen_systemnEZ(*nStars*)

Ranomly generates the number of Exo-Zodi

Parameters
nStars (*int*) – number of exo-zodi to generate

Returns
numpy array of exo-zodi values in number of local zodi

Return type
ndarray(float)

generate_fZ(*Obs*, *TL*, *TK*, *mode*, *hashname*, *koTimes=None*)

Calculates fZ values for all stars over an entire orbit of the sun

Parameters

- **Obs** (*Observatory*) – Observatory class object
- **TL** (*TargetList*) – TargetList class object
- **TK** (*TimeKeeping*) – TimeKeeping object
- **mode** (*dict*) – Selected observing mode
- **hashname** (*str*) – hashname describing the files specific to the current json script
- **koTimes** (*Time(ndarray(float)), optional*) – Absolute MJD mission times from start to end in steps of 1 d

Returns
None

Updates Attributes:

fZMap[n, TL.nStars] (~astropy.units.Quantity(~numpy.ndarray(float))):
Surface brightness of zodiacal light in units of 1/arcsec2 for every star for every ko_dtStep

fZTimes (~astropy.time.Time(~numpy.ndarray(float)), optional):
Absolute MJD mission times from start to end, updated if koTimes does not exist

`global_zodi_min`(*mode*)

This is used to determine the minimum zodi value globally, for the prototype it simply returns the same value that fZ always does

Parameters
mode (*dict*) – Selected observing mode

Returns
The global minimum zodiacal light value for the observing mode, in (1/arcsec**2)

Return type
Quantity

`load_zodi_spatial_data`()

Zodi spatial variation data from Table 17 of [Leinert1998]

Zodiacal Light brightness as function of solar LON (rows) and LAT (columns) Values are given in 10^-8 W m2 sr1 m1 at a wavelength of 500 nm

`zodi_points`

nx2 lat,lon pairs
Type
numpy.ndarry

`zodi_values`

n intensity values (units of W/m2/sr/um)
Type
astropy.units.Quantity

`load_zodi_wavelength_data`()

Zodi wavelength dependence, from Table 19 of [Leinert1998] interpolated w/ a quadratic in log-log space

Creates an interpolant (scipy.interpolate.interp1d) assigned to attribute `logf` which takes as an argument log_10(wavelength in um) and returns log10(specific intensity in W/m^2/um/sr)

`zodi_color_correction_factor`(*lam*, *photon_units=False*)

Compute zodiacal light color correction factor. This is a multiplicative factor to apply to zodiacal light intensity computed at a reference wavelength (500 nm for the Leinert data used in this prototype).

Parameters

- **lam** (*astropy.units.Quantity*) – Wavelength(s) of interest
- **photon_units** (*bool*) – Convert all quantities to photon units before computing ratio. Defaults False (leave all quantities in power units).

Returns
Specific intensity of zodiacal light at requested wavelength(s) scaled by the value at the reference wavelength (500 nm). Has same dimension as input.

Return type
float or numpy.ndarray

zodi_intensity_at_location(*lons*, *lats*, *photon_units=False*)

Compute zodiacal light specific intensity as a function of look vector at reference wavelength (500 nm)

Parameters

- **lons** (*astropy.units.Quantity*) – Ecliptic longitude minus solar ecliptic longitude
- **lats** (*astropy.units.Quantity*) – Ecliptic latitude. Must be of same dimension as lons.
- **photon_units** (*bool*) – Convert all quantities to photon units before computing ratio. Defaults False (leave all quantities in power units).

Returns

Specific intensity of zodiacal light at requested wavelength(s). Has same dimension as input. Default units of W m-2 um-1 sr-1 if `photon_units` is False, otherwise ph s-1 m-2 um-1 sr-1

Return type

astropy.units.Quantity

zodi_intensity_at_wavelength(*lam*, *photon_units=False*)

Compute zodiacal light specific intensity as a function of wavelength

Parameters

- **lam** (*astropy.units.Quantity*) – Wavelength(s) of interest
- **photon_units** (*bool*) – Convert all quantities to photon units before computing ratio. Defaults False (leave all quantities in power units).

Returns

Specific intensity of zodiacal light at requested wavelength(s). Has same dimension as input. Default units of W m-2 um-1 sr-1 if `photon_units` is False, otherwise ph s-1 m-2 um-1 sr-1

Return type

astropy.units.Quantity

zodi_latitudinal_correction_factor(*theta*, *model=None*, *interp_at=135*)

Compute zodiacal light latitudinal correction factor. This is a multiplicative factor to apply to zodiacal light intensity to account for the orientation of the dust disk with respect to the observer.

Parameters

- **theta** (*astropy.units.Quantity*) – Angle of disk. For local zodi, this is equivalent to the absolute value of the ecliptic latitude of the look vector. For exozodi, this is 90 degrees minus the inclination of the orbital plane.
- **model** (*str, optional*) – Model to use. Options are Lindler2006, Stark2014, or interp (case insensitive). See *Zodiacal and Exozodiacal Light* for details. Defaults to None
- **interp_at** (*float*) – If `model` is 'interp', interpolate Leinert Table 17 at this longitude. Defaults to 135.

Returns

Correction factor of zodiacal light at requested angles. Has same dimension as input.

Return type

float or numpy.ndarray

Note: Unlike the color correction factor, this quantity is wavelength independent and thus does not change if using power or photon units.

2.28.1.9 EXOSIMS.SimulatedUniverse package

Submodules

EXOSIMS.SimulatedUniverse.DulzPlavchanUniverse module

class EXOSIMS.SimulatedUniverse.DulzPlavchanUniverse.**DulzPlavchanUniverse**(***specs*)

Bases: *SimulatedUniverse*

Simulated Universe module based on Dulz and Plavchan occurrence rates.

gen_physical_properties(***specs*)

Generating universe based on Dulz and Plavchan occurrence rate tables.

EXOSIMS.SimulatedUniverse.DulzPlavchanUniverseEarthsOnly module

class EXOSIMS.SimulatedUniverse.DulzPlavchanUniverseEarthsOnly.**DulzPlavchanUniverseEarthsOnly**(*earthPF=Tr*
***specs*)

Bases: *SimulatedUniverse*

Simulated Universe module based on Dulz and Plavchan occurrence rates. .. attribute:: earthPF

indicates whether to use Earth's phase function or not

type
boolean

gen_physical_properties(***specs*)

Generating universe based on Dulz and Plavchan occurrence rate tables.

EXOSIMS.SimulatedUniverse.KeplerLikeUniverse module

class EXOSIMS.SimulatedUniverse.KeplerLikeUniverse.**KeplerLikeUniverse**(***specs*)

Bases: *SimulatedUniverse*

Simulated universe implementation inteded to work with the Kepler-like planetary population implementations.

Parameters
specs – user specified values

Notes

The occurrence rate in these universes is set entirely by the radius distribution.

gen_physical_properties(***specs*)

Generates the planetary systems' physical properties. Populates arrays of the orbital elements, albedos, masses and radii of all planets, and generates indices that map from planet to parent star.

All parameters except for albedo and mass are sampled, while those are calculated via the physical model.

EXOSIMS.SimulatedUniverse.KnownRVPlanetsUniverse module

class EXOSIMS.SimulatedUniverse.KnownRVPlanetsUniverse.**KnownRVPlanetsUniverse**(***specs*)

Bases: *SimulatedUniverse*

Simulated universe implementation inteded to work with the Known RV planet planetary population and target list implementations.

Parameters
specs – user specified values

gen_physical_properties(*missionStart=60634*, ***specs*)

Generates the planetary systems' physical properties. Populates arrays of the orbital elements, albedos, masses and radii of all planets, and generates indices that map from planet to parent star.

All parameters are generated by adding consistent error terms to the catalog values for each planet.

EXOSIMS.SimulatedUniverse.SAG13Universe module

class EXOSIMS.SimulatedUniverse.SAG13Universe.**SAG13Universe**(*earthPF=False*, ***specs*)

Bases: *SimulatedUniverse*

Simulated Universe module based on SAG13 Planet Population module.

earthPF

Determines whether to use just Earth's phase function or not

Type
bool

gen_physical_properties(***specs*)

Generating universe based on SAG13 planet radius and period sampling.

All parameters except for albedo and mass are sampled, while those are calculated via the physical model.

EXOSIMS.SimulatedUniverse.SolarSystemUniverse module

class EXOSIMS.SimulatedUniverse.SolarSystemUniverse.**SolarSystemUniverse**(***specs*)

Bases: *SimulatedUniverse*

Simulated Universe with copies of the solar system planets assigned to all stars

gen_physical_properties(***specs*)

Generating copies of the solar system around all targets

gen_solar_system_planet_mass(*nPlans*)

Generated planet masses for each planet :param float: nPlans, the number of planets

Returns
Mp_tiled, the masses of each planet in kg

Return type
ndarray

2.28.1.10 EXOSIMS.StarCatalog package

Submodules

EXOSIMS.StarCatalog.EXOCAT1 module

class EXOSIMS.StarCatalog.EXOCAT1.**EXOCAT1**(*catalogpath=None*, *wdsfilepath=None*, ***specs*)

Bases: *StarCatalog*

EXOCAT Catalog class

This class populates the star catalog used in EXOSIMS from Margaret Turnbull's EXOCAT catalog, retrieved from the NASA Exoplanet Archive as a VOTABLE. Documentation of fields available at: https://exoplanetarchive.ipac.caltech.edu/docs/API_mission_stars.html

Only StarCatalog prototype attributes are used.

EXOSIMS.StarCatalog.FakeCatalog module

class EXOSIMS.StarCatalog.FakeCatalog.**FakeCatalog**(*ntargs=1000*, *star_dist=5*, *ra0=0*, *dec0=0*, ***specs*)

Bases: *StarCatalog*

Fake Catalog class This class generates an artificial target list of stars with a logistic distribution.

Parameters

- **ntargs** (*int*) – Number of targets
- **star_dist** (*float*) – Distance of the stars from observer
- **ra0** (*float*) – Reference right ascension
- **dec0** (*float*) – Reference declination

get_angularDistributions(*f*, *d*, *pos=True*)

Get the distribution of target positions

Parameters

- **f** (*array*) – Distribution function evaluated
- **d** (*float*) – Star distance
- **pos** (*boolean*) – North or south

Returns

array:
Right ascension values

array:
Declination values

array:
Distances of the star

Return type
tuple

inverse_method(*N*, *d*)

Obtain coordinates for the targets from the inverse of a logistic function

Parameters

- **N** (*int*) – Number of targets
- **d** (*float*) – Star distance

Returns
The coordinates for the targets

Return type
SkyCoord module

EXOSIMS.StarCatalog.FakeCatalog_UniformAngles module

class EXOSIMS.StarCatalog.FakeCatalog_UniformAngles.**FakeCatalog_UniformAngles**(*ntargs=20*, *star_dist=5*, ***specs*)

Bases: *StarCatalog*

Fake Catalog of stars separated uniformly by angle

Generate a fake catalog of stars that are uniformly separated.

Parameters

- **ntargs** (*int*) – Sqrt of number of target stars to generate. ntargs by ntargs grid in ra and dec.
- **star_dist** (*float*) – Star distance from observer
- **specs** – Additional parameters passed to StarCatalog parent class

EXOSIMS.StarCatalog.FakeCatalog_UniformSpacing_wInput module

class EXOSIMS.StarCatalog.FakeCatalog_UniformSpacing_wInput.**FakeCatalog_UniformSpacing_wInput**(*lat_sep=0.3*
lon_sep=0.3
star_dist=1(
lat_extra=a
dtype=float(
lon_extra=a
dtype=float(
dist_extra=a
dtype=float(
***specs*)

Bases: *StarCatalog*

EXOSIMS.StarCatalog.GaiaCat1 module

class EXOSIMS.StarCatalog.GaiaCat1.**GaiaCat1**(*catalogfile='Glt15Dlt200DSNgt4-result.fits'*, ***specs*)

Bases: *StarCatalog*

Gaia-derived Catalog class

This class populates the star catalog used in EXOSIMS from a dump of Gaia DR2 data.

Only StarCatalog prototype attributes are used.

EXOSIMS.StarCatalog.HIPfromSimbad module

class EXOSIMS.StarCatalog.HIPfromSimbad.**HIPfromSimbad**(*catalogpath=None*, ***specs*)

Bases: *StarCatalog*

Catalog generator class that uses astroquery to get stellar properties from SIMBAD

Sonny Rappaport, August 2021: Fixed several typos.

EXOSIMS.StarCatalog.HWOMissionStars module

class EXOSIMS.StarCatalog.HWOMissionStars.**HWOMissionStars**(*forceNew=False*, ***specs*)

Bases: *StarCatalog*

HWO Mission Star List. Documentation available at: https://exoplanetarchive.ipac.caltech.edu/docs/2645_NASA_ExEP_Target_List_HWO_Documentation_2023.pdf # noqa: E501

Parameters

forceNew (*bool*) – Run a fresh query even if results exist on disk. Defaults False.

EXOSIMS.StarCatalog.SIMBAD300Catalog module

class EXOSIMS.StarCatalog.SIMBAD300Catalog.**SIMBAD300Catalog**(*cachedir=None*, ***specs*)

Bases: *SIMBADCatalog*

SIMBAD300 Catalog class

This class populates the star catalog used in EXOSIMS from the SIMBAD300 catalog.

EXOSIMS.StarCatalog.SIMBADCatalog module

class EXOSIMS.StarCatalog.SIMBADCatalog.**SIMBADCatalog**(*ntargs=1*, *cachedir=None*, *VmagFill=0.1*, ***specs*)

Bases: *StarCatalog*

SIMBAD Catalog class

This class provides the functions to populate the star catalog used in EXOSIMS from the SIMBAD star catalog data.

SIMBAD_mat2pkl(*matpath*, *pklpath*)

Writes pickled dictionary file from .mat file

This method takes a .mat star catalog, converts it to a Python dictionary, pickles the dictionary, and stores it in the StarCatalog directory.

Parameters

- **matpath** (*string*) – path to .mat file
- **pklpath** (*str*) – pat to .pkl file to be written

Returns
True if successful, False if not

Return type
bool (boolean)

Stores pickled dictionary file with same name as .mat file (and extension of .pkl) containing lists of required values needed to populate the Star Catalog object in StarCatalog directory.

populatepkl(*pklpath*, ***specs*)

Populates the star catalog and returns True if successful

This method populates the star catalog from a pickled dictionary file housed in the StarCatalog directory and returns True if successful.

Parameters
pklpath (*string*) – path to the pickled dictionary file with extension .pkl

Returns
True if successful, False if not

Return type
bool (boolean)

2.28.1.11 EXOSIMS.SurveyEnsemble package

Submodules

EXOSIMS.SurveyEnsemble.IPClusterEnsemble module

class EXOSIMS.SurveyEnsemble.IPClusterEnsemble.**IPClusterEnsemble**(***specs*)

Bases: *SurveyEnsemble*

Parallelized suvey ensemble based on IPython parallel (ipcluster)

run_ensemble(*sim*, *nb_run_sim*, *run_one=None*, *genNewPlanets=True*, *rewindPlanets=True*, *kwargs={}*)

Parameters
sim –

2.28.1.12 EXOSIMS.SurveySimulation package

Submodules

EXOSIMS.SurveySimulation.KnownRVSurvey module

class EXOSIMS.SurveySimulation.KnownRVSurvey.**KnownRVSurvey**(***specs*)

Bases: *SurveySimulation*

Survey Simulation module based on Know RV planets

This class uses estimates of delta magnitude (int_dMag) and instrument working angle (int_WA) for integration time calculation, specific to the known RV planets.

Parameters
****specs** – user specified values

EXOSIMS.SurveySimulation.SLSQPScheduler module

class EXOSIMS.SurveySimulation.SLSQPScheduler.**SLSQPScheduler**(*cacheOptTimes=False*, *staticOptTimes=False*, *selectionMetric='maxC'*, *Izod='current'*, *maxiter=60*, *ftol=0.001*, ***specs*)

Bases: *SurveySimulation*

This class implements a continuous optimization of integration times using the scipy minimize function with method SLSQP. ortools with the CBC linear solver is used to find an initial solution consistent with the constraints. For details see Keithly et al. 2019. Alternatively: Savransky et al. 2017 (SPIE).

Parameters
****specs** – user specified values

Notes

Due to the time costs of the current comp_per_inttime calculation in GarrettCompleteness this should be used with BrownCompleteness.

Requires ortools

arbitrary_time_advancement(*dt*)

Handles fully dynamically scheduled case where OBduration is infinite and missionPortion is less than 1. Input dt is the total amount of time, including all overheads and extras used for the previous observation.

calc_targ_intTime(*sInds*, *startTimes*, *mode*)

Given a subset of targets, calculate their integration times given the start of observation time.

This implementation updates the optimized times based on current conditions and mission time left.

Note: next_target filter will discard targets with zero integration times.

Parameters

- **sInds** (*integer array*) – Indices of available targets
- **startTimes** (*astropy quantity array*) – absolute start times of observations. must be of the same size as sInds

- **mode** (*dict*) – Selected observing mode for detection

Returns
Integration times for detection. Same dimension as sInds

Return type
astropy Quantity array

choose_next_target(*old_sInd*, *sInds*, *slewTimes*, *intTimes*)

Given a subset of targets (pre-filtered by method next_target or some other means), select the best next one.

Parameters

- **old_sInd** (*integer*) – Index of the previous target star
- **sInds** (*integer array*) – Indices of available targets
- **slewTimes** (*astropy quantity array*) – slew times to all stars (must be indexed by sInds)
- **intTimes** (*astropy Quantity array*) – Integration times for detection in units of day

Returns

sInd (integer):
Index of next target star

waitTime (astropy Quantity):
the amount of time to wait (this method returns None)

Return type
tuple

inttimesfeps(*eps*, *Cb*, *Csp*)

Compute the optimal subset of targets for a given epsilon value where epsilon is the maximum completeness gradient.

Everything is in units of days

objfun(*t*, *sInds*, *fZ*)

Objective Function for SLSQP minimization. Purpose is to maximize summed completeness

Parameters

- **t** (*ndarray*) – Integration times in days. NB: NOT an astropy quantity.
- **sInds** (*ndarray*) – Target star indices (of same size as t)
- **fZ** (*astropy Quantity*) – Surface brightness of local zodiacal light in units of 1/arcsec2 Same size as t

objfun_deriv(*t*, *sInds*, *fZ*)

Jacobian of objective Function for SLSQP minimization.

Parameters

- **t** (*astropy Quantity*) – Integration times in days. NB: NOT an astropy quantity.
- **sInds** (*ndarray*) – Target star indices (of same size as t)
- **fZ** (*astropy Quantity*) – Surface brightness of local zodiacal light in units of 1/arcsec2 Same size as t

whichTimeComesNext(*absTs*)

Determine which absolute time comes next from current time Specifically designed to determine when the next local zodiacal light event occurs form fZQuads

Parameters
absTs (*list*) – the absolute times of different events (list of absolute times)

Returns
the absolute time which occurs next

Return type
astropy time quantity

EXOSIMS.SurveySimulation.cbytScheduler module

class EXOSIMS.SurveySimulation.cbytScheduler.**cbytScheduler**(***specs*)

Bases: *SurveySimulation*

cbytScheduler - Completeness-by-time Scheduler

This class implements a Scheduler that selects the current highest Completeness/Integration Time.

Parameters
****specs** – user specified values

choose_next_target(*old_sInd*, *sInds*, *slewTimes*, *intTimes*)

Choose next target based on truncated depth first search of linear cost function.

Parameters

- **old_sInd** (*int*) – Index of the previous target star
- **sInds** (*int numpy.ndarray*) – Indices of available targets
- **slewTimes** (*astropy.units.Quantity numpy.ndarray*) – slew times to all stars (must be indexed by sInds)
- **intTimes** (*astropy.units.Quantity numpy.ndarray*) – Integration times for detection in units of day

Returns

sInd (int):
Index of next target star

waitTime (astropy.units.Quantity or None):
the amount of time to wait (this method returns None)

Return type
tuple

EXOSIMS.SurveySimulation.coroOnlyScheduler module

class EXOSIMS.SurveySimulation.coroOnlyScheduler.**coroOnlyScheduler**(*revisit_wait=0.5*, *revisit_weight=1.0*, *n_det_remove=3*, *n_det_min=3*, *max_successful_chars=1*, *max_successful_dets=4*, *lum_exp=1*, *promote_by_time=False*, *detMargin=0.0*, ***specs*)

Bases: *SurveySimulation*

coroOnlyScheduler - Coronograph Only Scheduler

This scheduler inherits directly from the prototype SurveySimulation module.

The coronlyScheduler operates using only a coronograph. The scheduler makes detections until stars can be promoted into a characterization list, at which point they are charcterized.

Parameters

- **revisit_wait** (*float*) – Wait time threshold for star revisits. The value given is the fraction of a characterized planet's period that must be waited before scheduling a revisit.
- **revisit_weight** (*float*) – Weight used to increase preference for coronograph revisits.
- **n_det_remove** (*integer*) – Number of failed detections before a star is removed from the target list.
- **n_det_min** (*integer*) – Minimum number of detections required for promotion to char target.
- **max_successful_chars** (*integer*) – Maximum number of successful characterizions before star is taken off target list.
- **max_successful_dets** (*integer*) – Maximum number of successful detections before star is taken off target list.
- **lum_exp** (*int*) – The exponent to use for luminosity weighting on coronograph targets.
- **promote_by_time** (*bool*) – Only promote stars that have had detections that span longer than half a period.
- **detMargin** (*float*) – Acts in the same way a charMargin. Adds a multiplyer to the calculated detection time.
- ****specs** – user specified values

choose_next_target(*old_sInd*, *sInds*, *slewTimes*, *t_dets*)

Choose next telescope target based on star completeness and integration time.

Parameters

- **old_sInd** (*integer*) – Index of the previous target star
- **sInds** (*integer array*) – Indices of available targets
- **t_dets** (*astropy Quantity array*) – Integration times for detection in units of day

Returns

Index of next target star

Return type
sInd (integer)

`initializeStorageArrays`()

Initialize all storage arrays based on # of stars and targets

`next_target`(*old_sInd*, *det_modes*, *char_modes*)

Finds index of next target star and calculates its integration time.

This method chooses the next target star index based on which stars are available, their integration time, and maximum completeness. Returns None if no target could be found.

Parameters

- **old_sInd** (*integer*) – Index of the previous target star
- **mode** (*dict*) – Selected observing mode for detection

Returns

DRM (dict):
Design Reference Mission, contains the results of one complete observation (detection and characterization)

sInd (integer):
Index of next target star. Defaults to None.

intTime (astropy Quantity):
Selected star integration time for detection in units of day. Defaults to None.

waitTime (astropy Quantity):
a strategically advantageous amount of time to wait in the case of an occulter for slew times

Return type
tuple

`observation_characterization`(*sInd*, *mode*, *mode_index*)

Finds if characterizations are possible and relevant information

Parameters

- **sInd** (*integer*) – Integer index of the star of interest
- **mode** (*dict*) – Selected observing mode for characterization

Returns

Characterization status for each planet orbiting the observed
target star including False Alarm if any, where 1 is full spectrum, -1 partial spectrum, and 0 not characterized

fZ (astropy Quantity):
Surface brightness of local zodiacal light in units of 1/arcsec2

systemParams (dict):
Dictionary of time-dependant planet properties averaged over the duration of the integration

SNR (float ndarray):
Characterization signal-to-noise ratio of the observable planets. Defaults to None.

intTime (astropy Quantity):
Selected star characterization time in units of day. Defaults to None.

Return type
characterized (integer list)

revisitFilter(*sInds*, *tmpCurrentTimeNorm*)

Helper method for Overloading Revisit Filtering

Parameters

- **list** (*sInds - indices of stars still in observation*) –
- **tmpCurrentTimeNorm** (*MJD*) –
- **form** (*added in MJD*) –

Returns
sInds - indices of stars still in observation list

Return type
ndarray(int)

run_sim()

Performs the survey simulation

scheduleRevisit(*sInd*, *smin*, *det*, *pInds*)

A Helper Method for scheduling revisits after observation detection :param sInd - sInd of the star just detected: :param smin - minimum separation of the planet to star of planet just detected: :param det -: :param pInds - Indices of planets around target star:

Returns
updates self.starRevisit attribute

test_observation_characterization(*sInd*, *mode*, *mode_index*)

Finds if characterizations are possible and relevant information

Parameters

- **sInd** (*integer*) – Integer index of the star of interest
- **mode** (*dict*) – Selected observing mode for characterization

Returns

Characterization status for each planet orbiting the observed
target star including False Alarm if any, where 1 is full spectrum, -1 partial spectrum, and 0 not characterized

fZ (astropy Quantity):
Surface brightness of local zodiacal light in units of 1/arcsec2

systemParams (dict):
Dictionary of time-dependant planet properties averaged over the duration of the integration

SNR (float ndarray):
Characterization signal-to-noise ratio of the observable planets. Defaults to None.

intTime (astropy Quantity):
Selected star characterization time in units of day. Defaults to None.

Return type
characterized (integer list)

EXOSIMS.SurveySimulation.linearJScheduler module

class EXOSIMS.SurveySimulation.linearJScheduler.**linearJScheduler**(*coeffs=[1, 1, 1, 1, 2, 1]*, *revisit_wait=91.25*, *find_known_RV=False*, ***specs*)

Bases: *SurveySimulation*

This class implements the linear cost function scheduler described in [Savransky2010]

Parameters

- **coeffs** (*iterable 6x1*) – Cost function coefficients: slew distance, completeness, least visited planet ramp, unvisited known RV planet ramp, least visited ramp, unvisited ramp
- **revisit_wait** (*float*) – The time required for the scheduler to wait before a target may be revisited
- **find_known_RV** (*boolean*) – A flag that turns on the ability to identify known RV stars. The stars with known rocky planets have their int_comp value set to 1.0.
- **specs** (*dict*) – *Input Specification*

choose_next_target(*old_sInd*, *sInds*, *slewTimes*, *intTimes*)

Choose next target based on truncated depth first search of linear cost function.

Parameters

- **old_sInd** (*integer*) – Index of the previous target star
- **sInds** (*integer array*) – Indices of available targets
- **slewTimes** (*astropy quantity array*) – slew times to all stars (must be indexed by sInds)
- **intTimes** (*astropy Quantity array*) – Integration times for detection in units of day

Returns

sInd (int):
Index of next target star

waitTime (astropy.units.Quantity or None):
the amount of time to wait (this method returns None)

Return type
tuple

next_target(*old_sInd*, *mode*)

Finds index of next target star and calculates its integration time.

This method chooses the next target star index based on which stars are available, their integration time, and maximum completeness. Returns None if no target could be found.

Parameters

- **old_sInd** (*integer*) – Index of the previous target star
- **mode** (*dict*) – Selected observing mode for detection

Returns

Design Reference Mission, contains the results of one complete
observation (detection and characterization)

sInd (integer):
Index of next target star. Defaults to None.

intTime (astropy Quantity):
Selected star integration time for detection in units of day. Defaults to None.

waitTime (astropy Quantity):
a strategically advantageous amount of time to wait in the case of an occulter for slew times

Return type
DRM (dict)

`observation_characterization`(*sInd*, *mode*)

Finds if characterizations are possible and relevant information

Parameters

- **sInd** (*int*) – Integer index of the star of interest
- **mode** (*dict*) – Selected observing mode for characterization

Returns

characterized (integer list):
Characterization status for each planet orbiting the observed target star including False Alarm if any, where 1 is full spectrum, -1 partial spectrum, and 0 not characterized

fZ (astropy Quantity):
Surface brightness of local zodiacal light in units of 1/arcsec2

systemParams (dict):
Dictionary of time-dependant planet properties averaged over the duration of the integration

SNR (float ndarray):
Characterization signal-to-noise ratio of the observable planets. Defaults to None.

intTime (astropy Quantity):
Selected star characterization time in units of day. Defaults to None.

Return type
tuple

`revisitFilter`(*sInds*, *tmpCurrentTimeNorm*)

Helper method for Overloading Revisit Filtering

Parameters

- **sInds** (*int numpy.ndarray*) – indices of stars still in observation list
- **tmpCurrentTimeNorm** (*MJD*) – the simulation time after overhead was added in MJD

Returns
sInds - indices of stars still in observation list

Return type
int numpy.ndarray

`scheduleRevisit`(*sInd*, *smin*, *det*, *pInds*)

A Helper Method for scheduling revisits after observation detection

Parameters

- **sInd** (*int*) – sInd of the star just detected
- **smin** (*float*) – minimum separation of the planet to star of planet just detected

- **det** (*bool numpy.ndarray*) – detection array
- **pInds** (*int numpy.ndarray*) – Indices of planets around target star

Returns
None

updates self.starRevisit attribute

EXOSIMS.SurveySimulation.linearJScheduler_3DDPC module

class EXOSIMS.SurveySimulation.linearJScheduler_3DDPC.**linearJScheduler_3DDPC**(***specs*)

Bases: *linearJScheduler_DDPC*

linearJScheduler_3DDPC - linearJScheduler 3 Dual Detection Parallel Charachterization

This scheduler inherits from the LJS_DDPC, but is capable of taking in six detection modes and six characterization modes. Detections can then be performed using a dual-band mode that is selected from the best available mode-pair, while characterizations are performed in parallel.

next_target(*old_sInd*, *modes*)

Finds index of next target star and calculates its integration time.

This method chooses the next target star index based on which stars are available, their integration time, and maximum completeness. Returns None if no target could be found.

Parameters

- **old_sInd** (*integer*) – Index of the previous target star
- **mode** (*dict*) – Selected observing modes for detection

Returns

Design Reference Mission, contains the results of one complete
observation (detection and characterization)

sInd (integer):
Index of next target star. Defaults to None.

intTime (astropy Quantity):
Selected star integration time for detection in units of day. Defaults to None.

waitTime (astropy Quantity):
a strategically advantageous amount of time to wait in the case of an occulter for slew times

det_mode (dict):
Selected detection mode

Return type
DRM (dict)

run_sim()

Performs the survey simulation

EXOSIMS.SurveySimulation.linearJScheduler_DDPC module

class EXOSIMS.SurveySimulation.linearJScheduler_DDPC.**linearJScheduler_DDPC**(*revisit_weight=1.0*, ***specs*)

Bases: *linearJScheduler*

linearJScheduler_DDPC - linearJScheduler Dual Detection Parallel Charachterization

This scheduler inherits from the LJS, but is capable of taking in two detection modes and two chracterization modes. Detections can then be performed using a dual-band mode, while characterizations are performed in parallel.

choose_next_target(*old_sInd*, *sInds*, *slewTimes*, *intTimes*)

Choose next target based on truncated depth first search of linear cost function.

Parameters

- **old_sInd** (*integer*) – Index of the previous target star
- **sInds** (*integer array*) – Indices of available targets
- **slewTimes** (*astropy quantity array*) – slew times to all stars (must be indexed by sInds)
- **intTimes** (*astropy Quantity array*) – Integration times for detection in units of day

Returns
Index of next target star

Return type
sInd (integer)

next_target(*old_sInd*, *modes*)

Finds index of next target star and calculates its integration time.

This method chooses the next target star index based on which stars are available, their integration time, and maximum completeness. Returns None if no target could be found.

Parameters

- **old_sInd** (*integer*) – Index of the previous target star
- **modes** (*dict*) – Selected observing modes for detection

Returns

DRM (dict):
Design Reference Mission, contains the results of one complete observation (detection and characterization)

sInd (integer):
Index of next target star. Defaults to None.

intTime (astropy Quantity):
Selected star integration time for detection in units of day. Defaults to None.

waitTime (astropy Quantity):
a strategically advantageous amount of time to wait in the case of an occulter for slew times

det_mode (dict):
Selected detection mode

Return type
tuple

observation_characterization(*sInd*, *modes*)

Finds if characterizations are possible and relevant information

Parameters

- **sInd** (*integer*) – Integer index of the star of interest
- **modes** (*dict*) – Selected observing modes for characterization

Returns

Characterization status for each planet orbiting the observed
target star including False Alarm if any, where 1 is full spectrum, -1 partial spectrum, and 0 not characterized

fZ (astropy Quantity):
Surface brightness of local zodiacal light in units of 1/arcsec2

systemParams (dict):
Dictionary of time-dependant planet properties averaged over the duration of the integration

SNR (float ndarray):
Characterization signal-to-noise ratio of the observable planets. Defaults to None.

intTime (astropy Quantity):
Selected star characterization time in units of day. Defaults to None.

Return type
characterized (integer list)

run_sim()

Performs the survey simulation

EXOSIMS.SurveySimulation.linearJScheduler_det_only module

class EXOSIMS.SurveySimulation.linearJScheduler_det_only.**linearJScheduler_det_only**(***specs*)

Bases: *linearJScheduler*

linearJScheduler_det_only - linearJScheduler Detections Only

This class implements the linear cost function scheduler described in Savransky et al. (2010).

This scheduler inherits from the linearJScheduler module but performs only detections.

Parameters
specs – user specified values

choose_next_target(*old_sInd*, *sInds*, *slewTimes*, *intTimes*)

Choose next target based on truncated depth first search of linear cost function.

Parameters

- **old_sInd** (*integer*) – Index of the previous target star
- **sInds** (*integer array*) – Indices of available targets
- **slewTimes** (*astropy quantity array*) – slew times to all stars (must be indexed by sInds)
- **intTimes** (*astropy Quantity array*) – Integration times for detection in units of day

Returns
Index of next target star

Return type
sInd (integer)

next_target(*old_sInd*, *mode*)

Finds index of next target star and calculates its integration time.

This method chooses the next target star index based on which stars are available, their integration time, and maximum completeness. Returns None if no target could be found.

Parameters

- **old_sInd** (*integer*) – Index of the previous target star
- **mode** (*dict*) – Selected observing mode for detection

Returns

DRM (dict):
Design Reference Mission, contains the results of one complete observation (detection and characterization)

sInd (integer):
Index of next target star. Defaults to None.

intTime (astropy Quantity):
Selected star integration time for detection in units of day. Defaults to None.

waitTime (astropy Quantity):
a strategically advantageous amount of time to wait in the case of an occulter for slew times

Return type
tuple

revisitFilter(*sInds*, *tmpCurrentTimeNorm*)

Helper method for Overloading Revisit Filtering

Parameters

- **list** (*sInds - indices of stars still in observation*) –
- **tmpCurrentTimeNorm** (*MJD*) –
- **form** (*was added in MJD*) –

Returns
sInds - indices of stars still in observation list

run_sim()

Performs the survey simulation

scheduleRevisit(*sInd*, *smin*, *det*, *pInds*)

A Helper Method for scheduling revisits after observation detection :param sInd - sInd of the star just detected: :param smin - minimum separation of the planet to star of planet just detected: :param det -: :param pInds - Indices of planets around target star:

Returns
updates self.starRevisit attribute

EXOSIMS.SurveySimulation.linearJScheduler_orbitChar module

class EXOSIMS.SurveySimulation.linearJScheduler_orbitChar.**linearJScheduler_orbitChar**(*coeffs=[1, 1, 1, 1, 2, 1]*, *revisit_wait=0.5*, *n_det_remove=3*, *n_det_min=3*, *max_successful_dets=4*, *max_successful_chars=*, *det_only=False*, *char_only=False*, ***specs*)

Bases: *SurveySimulation*

This class implements a varient of the linear cost function scheduler described in Savransky et al. (2010).

It inherits directly from the protoype SurveySimulation class.

The LJS_orbitChar scheduler performs scheduled starshade visits to both detect and characterize targets. Once a target is detected, it will be subsequently characterized. If the characterization is successful, that taget will be marked for further detections to characeterize it's orbit.

Parameters

- **coeffs** (*iterable 6x1*) – Cost function coefficients: slew distance, completeness, least visited known RV planet ramp, unvisited known RV planet ramp, least visited ramp, unvisited ramp
- **revisit_wait** (*float*) – Wait time threshold for star revisits. The value given is the fraction of a characterized planet's period that must be waited before scheduling a revisit.
- **n_det_remove** (*int*) – Number of failed detections before a star is removed from the target list.
- **n_det_min** (*int*) – Minimum number of detections required for promotion to char target.
- **max_successful_dets** (*int*) – Maximum number of successful detections before star is taken off target list.
- **max_successful_chars** (*int*) – Maximum number of successful characterizations on a given star before it is removed from the target list.
- **det_only** (*bool*) – Run the sim only performing detections and no chars.
- **(bool** (*char_only*) – Run the sim performing only chars, particularly for precursor RV using known_rocky.
- **specs** (*dict*) – *Input Specification*

choose_next_target(*old_sInd*, *sInds*, *slewTimes*, *intTimes*)

Choose next target based on truncated depth first search of linear cost function.

Parameters

- **old_sInd** (*int*) – Index of the previous target star
- **sInds** (*int array*) – Indices of available targets

- **slewTimes** (*astropy quantity array*) – slew times to all stars (must be indexed by sInds)
- **intTimes** (*astropy.units.Quantity array*) – Integration times for detection in units of day

Returns

sInd (int):
Index of next target star

waitTime (astropy.units.Quantity):
the amount of time to wait (this method returns None)

Return type
tuple

next_target(*old_sInd*, *mode*, *char_mode*)

Finds index of next target star and calculates its integration time.

This method chooses the next target star index based on which stars are available, their integration time, and maximum completeness. Returns None if no target could be found.

Parameters

- **old_sInd** (*int*) – Index of the previous target star
- **mode** (*dict*) – Selected observing mode for detection

Returns

DRM (dict):
Design Reference Mission, contains the results of one complete observation (detection and characterization)

sInd (int):
Index of next target star. Defaults to None.

intTime (astropy.units.Quantity):
Selected star integration time for detection in units of day. Defaults to None.

waitTime (astropy.units.Quantity):
a strategically advantageous amount of time to wait in the case of an occulter for slew times

Return type
tuple

observation_characterization(*sInd*, *mode*)

Finds if characterizations are possible and relevant information

Parameters

- **sInd** (*int*) – Integer index of the star of interest
- **mode** (*dict*) – Selected observing mode for characterization

Returns

characterized (int list):
Characterization status for each planet orbiting the observed target star including False Alarm if any, where 1 is full spectrum, -1 partial spectrum, and 0 not characterized

fZ (astropy.units.Quantity):
Surface brightness of local zodiacal light in units of 1/arcsec2

systemParams (dict):
Dictionary of time-dependant planet properties averaged over the duration of the integration

SNR (float numpy.ndarray):
Characterization signal-to-noise ratio of the observable planets. Defaults to None.

intTime (astropy.units.Quantity):
Selected star characterization time in units of day. Defaults to None.

Return type
tuple

revisitFilter(*sInds*, *tmpCurrentTimeNorm*)

Helper method for Overloading Revisit Filtering

Parameters

- **list** (*sInds - indices of stars still in observation*) –
- **tmpCurrentTimeNorm** (*MJD*) –
- **form** (*in MJD*) –

Returns
sInds - indices of stars still in observation list

run_sim()

Performs the survey simulation

scheduleRevisit(*sInd*, *smin*, *det*, *pInds*)

A Helper Method for scheduling revisits after observation detection

Parameters

- **detected** (*smin - minimum separation of the planet to star of planet just*) –
- **detected** –
- **-** (*det*) –
- **star** (*pInds - Indices of planets around target*) –

Returns
None

updates self.starRevisit attribute

EXOSIMS.SurveySimulation.occulterJScheduler module

class EXOSIMS.SurveySimulation.occulterJScheduler.**occulterJScheduler**(*nSteps=1*, *useAngles=False*, ***specs*)

Bases: *linearJScheduler*

This class inherits linearJScheduler and works best when paired with the SotoStarshade Observatory class.

Parameters

- **nSteps** (*integer 1x1*) – Number of steps to take when calculating the cost function.
- **useAngles** (*bool*) – Use interpolated dV angles.

- ****specs** – user specified values

choose_next_target(*old_sInd*, *sInds*, *slewTimes*, *intTimes*)

Helper method for method next_target to simplify alternative implementations.

Given a subset of targets (pre-filtered by method next_target or some other means), select the best next one. The prototype uses completeness as the sole heuristic.

Parameters

- **old_sInd** (*integer*) – Index of the previous target star
- **sInds** (*integer array*) – Indices of available targets
- **slewTimes** (*astropy quantity array*) – slew times to all stars (must be indexed by sInds)
- **intTimes** (*astropy Quantity array*) – Integration times for detection in units of day

Returns

sInd (integer):
Index of next target star

waitTime (astropy Quantity):
some strategic amount of time to wait in case an occulter slew is desired (default is None)

Return type
tuple

EXOSIMS.SurveySimulation.randomWalkScheduler module

class EXOSIMS.SurveySimulation.randomWalkScheduler.**randomWalkScheduler**(*scriptfile=None*, *ntFlux=1*, *nVisitsMax=5*, *charMargin=0.15*, *dt_max=1.0*, *record_counts_path=None*, *nokoMap=False*, *nofZ=False*, *cachedir=None*, *defaultAddExoplanetObsTime=True*, *find_known_RV=False*, *include_known_RV=None*, *make_debug_bird_plots=False*, *debug_plot_path=None*, ***specs*)

Bases: *SurveySimulation*

This class implements a random walk scheduler that selects the next target at random from the pool of currently available targets.

This is useful for mapping out the space of possible mission outcomes for a fixed population of planets in order to validate other schedulers.

choose_next_target(*old_sInd*, *sInds*, *slewTimes*, *intTimes*)

Choose next target at random

Parameters

- **old_sInd** (*int*) – Index of the previous target star
- **sInds** (*int array*) – Indices of available targets
- **slewTimes** (*astropy quantity array*) – slew times to all stars (must be indexed by sInds)
- **intTimes** (*astropy Quantity array*) – Integration times for detection in units of day

Returns

sInd (int):
Index of next target star

waitTime (astropy Quantity):
the amount of time to wait (this method returns None)

Return type
tuple

EXOSIMS.SurveySimulation.randomWalkScheduler2 module

class EXOSIMS.SurveySimulation.randomWalkScheduler2.**randomWalkScheduler2**(*occHIPs=[]*, ***specs*)

Bases: *SurveySimulation*

This class implements a random walk scheduler that selects the next target at random from the pool of currently available targets.

This is useful for mapping out the space of possible mission outcomes for a fixed population of planets in order to validate other schedulers.

The random walk will attempt to first choose from occulter targets before defaulting back to the general target list.

Parameters

- **occHIPs** (*iterable nx1*) – List of star HIP numbers to initialize occulter target list.
- ****specs** – user specified values

choose_next_target(*old_sInd*, *sInds*, *slewTimes*, *intTimes*)

Choose next target at random

Parameters

- **old_sInd** (*integer*) – Index of the previous target star
- **sInds** (*integer array*) – Indices of available targets
- **slewTimes** (*astropy quantity array*) – slew times to all stars (must be indexed by sInds)
- **intTimes** (*astropy Quantity array*) – Integration times for detection in units of day

Returns
Index of next target star

Return type
sInd (integer)

EXOSIMS.SurveySimulation.tieredScheduler module

class EXOSIMS.SurveySimulation.tieredScheduler.**tieredScheduler**(*coeffs=[2, 1, 1, 8, 4, 1, 1]*, *occHIPs=[]*, *topstars=0*, *revisit_wait=0.5*, *revisit_weight=1.0*, *GAPortion=0.25*, *int_inflection=False*, *GA_simult_det_fraction=0.07*, *promote_hz_stars=False*, *phase1_end=365*, *n_det_remove=3*, *n_det_min=3*, *occ_max_visits=3*, *max_successful_chars=1*, *max_successful_dets=4*, *nmax_promo_det=4*, *lum_exp=1*, *tot_det_int_cutoff=None*, ***specs*)

Bases: *SurveySimulation*

This class implements a tiered scheduler that independently schedules the observatory while the starshade slews to its next target.

Parameters

- **coeffs** (*iterable 7x1*) – Cost function coefficients: slew distance, completeness, intTime, deep-dive least visited ramp, deep-dive unvisited ramp, unvisited ramp, and least-visited ramp
- **occHIPs** (*iterable nx1*) – List of star HIP numbers to initialize occulter target list.
- **topstars** (*integer*) – Number of HIP numbers to recieve preferential treatment.
- **revisit_wait** (*float*) – Wait time threshold for star revisits. The value given is the fraction of a characterized planet's period that must be waited before scheduling a revisit.
- **revisit_weight** (*float*) – Weight used to increase preference for coronograph revisits.
- **GAPortion** (*float*) – Portion of mission time used for general astrophysics.
- **int_inflection** (*boolean*) – Calculate integration time using the pre-calculated integration time curves. Default is False.
- **GA_simult_det_fraction** (*float*) – Fraction of detection time to be considered as GA time.
- **promote_hz_stars** (*boolean*) – Flag that allows promotion of targets with planets in the habitable zone to the occulter target list.
- **phase1_end** (*int*) – Number of days to wait before the end of phase 1, when phase 1 ends, target promotion begins.
- **n_det_remove** (*int*) – Minimum number of visits with no detections required to filter off star
- **n_det_min** (*int*) – Minimum number of detections required for promotion

- **occ_max_visits** (*int*) – Number of maximum visits to a star allowed by the occulter.
- **max_successful_chars** (*int*) – Maximum number of successful characterizations on a given star before it is removed from the target list.
- **max_successful_dets** (*int*) – Maximum number of successful detections on a given star before it is removed from the target list.
- **nmax_promo_det** (*int*) – Number of detection on a star required to be promoted regardless of detection occurance times.
- **lum_exp** (*int*) – Exponent used in the luminosity weighting function.
- **tot_det_int_cutoff** (*float*) – Number of total days the scheduler is allowed to spend on detections.
- ****specs** – user specified values

calc_int_inflection(*t_sInds*, *fEZ*, *startTime*, *WA*, *mode*, *ischar=False*)

Calculate integration time based on inflection point of Completeness as a function of int_time

Parameters

- **t_sInds** (*integer array*) – Indices of the target stars
- **fEZ** (*astropy Quantity array*) – Surface brightness of exo-zodiacal light in units of 1/arcsec2
- **startTime** (*astropy Quantity array*) – Surface brightness of local zodiacal light in units of 1/arcsec2
- **WA** (*astropy Quantity*) – Working angle of the planet of interest in units of arcsec
- **mode** (*dict*) – Selected observing mode

Returns
The suggested integration time

Return type
int_times (astropy quantity array)

choose_next_occulter_target(*old_occ_sInd*, *occ_sInds*, *intTimes*)

Choose next target for the occulter based on truncated depth first search of linear cost function.

Parameters

- **old_occ_sInd** (*integer*) – Index of the previous target star
- **occ_sInds** (*integer array*) – Indices of available targets
- **intTimes** (*astropy Quantity array*) – Integration times for detection in units of day

Returns
Index of next target star

Return type
sInd (int)

choose_next_telescope_target(*old_sInd*, *sInds*, *t_dets*)

Choose next telescope target based on star completeness and integration time.

Parameters

- **old_sInd** (*integer*) – Index of the previous target star
- **sInds** (*integer array*) – Indices of available targets

- **t_dets** (*astropy Quantity array*) – Integration times for detection in units of day

Returns
Index of next target star

Return type
sInd (integer)

next_target(*old_sInd*, *old_occ_sInd*, *det_mode*, *char_mode*)

Finds index of next target star and calculates its integration time.

This method chooses the next target star index based on which stars are available, their integration time, and maximum completeness. Returns None if no target could be found.

Parameters

- **old_sInd** (*integer*) – Index of the previous target star for the telescope
- **old_occ_sInd** (*integer*) – Index of the previous target star for the occulter
- **det_mode** (*dict*) – Selected observing mode for detection
- **char_mode** (*dict*) – Selected observing mode for characterization

Returns

DRM (dicts):
Contains the results of survey simulation

sInd (integer):
Index of next target star. Defaults to None.

occ_sInd (integer):
Index of next occulter target star. Defaults to None.

t_det (astropy Quantity):
Selected star integration time for detection in units of day. Defaults to None.

Return type
tuple

observation_characterization(*sInd*, *mode*)

Finds if characterizations are possible and relevant information

Parameters

- **sInd** (*integer*) – Integer index of the star of interest
- **mode** (*dict*) – Selected observing mode for characterization

Returns

Characterization status for each planet orbiting the observed
target star including False Alarm if any, where 1 is full spectrum, -1 partial spectrum, and 0 not characterized

fZ (astropy Quantity):
Surface brightness of local zodiacal light in units of 1/arcsec2

systemParams (dict):
Dictionary of time-dependant planet properties averaged over the duration of the integration

SNR (float ndarray):
Characterization signal-to-noise ratio of the observable planets. Defaults to None.

intTime (astropy Quantity):
Selected star characterization time in units of day. Defaults to None.

Return type
characterized (integer list)

`promote_coro_targets`(*occ_sInds*, *sInds*)

Determines which coronograph targets to promote to occulter targets

Parameters

- **occ_sInds** (*numpy array*) – occulter targets
- **sInds** (*numpy array*) – coronograph targets

Returns
updated occulter targets

Return type
occ_sInds (numpy array)

`revisitFilter`(*sInds*, *tmpCurrentTimeNorm*)

Helper method for Overloading Revisit Filtering

Parameters

- **list** (*sInds - indices of stars still in observation*) –
- **tmpCurrentTimeNorm** (*MJD*) –
- **form** (*in MJD*) –

Returns
sInds - indices of stars still in observation list

`run_sim`()

Performs the survey simulation

Returns
Message printed at the end of a survey simulation.

Return type
mission_end (string)

`scheduleRevisit`(*sInd*, *smin*, *det*, *pInds*)

A Helper Method for scheduling revisits after observation detection

Parameters

- **detected** (*smin - minimum separation of the planet to star of planet just*) –
- **detected** –
- **-** (*det*) –
- **star** (*pInds - Indices of planets around target*) –

Returns
updates self.starRevisit attribute

EXOSIMS.SurveySimulation.tieredScheduler_DD module

class EXOSIMS.SurveySimulation.tieredScheduler_DD.**tieredScheduler_DD**(***specs*)

Bases: *tieredScheduler*

tieredScheduler_DD - tieredScheduler Dual Detection

This class implements a version of the tieredScheduler that performs dual-band detections

next_target(*old_sInd*, *old_occ_sInd*, *det_modes*, *char_mode*)

Finds index of next target star and calculates its integration time.

This method chooses the next target star index based on which stars are available, their integration time, and maximum completeness. Returns None if no target could be found.

Parameters

- **old_sInd** (*integer*) – Index of the previous target star for the telescope
- **old_occ_sInd** (*integer*) – Index of the previous target star for the occulter
- **det_modes** (*dict array*) – Selected observing mode for detection
- **char_mode** (*dict*) – Selected observing mode for characterization

Returns

DRM (dicts):
Contains the results of survey simulation

sInd (integer):
Index of next target star. Defaults to None.

occ_sInd (integer):
Index of next occulter target star. Defaults to None.

t_det (astropy Quantity):
Selected star integration time for detection in units of day. Defaults to None.

Return type
tuple

run_sim()

Performs the survey simulation

Returns
Message printed at the end of a survey simulation.

Return type
mission_end (string)

2.28.1.13 EXOSIMS.TargetList package

Submodules

EXOSIMS.TargetList.EclipticTargetList module

class EXOSIMS.TargetList.EclipticTargetList.**EclipticTargetList**(***specs*)

Bases: *TargetList*

Target list in which star positions may be obtained in heliocentric equatorial or ecliptic coordinates.

Parameters
****specs** – user specified values

nan_filter()

Populates Target List and filters out values which are nan

revise_lists(*sInds*)

Replaces Target List catalog attributes with filtered values, and updates the number of target stars.

Parameters
sInds (*integer ndarray*) – Integer indices of the stars of interest

starprop(*sInds*, *currentTime*, *eclip=False*)

Finds target star positions vector in heliocentric equatorial (default) or ecliptic frame for current time (MJD).

This method uses ICRS coordinates which is approximately the same as equatorial coordinates.

Parameters

- **sInds** (*integer ndarray*) – Integer indices of the stars of interest
- **currentTime** (*astropy Time*) – Current absolute mission time in MJD
- **eclip** (*boolean*) – Boolean used to switch to heliocentric ecliptic frame. Defaults to False, corresponding to heliocentric equatorial frame.

Returns
Target star positions vector in heliocentric equatorial (default) or ecliptic frame in units of pc. Will return an m x n x 3 array where m is size of currentTime, n is size of sInds. If either m or n is 1, will return n x 3 or m x 3.

Return type
r_targ (astropy Quantity array)

Note: Use eclip=True to get ecliptic coordinates.

EXOSIMS.TargetList.GaiaCatTargetList module

class EXOSIMS.TargetList.GaiaCatTargetList.**GaiaCatTargetList**(***specs*)

Bases: *TargetList*

Target list based on Gaia catalog inputs.

Parameters
****specs** – user specified values

set_catalog_attributes()

Hepler method that sets possible and required catalog attributes.

Sets attributes:

catalog_atts (list):
Attributes to try to copy from star catalog. Missing ones will be ignored and removed from this list.

required_catalog_atts(list):
Attributes that cannot be missing or nan.

stellar_mass()

Populates target list with 'true' and 'approximate' stellar masses

This method calculates stellar mass via interpolation of data from: "A Modern Mean Dwarf Stellar Color and Effective Temperature Sequence" http://www.pas.rochester.edu/~emamajek/EEM_dwarf_UBVIJHK_colors_Teff.txt Eric Mamajek (JPL/Caltech, University of Rochester)

For more details see MeanStars documentation.

Function called by reset sim

EXOSIMS.TargetList.KnownRVPlanetsTargetList module

class EXOSIMS.TargetList.KnownRVPlanetsTargetList.**KnownRVPlanetsTargetList**(***specs*)

Bases: *TargetList*

Target list based on population of known RV planets from IPAC. Intended for use with the KnownRVPlanets family of modules.

Parameters
****specs** – *Input Specification*

filter_target_list(***specs*)

Filtering is done as part of populating the table, so this method is overloaded to do nothing.

populate_target_list(***specs*)

This function is responsible for populating values from the star catalog into the target list attributes and enforcing attribute requirements.

Parameters
****specs** – *Input Specification*

set_catalog_attributes()

Hepler method that sets possible and required catalog attributes.

Sets attributes:

catalog_atts (list):
Attributes to try to copy from star catalog. Missing ones will be ignored and removed from this list.

required_catalog_atts(list):
Attributes that cannot be missing or nan.

2.28.1.14 EXOSIMS.TimeKeeping package

2.28.1.15 EXOSIMS.ZodiacalLight package

Submodules

EXOSIMS.ZodiacalLight.Mennesson module

class EXOSIMS.ZodiacalLight.Mennesson.**Mennesson**(*EZ_distribution='nominal_maxL_distribution.fits'*, ***specs*)

Bases: *Stark*

Mennesson Zodiacal Light class

gen_systemnEZ(*nStars*)

Ranomly generates the number of Exo-Zodi :param nStars: number of exo-zodi to generate :type nStars: int

Returns
numpy array of exo-zodi randomly selected from fitsdata

Return type
nEZ (numpy array)

zodi_latitudinal_correction_factor(*theta*, *model=None*, *interp_at=135*)

Compute zodiacal light latitudinal correction factor. This is a multiplicative factor to apply to zodiacal light intensity to account for the orientation of the dust disk with respect to the observer.

Parameters

- **theta** (*astropy.units.Quantity*) – Angle of disk. For local zodi, this is equivalent to the absolute value of the ecliptic latitude of the look vector. For exozodi, this is 90 degrees minus the inclination of the orbital plane.
- **model** (*str, optional*) – Model to use. Options are Lindler2006, Stark2014, or interp (case insensitive). See *Zodiacal and Exozodiacal Light* for details. Defaults to None
- **interp_at** (*float*) – If `model` is 'interp', interpolate Leinert Table 17 at this longitude. Defaults to 135.

Returns
Correction factor of zodiacal light at requested angles. Has same dimension as input.

Return type
float or numpy.ndarray

Note: Unlike the color correction factor, this quantity is wavelength independent and thus does not change if using power or photon units.

Note: The systems in the data file are all at 60 degrees inclination, so we scale by the 90-60=30 degree value of the correction factor.

EXOSIMS.ZodiacalLight.Stark module

class EXOSIMS.ZodiacalLight.Stark.**Stark**(*magZ=23.0*, *magEZ=22.0*, *varEZ=0.0*, ***specs*)

Bases: *ZodiacalLight*

Stark Zodiacal Light class

This class contains all variables and methods necessary to perform Zodiacal Light Module calculations in exoplanet mission simulation using the model from Stark et al. 2014.

calcfZmax(*sInds*, *Obs*, *TL*, *TK*, *mode*, *hashname*, *koTimes=None*)

Finds the maximum zodiacal light values for each star over an entire orbit of the sun not including keeoput angles

Parameters

- **sInds** (*integer array*) – the star indicies we would like fZmax and fZmaxInds returned for

- **Obs** (*module*) – Observatory module
- **TL** (*TargetList object*) – Target List Module
- **TK** (*TimeKeeping object*) – TimeKeeping object
- **mode** (*dict*) – Selected observing mode
- **hashname** (*string*) – hashname describing the files specific to the current json script
- **koTimes** (*Time(ndarray(float)), optional*) – Absolute MJD mission times from start to end in steps of 1 d

Returns

valfZmax[sInds] (~astropy.units.Quantity(~numpy.ndarray(float))):
the maximum fZ with units 1/arcsec**2

absTimefZmax[sInds] (astropy.time.Time):
returns the absolute Time the maximum fZ occurs

Return type
tuple

calcfZmin(*sInds*, *Obs*, *TL*, *TK*, *mode*, *hashname*, *koMap=None*, *koTimes=None*)

Finds the minimum zodiacal light values for each star over an entire orbit of the sun not including keeoput angles

Parameters

- **sInds[sInds]** (*integer array*) – the star indicies we would like fZmin and fZminInds returned for
- **Obs** (*module*) – Observatory module
- **TL** (*module*) – Target List Module
- **TK** (*TimeKeeping object*) – TimeKeeping object
- **mode** (*dict*) – Selected observing mode
- **hashname** (*string*) – hashname describing the files specific to the current json script
- **koMap** (*boolean ndarray, optional*) – True is a target unobstructed and observable, and False is a target unobservable due to obstructions in the keepout zone.
- **koTimes** (*Time(ndarray(float)), optional*) – Absolute MJD mission times from start to end in steps of 1 d

Returns

fZmins[n, TL.nStars] (~astropy.units.Quantity(~numpy.ndarray(float))):
fZMap, but only fZmin candidates remain. All other values are set to the maximum floating number. Units are 1/arcsec2

fZtypes [n, TL.nStars] (~numpy.ndarray(float)):
ndarray of flags for fZmin types that map to fZmins 0 - entering KO 1 - exiting KO 2 - local minimum max float - not a fZmin candidate

Return type
tuple

fZ(*Obs*, *TL*, *sInds*, *currentTimeAbs*, *mode*)

Returns surface brightness of local zodiacal light

Parameters

- **Obs** (*Observatory module*) – Observatory class object
- **TL** (*TargetList module*) – TargetList class object
- **sInds** (*integer ndarray*) – Integer indices of the stars of interest
- **currentTimeAbs** (*astropy Time array*) – Current absolute mission time in MJD
- **mode** (*dict*) – Selected observing mode

Returns
Surface brightness of zodiacal light in units of 1/arcsec2

Return type
Quantity(*ndarray*(float))

global_zodi_min(*mode*)

This is used to determine the minimum zodi value globally with a color correction

Parameters
mode (*dict*) – Selected observing mode

Returns
The global minimum zodiacal light value for the observing mode, in (1/arcsec**2)

Return type
Quantity

2.28.1.16 EXOSIMS.util package

Subpackages

EXOSIMS.util.KeplerSTM_C package

Submodules

EXOSIMS.util.KeplerSTM_C.CyKeplerSTM module

Submodules

EXOSIMS.util.CheckScript module

class EXOSIMS.util.CheckScript.**CheckScript**(*scriptfile*, *outspec*)

Bases: object

Class that facilitates the comparison of the input script fiel for EXOSIMS and the outspec for a simulation. CheckScript highlights any differences between the two.

recurse(*json1*, *json2*, *pretty_print=False*, *recurse_level=0*, *outtext=''*)

This function iterates recursively through the JSON structures of the script file and the simulation outspec, checking them against one another. Outputs the following warnings:

1. Catches parameters that are never used in the sim or are not in the outspec
2. Catches parameters that are unspecified in the script file and notes default value used
3. Catches mismatches in the modules being imported
4. Catches cases where the value in the script file does not match the value in the outspec

Parameters

- **json1** (*dict*) – The scriptfile json input.
- **json2** (*dict*) – The outspec json input
- **pretty_print** (*boolean*) – Write output to a single return string rather than sequentially
- **recurse_level** (*int*) – The current level of recursion
- **outtext** (*string*) – The concatinated output text

Returns
The concatinated output text

Return type
outtext (string)

write_file(*filename*)

EXOSIMS.util.InverseTransformSampler module

class EXOSIMS.util.InverseTransformSampler.**InverseTransformSampler**(*f*, *xMin*, *xMax*, *nints=10000*)

Bases: object

Approximate Inverse Transform Sampler for arbitrary distributions defined via a PDF encoded as a function (or lambda function)

Parameters

- **f** (*function*) – Probability density function. Must be able to operate on numpy ndarrays. Function does not need to be normalized over the sampling interval.
- **xMin** (*float*) – Minimum of interval to sample (inclusive).
- **xMax** (*float*) – Maximum of interval to sample (inclusive).
- **nints** (*int*) – Number of intervals to use in approximating CDF. Defaults to 10000

f, xMin, xMax
As above

Notes

If xMin == xMax, return values will all exactly equal xMin. To sample call the object with the desired number of samples.

EXOSIMS.util.RejectionSampler module

class EXOSIMS.util.RejectionSampler.**RejectionSampler**(*f*, *xMin*, *xMax*)

Bases: object

Simple Rejection Sampler for arbitrary distributions defined via a PDF encoded as a function (or lambda function)

Parameters

- **f** (*function*) – Probability density function. Must be able to operate on numpy ndarrays. Function does not need to be normalized over the sampling interval.

- **xMin** (*float*) – Minimum of interval to sample (inclusive).
- **xMax** (*float*) – Maximum of interval to sample (inclusive).

f, xMin, xMax

As above

Notes

If xMin == xMax, return values will all exactly equal xMin. To sample call the object with the desired number of samples.

calcM()

Calculate the maximum bound of the distribution over the sampling interval.

EXOSIMS.util.csv_fix module

EXOSIMS.util.csv_fix.**csv_fix**(*folder*, *global_changes=[]*, ***kwargs*)

This function changes the headers of csv files to match EXOSIMS conventions. It was written primarily for coronagraph performance specs such as "coro_area" that have associated lambda functions to standardize the inputs.

Note that it looks for all csv files in every subfolder and is just a search and replace in text

Parameters

- **folder** (*str or Path object*) – The path to the folder
- **global_changes** (*list of tuple*) – A global change represents a change to be made in all csv files where the first value of the tuple is the change to be searched for and the second value is the replacement value
- **kwargs** (*list of tuples*) – This is used for file specific changes, the keyword indicates what files to be changed and the tuple corresponding to the keyword is the change. See Notes.

Returns

None

Notes

For example if you want to change every "I" to "intensity", but only in files that have "CGPERF" in their name you would call the function as: `csv_fix(folder, CGPERF=[("I", "intensity")])` The same can be done with multiple changes for the files, so now you also want to change "occTrans" to "occ_trans" in CGPERF: `csv_fix(folder, CGPERF=[("I", "intensity"), ("occTrans", "occ_trans")])` But if that change is in files with "OTHEREXAMPLE" in their name the call is: `csv_fix(folder, CGPERF=[("I", "intensity")], OTHEREXAMPLE=[("occTrans", "occ_trans")])`

EXOSIMS.util.deltaMag module

EXOSIMS.util.deltaMag.**betaStar_Lambert**()

Compute the Lambert phase function deltaMag-maximizing phase angle

Parameters
None –

Returns
Value of beta^* in radians

Return type
float

EXOSIMS.util.deltaMag.**deltaMag**(*p*, *Rp*, *d*, *Phi*)

Calculates delta magnitudes for a set of planets, based on their albedo, radius, and position with respect to host star.

Parameters

- **p** (*ndarray*) – Planet albedo
- **Rp** (*astropy Quantity array*) – Planet radius in units of km
- **d** (*astropy Quantity array*) – Planet-star distance in units of AU
- **Phi** (*ndarray*) – Planet phase function

Returns
Planet delta magnitudes

Return type
ndarray

EXOSIMS.util.deltaMag.**max_deltaMag_Lambert**(*Completeness*, *s=None*)

Calculate the maximum deltaMag at given separation(s) assuming a Lambert phase function

Parameters

- **Completeness** (BrownCompleteness) – BrownCompleteness object
- **s** (*float or ndarray, optional*) – Projected separations (in AU) to compute minimum delta mag at. If None (default) then uses Completeness.xnew

Returns
Maximum deltaMag values

Return type
ndarray

EXOSIMS.util.deltaMag.**min_deltaMag_Lambert**(*Completeness*, *s=None*)

Calculate the minimum deltaMag at given separation(s) assuming a Lambert phase function

Parameters

- **Completeness** (BrownCompleteness) – BrownCompleteness object
- **s** (*float or ndarray, optional*) – Projected separations (in AU) to compute minimum delta mag at. If None (default) then uses Completeness.xnew

Returns
Minimum deltaMag values

Return type
ndarray

EXOSIMS.util.eccanom module

EXOSIMS.util.eccanom.**eccanom**(*M*, *e*)

Finds eccentric anomaly from mean anomaly and eccentricity

This method uses algorithm 2 from Vallado to find the eccentric anomaly from mean anomaly and eccentricity.

Parameters

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

Returns
eccentric anomaly

Return type
E (float or ndarray)

EXOSIMS.util.fakeMultiRunAnalysis module

Template Plotting utility for post processing automation Written by: Dean Keithly Written on: 10/11/2018

class EXOSIMS.util.fakeMultiRunAnalysis.**fakeMultiRunAnalysis**(*args=None*)

Bases: object

Template format for adding multiRunPostProcessing to any plotting utility multiRunPostProcessing method is a required method with the below inputs to work with runPostProcessing.py

multiRunPostProcessing(*PPoutpath=None*, *folder=None*)

This is called by runPostProcessing :param PPoutpath: :type PPoutpath: string :param folder: :type folder: string

Return:

EXOSIMS.util.fakeSingleRunAnalysis module

Template Plotting utility for post processing automation Written by: Dean Keithly Written on: 10/11/2018

class EXOSIMS.util.fakeSingleRunAnalysis.**fakeSingleRunAnalysis**(*args=None*)

Bases: object

Template format for adding singleRunPostProcessing to any plotting utility singleRunPostProcessing method is a required method with the below inputs to work with runPostProcessing.py

singleRunPostProcessing(*PPoutpath=None*, *folder=None*)

This is called by runPostProcessing :param PPoutpath: :type PPoutpath: string :param folder: :type folder: string

EXOSIMS.util.getExoplanetArchive module

EXOSIMS.util.getExoplanetArchive.**cacheExoplanetArchiveQuery**(*basestr: str*, *querystring: str*, *forceNew: bool = False*, ***specs: Any*) → DataFrame

Look for cached query results, and return newest one. If none exist, execute the query.

Parameters

- **basestr** (*str*) – Base of the cache filename.
- **querystring** (*str*) – Exact query string to use. See `queryExoplanetArchive` for details
- **forceNew** (*bool*) – Run a fresh query even if results exist on disk.
- ****specs** (*any*) – Any additional kewyords to pass to `get_downloads_dir`

Returns
Result of query

Return type
pandas.DataFrame

EXOSIMS.util.getExoplanetArchive.**getExoplanetArchiveAliases**(*name: str*) → Optional[Dict[str, Any]]

Query the exoplanet archive's system alias service and return results

See: https://exoplanetarchive.ipac.caltech.edu/docs/sysaliases.html

Parameters
name (*str*) – Target name to resolve

Returns
Dictionary

Return type
dict or None

Note: This has a tendency to get stuck when run in a loop. This is set up to fail after 10 seconds and retry once with a 30 second timeout.

EXOSIMS.util.getExoplanetArchive.**getExoplanetArchivePS**(*forceNew: bool = False*, ***specs: Dict[Any, Any]*) → DataFrame

Get the contents of the Exoplanet Archive's Planetary Systems table and cache results. If a previous query has been saved to disk, load that.

Parameters
forceNew (*bool*) – Run a fresh query even if results exist on disk.

Returns
Planetary Systems table

Return type
pandas.DataFrame

EXOSIMS.util.getExoplanetArchive.**getExoplanetArchivePSCP**(*forceNew: bool = False*, ***specs: Any*) → DataFrame

Get the contents of the Exoplanet Archive's Planetary Systems Composite Parameters table and cache results. If a previous query has been saved to disk, load that.

Parameters
forceNew (*bool*) – Run a fresh query even if results exist on disk.

Returns
Planetary Systems composited parameters table

Return type
pandas.DataFrame

EXOSIMS.util.getExoplanetArchive.**getHWOStars**(*forceNew: bool = False*, ***specs: Any*) → DataFrame

Get the contents of the ExEP HWO Star List and cache results. If a previous query has been saved to disk, load that.

Parameters
forceNew (*bool*) – Run a fresh query even if results exist on disk.

Returns
Planetary Systems composited parameters table

Return type
pandas.DataFrame

See: https://exoplanetarchive.ipac.caltech.edu/docs/2645_NASA_ExEP_Target_List_HWO_Documentation_2023.pdf # noqa: E501

EXOSIMS.util.getExoplanetArchive.**queryExoplanetArchive**(*querystring: str*, *filename: Optional[str] = None*) → DataFrame

Query the exoplanet archive, optionally save results to disk, and return the result as a pandas dataframe.

Parameters

- **querystring** (*str*) – Exact query string to use. Do not include format (csv will be specified). See: https://exoplanetarchive.ipac.caltech.edu/docs/TAP/usingTAP.html for details. A valid string is: "select+*+from+pscomppars"
- **filename** (*str*) – Full path to save file. If None (default) results are not written to disk. Data will be written in pickle format.

Returns
Result of query

Return type
pandas.DataFrame

EXOSIMS.util.get_dirs module

The get_dirs utility module contains functions which set up and find the cache and download folders for EXOSIMS.

These folders are set up similar to Astropy where POSIX systems give:

/home/user/.EXOSIMS/cache /home/user/.EXOSIMS/downloads

and Windows systems generally give:

C:/Users/User/.EXOSIMS/cache C:/Users/User/.EXOSIMS/downloads

An additional function is given to download a file from a website and store in the downloads folder.

EXOSIMS.util.get_dirs.**get_cache_dir**(*cachedir: Optional[str] = None*) → str

Return EXOSIMS cache directory. Order of priority is: 1. Input (typically taken from JSON spec script) 2. EXOSIMS_CACHE_DIR environment variable 3. Default in $HOME/.EXOSIMS/cache (for whatever $HOME is returned by get_home_dir)

In each case, the directory is checked for read/write/access permissions. If any permissions are missing, will return default path.

Returns
Path to EXOSIMS cache directory

Return type
str

EXOSIMS.util.get_dirs.**get_downloads_dir**(*downloadsdir: Optional[str] = None*) → str

Return EXOSIMS downloads directory. Order of priority is:

1. Input (typically taken from JSON spec script)
2. EXOSIMS_CACHE_DIR environment variable
3. Default in $HOME/.EXOSIMS/downloads (for whatever $HOME is returned by get_home_dir)

In each case, the directory is checked for read/write/access permissions. If any permissions are missing, will return default path.

Returns
Path to EXOSIMS downloads directory

Return type
str

EXOSIMS.util.get_dirs.**get_exosims_dir**(*dirtype: str*, *indir: Optional[str] = None*) → str

Return path of EXOSIMS input/output directory. Nominally this is either for the cache directory or the downloads directory, but others may be added in the future.

Order of selection priority is:

1. Input path (typically taken from JSON spec script)
2. Environment variable (EXOSIMS_DIRTYPE_DIR)
3. Default (nominally $HOME/.EXOSIMS/dirtype for whatever $HOME is returned by get_home_dir)

In each case, the directory is checked for read/write/access permissions. If any permissions are missing, will return default path. If default is still not useable, will throw AssertionError.

Parameters

- **dirtype** (*str*) – Directory type (currently limited to 'cache' or 'downloads'
- **indir** (*str*) – Full path (may include environment variables and other resolveable elements). If set, will be tried first.

Returns
Path to EXOSIMS directory specified by dirtype

Return type
str

EXOSIMS.util.get_dirs.**get_home_dir**() → str

Finds the Home directory for the system.

Returns
Path to Home directory

Return type
str

EXOSIMS.util.get_dirs.**get_paths**(*qFile=None*, *specs=None*, *qFargs=None*)

This function gets EXOSIMS paths in priority order:

1. Argument specified path (runQueue argument)
2. Queue file specified path
3. JSON input specified path
4. Environment Variable
5. Current working directory

- Used by TimeKeeping to search for Observing Block Schedule Files
- Used by runQueue to get Script Paths, specify run output dir, and runLog.csv location
- All ENVIRONMENT set keys must contain the keyword 'EXOSIMS'

Parameters

- **qFile** (*str*) – Queue file
- **specs** (*dict*) – fields from a json script
- **qFargs** (*dict*) – arguments from the queue JSON file

Returns

dictionary containing paths to folders where each of these are located

Return type

dict

EXOSIMS.util.get_module module

EXOSIMS.util.get_module.**get_module**(*name*, *folder=None*, *silent=False*)

Import specific or Prototype class module.

There are three ways to use the name argument:

Case 1: Applies when name ends in .py: it is interpreted as the name of
a python source file implementing the stated module type. For example, $HOME/EXOSIMS_local/MyObservatory.py which would be a module that implements a MyObservatory class. Shell variables are expanded.

Case 2: Default case. The name is interpreted as an EXOSIMS module, which
must be loaded. If folder is given (best practice), we look first at EXOSIMS.folder.name for the module ("specific module"), and then at EXOSIMS.Prototypes.name ("prototype module"). As a shortcut (provided that folder is given) if the name is empty or all-blanks, we look at EXOSIMS.Prototypes.folder.

Case 3: Used when a . (a Python module separator) is part of the name. The
name is interpreted as a loadable Python module, and is loaded. This mechanism supports loading of modules from anywhere on the PYTHONPATH. For example, as in case 1, if $HOME/EXOSIMS_local/__init__.py exists (making EXOSIMS_local a valid Python package), and $HOME is on $PYTHONPATH, then giving name as EXOSIMS_local.MyObservatory will load the MyObservatory class within the given file. This supports locally-customized module sets.

Parameters

- **name** (*str*) – string containing desired class name (cases 2, 3 above) OR full path to desired module (case 1). The class name must be the same as the file name.

- **folder** (*str*) – The module type (e.g., Observatory or TimeKeeping), which is validated against the _modtype attribute of the loaded class. For specific modules, this is the same as the name of the folder housing the desired class.
- **silent** (*bool*) – Set True to suppress printing import information. Defaults False.

Returns
module (class) that was requested

Return type
desired_module (object)

EXOSIMS.util.get_module.**get_module_chain**(*names*)

Attempt to load a module from an ordered list of module names until one succeeds

Module names may be given fully as:
EXOSIMS.OpticalSystem.Nemati

or as wildcards like:
EXOSIMS.*.Nemati

Wildcards, if given, must match only one module.

EXOSIMS.util.get_module.**get_module_from_specs**(*specs*, *modtype*)

Import specific or Prototype class module using specs dictionary.

The universal idiom for initializing an EXOSIMS module follows the pattern:

```
get_module(specs['modules']['TimeKeeping'], 'TimeKeeping')(**specs)
```

Here, get_module loads the module, and the invocation with **specs runs its __init__ method.

The present function abstracts the first half of the idiom by returning:

```
get_module(specs['modules'][modtype], modtype)
```

thus the above idiom may be replaced by:

```
get_module_from_specs(specs, 'TimeKeeping')(**specs)
```

which is shorter, and avoids the duplication of the module type. Note: we do not abstract the specs as well, because many callers might wish to modify the given **specs with keywords, or to separate getting the class from initializing it.

Parameters

- **specs** (*dict*) – specs dictionary with a 'modules' key present.
- **modtype** (*str*) – The module type (e.g., Observatory or TimeKeeping) of the EXOSIMS module to fetch.

Returns
module (class) that was requested

Return type
desired_module (object)

EXOSIMS.util.get_module.**get_module_in_package**(*name*, *folder*)

Get an EXOSIMS module from a given package: handles most requests for modules.

Return value is a Python package matching the given name.

EXOSIMS.util.get_module.**modules_below_matching**(*pkg*, *name*)

Return a list of modules, below the named package, matching a given name.

Example usage:

```
pkgs = modules_below_matching('EXOSIMS', 'Nemati')
```

which would find the unique module `EXOSIMS.OpticalSystem.Nemati` and return a length-1 list of that string. It matches recursively, so intervening modules (in the above, `OpticalSystem`) do not matter.

EXOSIMS.util.get_module.**shorten_name**(*filename*)

Produce a shortened version of a file or package path for one-line printing.

EXOSIMS.util.get_module.**wildcard_expand**(*pattern*)

Expand a pattern like pkg.*.module into a full package name like pkg.subpkg.module.

The full package name, which is returned, must be unique, or an error is raised. Example usage:

```
module = wildcard_expand('EXOSIMS.*.Nemati')
```

which would find the unique module named `EXOSIMS.OpticalSystem.Nemati`

The returned value is a single string.

EXOSIMS.util.input_script_check module

EXOSIMS.util.input_script_check.**check_for_unused_kws**(*specs: Union[Dict[str, Any], str]*) → Tuple[List[str], Dict[str, Any]]

Check input specification for consistency with module inputs

Parameters
specs (*str or dict*) – Either full path to JSON script or an *Input Specification* dict

Returns

unused (list):
List of unused keywords

specs (dict):
Original input (useful if read from disk)

Return type
tuple

EXOSIMS.util.input_script_check.**parse_mods**(*specs: Dict[str, Any]*) → Dict[str, type]

Check for presence of all required modules in input specs and return list of module class types.

Parameters
specs (*str or dict*) – Either full path to JSON script or an *Input Specification* dict

Returns
dict of all module classes along with MissionSim

Return type
dict

EXOSIMS.util.keplerSTM module

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

Bases: object

calcSTM(*dt*)

calcSTM_vallado(*dt*)

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

psi2c2c3(*psi0*)

takeStep(*dt*)

updateState(*x0*)

EXOSIMS.util.keplerSTM_indprop module

class EXOSIMS.util.keplerSTM_indprop.**planSys**(*x0*, *mu*, *epsmult=4.0*, *noc=False*)

Bases: object

calcSTM(*dt*, *j*)

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

takeStep(*dt*)

updateState(*x0*)

EXOSIMS.util.keyword_fun module

EXOSIMS.util.keyword_fun.**check_opticalsystem_kws**(*specs: Dict[str, Any]*, *OS: Any*)

Check input specification against an optical system object

Parameters

- **specs** (*str or dict*) – Either full path to JSON script or an *Input Specification* dict
- **OS** (*OpticalSystem*) – OpticalSystem object

Returns

Description of what's wrong with the input (blank if its all good).

Return type

str

EXOSIMS.util.keyword_fun.**get_all_args**(*mod: type*) → List[str]

Return list of all arguments to inits of every base of input class

Parameters

mod (*type*) – Class of object of interest

Returns

list:

List of all arguments to mod.__init__()

EXOSIMS.util.keyword_fun.**get_all_mod_kws**(*mods: Dict[str, type]*) → Tuple[List[str], List[str]]

Collect all keywords from all modules

Parameters
mods (*dict*) – dict of all module classes along with MissionSim

Returns

allkws (list):
All keywords

allkwmods (list):
The names of the modules the keywords belong to.

ukws (~numpy.ndarray(str)):
Unique keywords (excluding self and scriptfile)

ukwcounts (~numpy.ndarray(int)):
Unique keyword counts

Return type
tuple

EXOSIMS.util.keyword_fun.**get_allmod_args**(*sim*) → Dict[str, List[str]]

Return list of all arguments to all inits of all modules in a MissionSim Object

Parameters
sim (*MissionSim*) – MissionSim object

Returns
Dictionary of all input arguments and the modules in which they appear

Return type
dict

EXOSIMS.util.makeSimilarScripts module

The purpose of this script is to take a template json script and make a series of "similar" json scripts from an original json script

This script is designed for:

- Sweep Single Parameter
- Sweep Multiple Parameters over Multiple Values
- Sweep Multiple Combinations of Parameters over

makeSimilarScripts.py is designed to run from the 'EXOSIMS/Scripts/' Folder

Another example %run makeSimilarScripts.py –makeSimilarInst '/full path to/makeSimilar.json'

Written by Dean Keithly on 6/28/2018

EXOSIMS.util.makeSimilarScripts.**createScriptFolder**(*makeSimilarInst*, *sourcefile*)

This method creates a 'Script Folder' - a new folder with name 'makeSimilarInst_sourcefile' in 'EXOSIMS/Scripts/' returns folderName

EXOSIMS.util.makeSimilarScripts.**createScriptName**(*prepend*, *makeSimilarInst*, *sourcefile*, *ind*)

This Script creates the ScriptName

EXOSIMS.util.makeSimilarScripts.**moveDictFiles**(*myDict*, *folderName*)

This Script copies the OB.csv files to the makeSimilar_Template folder

EXOSIMS.util.maxConsecutiveTrue module

Finds the maximum consecutive number of true values in a boolean array Written by: Dean Keithly Written On: 7/7/2021

EXOSIMS.util.maxConsecutiveTrue.**maxConsecutiveTrue**(*arr*)

Finds largest number of consecutive True booleans in the array :param ndarray: arr - boolean array

Returns
maxNum

Return type
float

EXOSIMS.util.memoize module

class EXOSIMS.util.memoize.**memoize**(*func*)

Bases: object

Decorator. Caches a function's return value each time it is called. If called later with the same arguments, the cached value is returned (not reevaluated).

Code taken directly from the PythonDecoratorLibrary:
https://wiki.python.org/moin/PythonDecoratorLibrary

Parameters
func (*function*) – Function or instance method to memoize

func

The memoized function or instance method

Type
function

cache

Dictionary with key-value pairs consisting of function arguments- function evaluations

Type
dict

EXOSIMS.util.partitionSphere module

EXOSIMS.util.partitionSphere.**add_caps**(*psi*)

EXOSIMS.util.partitionSphere.**partitionSphere**(*N*, *d*)

EXOSIMS.util.phaseFunctions module

Phase Functions

See also [Keithly2021]

EXOSIMS.util.phaseFunctions.**betaFunc**(*inc*, *v*, *w*)

Calculate the planet phase angle

Parameters

- **inc** (*float or numpy.ndarray*) – planet inclination in rad
- **v** (*numpy.ndarray*) – planet true anomaly in rad
- **w** (*numpy.ndarray*) – planet argument of periapsis

Returns
beta, planet phase angle

Return type
numpy.ndarray

EXOSIMS.util.phaseFunctions.**hyperbolicTangentPhaseFunc**(*beta*, *A*, *B*, *C*, *D*, *planetName=None*)

Optimal Parameters for Earth Phase Function basedon mallama2018 comparison using mallama2018PlanetProperties.py: A=1.85908529, B=0.89598952, C=1.04850586, D=-0.08084817 Optimal Parameters for All Solar System Phase Function basedon mallama2018 comparison using mallama2018PlanetProperties.py: A=0.78415 , B=1.86890455, C=0.5295894 , D=1.07587213

Parameters

- **beta** (*astropy.units.quantity.Quantity or numpy.ndarray*) – Phase Angle in radians
- **A** (*float*) – Hyperbolic phase function parameter
- **B** (*float*) – Hyperbolic phase function paramter
- **C** (*float*) – Hyperbolic phase function parameter
- **D** (*float*) – Hyperbolic phase function parameter
- **planetName** (*string or None*) – planet name string all lower case for one of 8 solar system planets

Returns
Phi, phase angle in degrees

Return type
numpy.ndarray

EXOSIMS.util.phaseFunctions.**hyperbolicTangentPhaseFuncInverse**(*Phi*, *A*, *B*, *C*, *D*, *planetName=None*)

Optimal Parameters for Earth Phase Function basedon mallama2018 comparison using mallama2018PlanetProperties.py: A=1.85908529, B=0.89598952, C=1.04850586, D=-0.08084817 Optimal Parameters for All Solar System Phase Function basedon mallama2018 comparison using mallama2018PlanetProperties.py: A=0.78415 , B=1.86890455, C=0.5295894 , D=1.07587213

Parameters

- **Phi** (*numpy.ndarray*) – phase angle in degrees
- **A** (*float*) – Hyperbolic phase function parameter

- **B** (*float*) – Hyperbolic phase function paramter
- **C** (*float*) – Hyperbolic phase function parameter
- **D** (*float*) – Hyperbolic phase function parameter
- **planetName** (*string or None*) – planet name string all lower case for one of 8 solar system planets

Returns
beta, Phase Angle in degrees

Return type
numpy.ndarray

EXOSIMS.util.phaseFunctions.**phase_Earth**(*beta*)

Earth phase function Valid from 0 to 180 deg

Parameters
beta (*numpy.ndarray*) – beta, phase angle in degrees

Returns
phase function values

Return type
numpy.ndarray

EXOSIMS.util.phaseFunctions.**phase_Jupiter_1**(*beta*)

Jupiter phase function Valid from 0 to 12 deg

Parameters
beta (*numpy.ndarray*) – beta, phase angle in degrees

Returns
phase function values

Return type
numpy.ndarray

EXOSIMS.util.phaseFunctions.**phase_Jupiter_2**(*beta*)

Jupiter phase function Valid from 12 to 130 deg

Parameters
beta (*numpy.ndarray*) – beta, phase angle in degrees

Returns
phase function values

Return type
numpy.ndarray

EXOSIMS.util.phaseFunctions.**phase_Jupiter_melded**(*beta*)

Jupiter phase function Valid from 0 to 130 degrees

Parameters
beta (*numpy.ndarray*) – beta, phase angle in degrees

Returns
phase function values

Return type
numpy.ndarray

EXOSIMS.util.phaseFunctions.**phase_Mars_1**(*beta*)

Mars phase function Valid from 0 to 50 deg

Parameters
beta (*numpy.ndarray*) – beta, phase angle in degrees

Returns
phase function values

Return type
numpy.ndarray

EXOSIMS.util.phaseFunctions.**phase_Mars_2**(*beta*)

Mars phase function Valid from 50 to 180 deg

Parameters
beta (*numpy.ndarray*) – beta, phase angle in degrees

Returns
phase function values

Return type
numpy.ndarray

EXOSIMS.util.phaseFunctions.**phase_Mars_melded**(*beta*)

Mars phase function Valid from 0 to 180 degrees

Parameters
beta (*numpy.ndarray*) – beta, phase angle in degrees

Returns
phase function values

Return type
numpy.ndarray

EXOSIMS.util.phaseFunctions.**phase_Mercury**(*beta*)

Mercury phase function Valid from 0 to 180 deg

Parameters
beta (*numpy.ndarray*) – beta, phase angle in degrees

Returns
phase function values

Return type
numpy.ndarray

EXOSIMS.util.phaseFunctions.**phase_Neptune**(*beta*)

Neptune phase function Valid for beta 0 to 133.14 deg

Parameters
beta (*numpy.ndarray*) – beta, phase angle in degrees

Returns
phase function values

Return type
numpy.ndarray

EXOSIMS.util.phaseFunctions.**phase_Neptune_melded**(*beta*)

Neptune phase function

Parameters
beta (*numpy.ndarray*) – beta, phase angle in degrees

Returns
phase function values

Return type
numpy.ndarray

EXOSIMS.util.phaseFunctions.**phase_Saturn_2**(*beta*)

Saturn phase function (Globe Only Earth Observations) Valid beta from 0 to 6.5 deg

Parameters
beta (*numpy.ndarray*) – beta, phase angle in degrees

Returns
phase function values

Return type
numpy.ndarray

EXOSIMS.util.phaseFunctions.**phase_Saturn_3**(*beta*)

Saturn phase function (Globe Only Pioneer Observations) Valid beta from 6 to 150. deg

Parameters
beta (*numpy.ndarray*) – beta, phase angle in degrees

Returns
phase function values

Return type
numpy.ndarray

EXOSIMS.util.phaseFunctions.**phase_Saturn_melded**(*beta*)

Saturn phase function

Parameters
beta (*numpy.ndarray*) – beta, phase angle in degrees

Returns
phase function values

Return type
numpy.ndarray

EXOSIMS.util.phaseFunctions.**phase_Uranus**(*beta*, *phi=-82.0*)

Uranus phase function Valid for beta 0 to 154 deg

Parameters
beta (*numpy.ndarray*) – beta, phase angle in degrees

Returns
phase function values

Return type
numpy.ndarray

EXOSIMS.util.phaseFunctions.**phase_Uranus_melded**(*beta*)

Uranus phase function

Parameters
beta (*numpy.ndarray*) – beta, phase angle in degrees

Returns
phase function values

Return type
numpy.ndarray

EXOSIMS.util.phaseFunctions.**phase_Venus_1**(*beta*)

Venus phase function Valid from 0 to 163.7 deg

Parameters
beta (*numpy.ndarray*) – beta, phase angle in degrees

Returns
phase function values

Return type
numpy.ndarray

EXOSIMS.util.phaseFunctions.**phase_Venus_2**(*beta*)

Venus phase function Valid from 163.7 to 179 deg

Parameters
beta (*numpy.ndarray*) – beta, phase angle in degrees

Returns
phase function values

Return type
numpy.ndarray

EXOSIMS.util.phaseFunctions.**phase_Venus_melded**(*beta*)

Venus phae function Valid from 0 to 180 deg

Parameters
beta (*numpy.ndarray*) – beta, phase angle in degrees

Returns
phase function values

Return type
numpy.ndarray

EXOSIMS.util.phaseFunctions.**phi_lambert**(*beta*, *phiIndex=array([], dtype=float64)*)

Lambert phase function (isotropic scattering)

See: [Sobolev1975]

Parameters

- **beta** (*astropy.units.quantity.Quantity or numpy.ndarray*) – phase angle array in radians
- **phiIndex** (*numpy.ndarray*) – array of indicies of type of exoplanet phase function to use, ints 0-7

Returns
Phi, phase function values between 0 and 1

Return type
numpy.ndarray

EXOSIMS.util.phaseFunctions.**phiprime_phi**(*phi*)

Helper method for Uranus phase function Valid for phi from -82 to 82 deg

Parameters
phi (*numpy.ndarray*) – phi, planet rotation axis offset in degrees, floats

Returns
phiprime, in deg, floats

Return type
numpy.ndarray

EXOSIMS.util.phaseFunctions.**quasiLambertPhaseFunction**(*beta*, *phiIndex=array([], dtype=float64)*)

Quasi Lambert Phase Function from [Agol2007]

Parameters

- **beta** (*astropy.units.quantity.Quantity or numpy.ndarray*) – planet phase angles in radians
- **phiIndex** (*numpy.ndarray*) – array of indicies of type of exoplanet phase function to use, ints 0-7

Returns
Phi, phase function value

Return type
numpy.ndarray

EXOSIMS.util.phaseFunctions.**quasiLambertPhaseFunctionInverse**(*Phi*, *phiIndex=array([], dtype=float64)*)

Quasi Lambert Phase Function Inverse

Parameters

- **Phi** (*numpy.ndarray*) – Phi, phase function value, floats
- **phiIndex** (*numpy.ndarray*) – array of indicies of type of exoplanet phase function to use, ints 0-7

Returns
beta, planet phase angles in rad, floats

Return type
numpy.ndarray

EXOSIMS.util.phaseFunctions.**realSolarSystemPhaseFunc**(*beta*, *phiIndex=array([], dtype=float64)*)

Uses the phase functions from Mallama 2018 implemented in mallama2018PlanetProperties.py

Parameters

- **beta** (*astropy.units.quantity.Quantity or numpy.ndarray*) – phase angle array in degrees
- **phiIndex** (*numpy.ndarray*) – array of indicies of type of exoplanet phase function to use, ints 0-7

Returns
Phi, phase function values between 0 and 1

Return type
numpy.ndarray

EXOSIMS.util.phaseFunctions.**transitionEnd**(*x*, *a*, *b*)

Smoothly transition from one 1 to 0

Smaller b is sharper step a is midpoint, s(a)=0.5

Parameters

- **x** (*numpy.ndarray*) – x, in deg input value in deg, floats
- **a** (*numpy.ndarray*) – a, transition midpoint in deg, floats
- **b** (*numpy.ndarray*) – transition slope

Returns
s, transition value from 1 to 0

Return type
numpy.ndarray

EXOSIMS.util.phaseFunctions.**transitionStart**(*x*, *a*, *b*)

Smoothly transition from one 0 to 1

Parameters

- **x** (*numpy.ndarray*) – x, in deg input value in deg, floats
- **a** (*numpy.ndarray*) – transition midpoint in deg, floats
- **b** (*numpy.ndarray*) – transition slope

Returns
s, Transition value from 0 to 1, floats

Return type
numpy.ndarray

EXOSIMS.util.photometricModels module

Various useful photometric models from the literature

class EXOSIMS.util.photometricModels.**Box1D**(*step=0.01*, **args*, ***kwargs*)

Bases: Box1D

Same as *synphot.models.Box1D*, except with `step` input.

sampleset(*step=None*, *minimal=False*)

Return `x` array that samples the feature.

Parameters

- **step** (*float*) – Distance of first and last points w.r.t. bounding box.
- **minimal** (*bool*) – Only return the minimal points needed to define the box; i.e., box edges and a point outside on each side.

EXOSIMS.util.photometricModels.**TraubApparentMagnitude**(*V*, *BV*, *lam*)

Star apparent magnitude at a given wavelength

This implements the model from [Traub2016] which has a stated valid range between 0.4 and 1 um.

Parameters

- **V** (*float or ndarray(float)*) – Johnson V-band magnitude(s)
- **BV** (*float or ndarray(float)*) – B-V color. Must be same type and dimensionality as V
- **lam** (*astropy.units.Quantity*) – Wavelength at which to evaluate the flux. Must be scalar.

Returns
Apparent magnitude at wavelength lam. This will have the same dimensionality as the V and BV inputs.

Return type
ndarray(float)

EXOSIMS.util.photometricModels.**TraubStellarFluxDensity**(*V*, *BV*, *lam*)

Stellar spectral flux density at a given wavelength.

This implements the model from [Traub2016] which has a stated valid range between 0.4 and 1 um.

> **Warning:** Values will be returned for wavelengths outside the valid range, but a warning will be generated.

Parameters

- **V** (*float or ndarray(float)*) – Johnson V-band magnitude(s)
- **BV** (*float or ndarray(float)*) – B-V color. Must be same type and dimensionality as V
- **lam** (*astropy.units.Quantity*) – Wavelength at which to evaluate the flux. Must be scalar.

Returns
Stellar flux desnsity at wavelength lam. This will have the same dimensionality as the V and BV inputs. Default units are ph/s/cm^2/nm

Return type
Quantity

EXOSIMS.util.photometricModels.**TraubZeroMagFluxDensity**(*lam*)

Zero-magnitude spectral flux density at a given wavelength

This implements the model from [Traub2016] which has a stated valid range between 0.4 and 1 um.

Parameters
lam (*astropy.units.Quantity*) – Wavelength at which to evaluate the flux.

Returns
Zero magnitude flux desnsity at wavelength lam. This will have the same dimensionality as the input. Default units are ph/s/cm^2/nm

Return type
Quantity

EXOSIMS.util.planet_star_separation module

Planet Star Separation Written By: Dean Keithly Written On: 11/13/2020

EXOSIMS.util.planet_star_separation.**planet_star_separation**(*a*, *e*, *v*, *w*, *i*)

Following directly from Keithly 2021. Calculates planet star separation given KOE

Parameters

- **a** (*numpy.ndarray(float)*) – planet semi-major axis in AU
- **e** (*numpy.ndarray(float)*) – planet eccentricity
- **v** (*numpy.ndarray(float)*) – planet true anomaly rad
- **w** (*numpy.ndarray(float)*) – planet argument of periapsis rad
- **i** (*numpy.ndarray(float)*) – planet inclination rad

Returns

planet-star separations in AU

Return type

numpy.ndarray(float)

EXOSIMS.util.process_opticalsys_package module

EXOSIMS.util.process_opticalsys_package.**check_header_vals**(*header_vals*, *hdr*)

Utility method for checking values in headers

Parameters

- **header_vals** (*dict, optional*) – Current value set
- **hdr** (*numpy.ndarray*) – Header

Returns

Updated header values

Return type

dict

EXOSIMS.util.process_opticalsys_package.**get_center_vals**(*hdr*, *data*)

Utility method for extracting center pixel values from header or data

Parameters

- **hdr** (*astropy.io.fits.header.Header*) – Header
- **data** (*numpy.ndarray*) – Data

Returns

[xcenter, ycenter]

Return type

list(float)

EXOSIMS.util.process_opticalsys_package.**process_opticalsys_package**(*basepath*, *stellar_intensity_file='stellar_intens.fits'*, *stellar_intensity_diameter_list_file='stellar_intens_*, *offax_psf_file='offax_psf.fits'*, *offax_psf_offset_list_file='offax_psf_offset_list.fits*, *sky_trans_file='sky_trans.fits'*, *resamp=4*, *outpath=None*, *outname=''*, *name=None*, *phot_aperture_radius=0.7071067811865476*, *fit_gaussian=False*, *use_phot_aperture_as_min=False*, *overwrite=True*, *units=None*, *to_arcsec=False*)

Process optical system package defined by Stark & Krist to EXOSIMS standard inputs.

Parameters

- **basepath** (*str*) – Full path to directory with all input files.
- **stellar_intensity_file** (*str, optional*) – Filename of stellar intensity PSFs. Defaults to "stellar_intens.fits". If None, no output generated.
- **stellar_intensity_diameter_list_file** (*str, optional*) – Filename of stellar diameters corresponding to stellar_intensity_file. Defaults to "stellar_intens_diam_list.fits". If None, no output generated.
- **offax_psf_file** (*str*) – Filename of off-axis PSFs. Defaults to "offax_psf.fits". If None, no output generated.
- **offax_psf_offset_list_file** (*str, optional*) – Filename of off-axis PSF astrophysical offsets corresponding to offax_psf_file. Defaults to "offax_psf_offset_list.fits". If None, no output generated.
- **sky_trans_file** (*str, optional*) – Filename of sky transmission map. Defualts to "sky_trans.fits" If None, no output generated.
- **resamp** (*float*) – Resampling factor for PSFs. Defaults to 4.
- **outpath** (*str, optional*) – Full path to directory to write results. If None, use basepath. Defaults None
- **outname** (*str*) – Prefix for all output files. If "" (default) then no prefix is added, otherwise all files will be named outname_(descriptor).fits.
- **name** (*str, optional*) – System name (for dictionary output). If None and outname is not "" then use outname. If None and outname is "" then write None. Defaults None.
- **phot_aperture_radius** (*float*) – Photometric aperture radius in native unit of input files (nominally λ/D). Defaults to $\sqrt{2}/2$. If fit_gaussian is True and use_phot_aperture_as_min is True then this value replaces any fit area that is smaller than the area of this value.
- **fit_gaussian** (*bool*) – Fit 2D Gaussians to off-axis PSFs to compute throughput and core area. Default False, in which case the core_area is always the value computed from phot_aperture_radius.
- **use_phot_aperture_as_min** (*bool*) – Only used if fit_gaussian is True. If True, any coputed core area values that are smaller than the area of the phot_aperture_radius are replaced with that value. Defaults False

- **overwrite** (*bool*) – Overwrite output files if they exist. Defaults True. If False, throws error.
- **units** (*str, optional*) – If set, overwrite any UNITS header keyword from the original headers with this value.
- **to_arcsec** (*bool*) – If True, convert all values going into the JSON script to arcseconds. Defaults False.

Returns
Starlight Suppression System dictionary describing the generated system

Return type
dict

Note: The default expectation is that all 5 input files will be provided and processed. However, any of these can be set to None, in which case processing will be skipped for that filetype. Paired files (i.e., `intens` and `intens_diam` will be skipped if either is None).

EXOSIMS.util.process_opticalsys_package.**update_WA_vals**(*IWA*, *OWA*, *WAs*)

Utility method for updating global IWA/OWA

Parameters

- **IWA** (*float, optional*) – Current IWA value or None
- **OWA** (*float, optional*) – Current OWA value or None
- **WAs** (*numpy.ndarray*) – Angular separations of current system

Returns
IWA (float or None) OWA (float or None)

Return type
tuple

EXOSIMS.util.radialfun module

Utilities for radial computations on rectangular data arrays

EXOSIMS.util.radialfun.**circ_aperture**(*im*, *rho*, *center*, *return_sum=False*)

Extract pixels in circular aperture

Parameters

- **im** (*numpy.ndarray*) – The input image. Must be 2-dimensional
- **rho** (*float*) – Radius of aperture (in pixels)
- **center** (*list(float,float)*) – [x,y] pixel coordinates of center of aperture
- **return_sum** (*bool*) – Return sum. Defaults False: returns all pixels in aperture.

Returns
1-dimensional array of pixel values inside aperture

Return type
numpy.ndarray

EXOSIMS.util.radialfun.**com**(*im0*, *fill_val=0*)

Find the center-of-mass centroid of an image

Parameters

- **im** (*numpy.ndarray*) – The input image. Must be 2-dimensional
- **fill_val** (*float*) – Replace any non finite values in the image with this value before re-sampling. Defaults to zero.

Returns

[x,y] pixel coordinates of COM

Return type

list

EXOSIMS.util.radialfun.**fitgaussian**(*im*)

Fit a 2D Gaussian to data

Parameters

im (*numpy.ndarray*) – 2D data array

Returns

a (float):
Amplitude

x0 (float):
Center (mean) x position

y0 (float):
Center (mean) y position

sx (float):
Standard deviation in x

xy (float):
Standard deviation in y

Return type

tuple

EXOSIMS.util.radialfun.**gaussian**(*a*, *x0*, *y0*, *sx*, *sy*)

Gaussian function

Parameters

- **a** (*float*) – Amplitude
- **x0** (*float*) – Center (mean) x position
- **y0** (*float*) – Center (mean) y position
- **sx** (*float*) – Standard deviation in x
- **xy** (*float*) – Standard deviation in y

Returns

Callable lambda function with input x,y returning value of Gaussian at those coordinates

Return type

lambda

EXOSIMS.util.radialfun.**genwindow**(*dims*)

Create a 2D Hann window of the given dimensions

Parameters
dims (*tuple or list*) – 2-element dimensions of window (can be the .shape output of an ndarray)

Returns
Window of dimensions dims.

Return type
numpy.ndarray

EXOSIMS.util.radialfun.**pixel_dists**(*dims*, *center*)

Compute pixel distances from center of an image

Parameters

- **dims** (*tuple(float,float)*) – Image dimensions
- **center** (*list(float,float)*) – [x,y] pixel coordinates of center

Returns
Array of dimension dims with distance from center of each pixel

Return type
numpy.ndarray

EXOSIMS.util.radialfun.**radial_average**(*im*, *center=None*, *nbins=None*)

Compute radial average on an image

Parameters

- **im** (*numpy.ndarray*) – The input image. Must be 2-dimensional
- **center** (*list(float,float), optional*) – [x,y] pixel coordinates of center to compute average about. If None (default) use geometric center of input.
- **nbins** (*int, optional*) – Number of bins to compute average in. If None (default) then set to floor(N/2) where N is the maximum dimension of the input image.

Returns

means (numpy.ndarray):
nbins element array with radial average values

bins (numpy.nadarray):
nbins+1 element array with bin boundaries.

bincents (numpy.ndarray):
nbins elements array with bin midpoints. Equivalent to `(bins[1:] + bins[:-1]) / 2`

Return type
tuple

EXOSIMS.util.radialfun.**resample_image**(*im*, *resamp=2*, *fill_val=0*)

Create a resampled image

Parameters

- **im** (*numpy.ndarray*) – The input image. Must be 2-dimensional
- **resamp** (*float*) – Resampling factor. Must be >=1. Defaults to 2

- **fill_val** (*float*) – Replace any non finite values in the image with this value before resampling. Defaults to zero.

Returns
Resampled image.

Return type
numpy.ndarray

EXOSIMS.util.read_ipcluster_ensemble module

EXOSIMS.util.read_ipcluster_ensemble.**gen_summary**(*run_dir*, *includeUniversePlanetPop=False*)

Create a summary dictionary from an ensemble directory generated by run_ipcluster_ensemble

Parameters

- **run_dir** (*string*) – Absolute path to run directory
- **includeUniversePlanetPop** (*boolean*) – A boolean flag dictating whether to include the universe planet population in the output or just the detected planets (default is false)

Returns
Dictionary of planet properties

Return type
out (dictionary)

EXOSIMS.util.read_ipcluster_ensemble.**read_all**(*run_dir*)

Helper function that reads in all pkl files from an nsemble directory generated by run_ipcluster_ensemble

Parameters
run_dir (*string*) – Absolute path to run directory

Returns
List of all pkl file contents in run_dir

Return type
allres (list)

EXOSIMS.util.statsFun module

EXOSIMS.util.statsFun.**calcM**(*f*, *xMin*, *xMax*)

EXOSIMS.util.statsFun.**eqLogSample**(*f*, *numTest*, *xMin*, *xMax*, *bins=10*)

EXOSIMS.util.statsFun.**simpSample**(*f*, *numTest*, *xMin*, *xMax*, *M=None*, *verb=False*)

Use the rejection sampling method to generate a probability distribution according to the given function f, between some range xMin and xMax. If xMin==xMax, return an array where all values are equal to this value.

EXOSIMS.util.utils module

Various utility methods

EXOSIMS.util.utils.**dictToSortedStr**(*indict: Dict[str, Any]*) → str

Utility method for generating a string representation of a dict with keys sorted alphabetically

Parameters
indict (*dict*) – Dictionary to stringify

Returns
Dictionary contents as string with keys sorted alphabetically

Return type
str

EXOSIMS.util.utils.**genHexStr**(*instr: str*) → str

Utility method generating an md5 hash from any input string

Parameters
instr (*str*) – Input string to hashify

Returns
hash

Return type
str

EXOSIMS.util.vprint module

EXOSIMS.util.vprint.**vprint**(*verbose*)

This function is equivalent to the python print function, with an extra boolean parameter that toggles the print when it is required.

Parameters
verbose (*bool*) – If True (default), the function will print the toprint string. If False, the function won't print anything.

Returns
The new print function with one argument, the string to be printed (toprint)

Return type
f (function)

EXOSIMS.util.waypoint module

EXOSIMS.util.waypoint.**waypoint**(*comps*, *intTimes*, *duration*, *mpath*, *tofile*)

Generates waypoint dictionary for MissionSim

Parameters

- **comps** (*array*) – An array of completeness values for all stars
- **intTimes** (*array*) – An array of predicted integration times for all stars
- **duration** (*int*) – The length of time allowed for the waypoint calculation, defaults to 365
- **mpath** (*string*) – The path to the directory to save a plot in.

- **tofile** (*string*) – Name of the file containing a plot of total completeness over mission time, by default genWaypoint does not create this plot

Returns
Output dictionary containing the number of stars visited, the total completness achieved, and the amount of time spent integrating.

Return type
dict

2.28.2 Submodules

2.28.3 EXOSIMS.MissionSim module

class EXOSIMS.MissionSim.**MissionSim**(*scriptfile=None*, *nopar=False*, *verbose=True*, *logfile=None*, *loglevel='INFO'*, *checkInputs=True*, ***specs*)

Bases: object

Mission Simulation (backbone) class

This class is responsible for instantiating all objects required to carry out a mission simulation.

Parameters

- **scriptfile** (*string*) – Full path to JSON script file. If not set, assumes that dictionary has been passed through specs.
- **nopar** (*bool*) – Ignore any provided ensemble module in the script or specs and force the prototype *SurveyEnsemble*. Defaults True
- **verbose** (*bool*) – Input to *vprint()*, toggling verbosity of print statements. Defaults True.
- **logfile** (*str of None*) – Path to the log file. If None, logging is turned off. If supplied but empty string (''), a temporary file is generated.
- **loglevel** (*str*) – The level of log, defaults to 'INFO'. Valid levels are: CRITICAL, ERROR, WARNING, INFO, DEBUG (case sensitive).
- **checkInputs** (*bool*) – Validate inputs against selected modules. Defaults True.
- ****specs** (*dict*) – *Input Specification*

StarCatalog

StarCatalog class object (only retained if keepStarCatalog is True)

Type
StarCatalog module

PlanetPopulation

PlanetPopulation class object

Type
PlanetPopulation module

PlanetPhysicalModel

PlanetPhysicalModel class object

Type
PlanetPhysicalModel module

`OpticalSystem`
: OpticalSystem class object

  Type
  : OpticalSystem module

`ZodiacalLight`
: ZodiacalLight class object

  Type
  : ZodiacalLight module

`BackgroundSources`
: Background Source class object

  Type
  : BackgroundSources module

`PostProcessing`
: PostProcessing class object

  Type
  : PostProcessing module

`Completeness`
: Completeness class object

  Type
  : Completeness module

`TargetList`
: TargetList class object

  Type
  : TargetList module

`SimulatedUniverse`
: SimulatedUniverse class object

  Type
  : SimulatedUniverse module

`Observatory`
: Observatory class object

  Type
  : Observatory module

`TimeKeeping`
: TimeKeeping class object

  Type
  : TimeKeeping module

`SurveySimulation`
: SurveySimulation class object

  Type
  : SurveySimulation module

SurveyEnsemble

SurveyEnsemble class object

Type
SurveyEnsemble module

modules

Dictionary of all modules, except StarCatalog

Type
dict

verbose

Boolean used to create the vprint function, equivalent to the python print function with an extra verbose toggle parameter (True by default). The vprint function can be accessed by all modules from EXOSIMS.util.vprint.

Type
bool

seed

Number used to seed the NumPy generator. Generated randomly by default.

Type
int

logfile

Path to the log file. If None, logging is turned off. If supplied but empty string (''), a temporary file is generated.

Type
str

loglevel

The level of log, defaults to 'INFO'. Valid levels are: CRITICAL, ERROR, WARNING, INFO, DEBUG (case sensitive).

Type
str

DRM2array(*key*, *DRM=None*)

Creates an array corresponding to one element of the DRM dictionary.

Parameters

- **key** (*str*) – Name of an element of the DRM dictionary
- **DRM** (*list(dict)*) – Design Reference Mission, contains the results of a survey simulation

Returns
Array containing all the DRM values of the selected element

Return type
ndarray or *Quantity*(*ndarray*)

checkScript(*scriptfile*, *prettyprint=False*, *tofile=None*)

Calls CheckScript and checks the script file against the mission outspec.

Parameters

- **scriptfile** (*str*) – The path to the scriptfile being used by the sim
- **prettyprint** (*bool*) – Outputs the results of Checkscript in a readable format.

- **tofile** (*str*) – Name of the file containing all output specifications (outspecs). Default to None.

Returns
Output string containing the results of the check.

Return type
str

check_ioscripts() → None

Collect all input and output scripts against selected module inits and report and discrepancies.

filter_status(*key*, *status*, *DRM=None*, *obsMode=None*)

Finds the values of one DRM element, corresponding to a status value, for detection or characterization.

Parameters

- **key** (*string*) – Name of an element of the DRM dictionary
- **status** (*integer*) – Status value for detection or characterization
- **DRM** (*list of dicts*) – Design Reference Mission, contains the results of a survey simulation
- **obsMode** (*string*) – Observing mode type ('det' or 'char')

Returns
Array containing all the DRM values of the selected element, and filtered by the value of the corresponding status array

Return type
elemStat (ndarray / astropy Quantity array)

genOutSpec(*tofile: Optional[str] = None*, *modnames: bool = False*) → Dict[str, Any]

Join all _outspec dicts from all modules into one output dict and optionally write out to JSON file on disk.

Parameters

- **tofile** (*str*) – Name of the file containing all output specifications (outspecs). Defaults to None.
- **modnames** (*bool*) – If True, populate outspec dictionary with the module it originated from, instead of the actual value of the keyword. Defaults False.

Returns
Dictionary containing the full *Input Specification*, including all filled-in default values. Combination of all individual module _outspec attributes.

Return type
dict

genWaypoint(*targetlist=[]*, *duration=365*, *tofile=None*, *charmode=False*)

generates a ballpark estimate of the expected number of star visits and the total completeness of these visits for a given mission duration

Parameters

- **duration** (*int*) – The length of time allowed for the waypoint calculation, defaults to 365
- **tofile** (*str*) – Name of the file containing a plot of total completeness over mission time, by default genWaypoint does not create this plot
- **charmode** (*bool*) – Run the waypoint calculation using either the char mode instead of the det mode

Returns
Output dictionary containing the number of stars visited, the total completeness achieved, and the amount of time spent integrating.

Return type
dict

get_logger(*logfile*, *loglevel*)

Set up logging object so other modules can use logging.info(), logging.warning, etc.

Parameters

- **logfile** (*string*) – Path to the log file. If None, logging is turned off. If supplied but empty string (''), a temporary file is generated.
- **loglevel** (*string*) – The level of log, defaults to 'INFO'. Valid levels are: CRITICAL, ERROR, WARNING, INFO, DEBUG (case sensitive).

Returns
Mission Simulation logger.

Return type
logger (logging object)

reset_sim(*genNewPlanets=True*, *rewindPlanets=True*, *seed=None*)

Convenience method that simply calls the SurveySimulation reset_sim method.

run_ensemble(*nb_run_sim*, *run_one=None*, *genNewPlanets=True*, *rewindPlanets=True*, *kwargs={}*)

Convenience method that simply calls the SurveyEnsemble run_ensemble method.

run_sim()

Convenience method that simply calls the SurveySimulation run_sim method.

2.28.4 EXOSIMS.e2eTests module

End to End Test Suite for EXOSIMS

Run as:
>python e2eTests.py

This code will sequentially execute all script files found in: `EXOSIMS_ROOT/EXOSIMS/Scripts/TestScripts` and print a summary of the results. A script execution includes instantiating a *MissionSim* object using the script, running a simulation via *run_sim()*, resetting the simulation using *reset_sim()*, and finally re-running the simulation a second time. Possible outcomes for each test are:

PASS

FAIL - Instantiation

FAIL - Execution

FAIL - Reset

EXOSIMS.e2eTests.**run_e2e_tests**()

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[Agol2007] Agol, E. (2007) "Rounding up the wanderers: optimizing coronagraphic searches for extrasolar planets", Monthly Notices of the Royal Astronomical Society, 374(1271), http://dx.doi.org/10.1111/j.1365-2966.2006.11232.x

[Apellaniz2006] Maíz Apellániz, J. (2006) "A Recalibration of Optical Photometry: Tycho-2, Strömgren, and Johnson Systems", The Astronomical Journal, 131, 1184, http://dx.doi.org/10.1086/499158

[Ballesteros2012] Ballesteros, F. J. (2012) "New insights into black bodies", EPL (Europhysics Letters), 97(34008), http://dx.doi.org/10.1209/0295-5075/97/34008

[Bessell1983] Bessell, M. S. (1983) "VRI photometry : an addendum.", Publications of the Astronomical Society of the Pacific, 95(480) http://dx.doi.org/10.1086/131196

[Bessell1988] Bessell, M. S., & Brett, J. M. (1988) "JHKLM Photometry: Standard Systems, Passbands, and Intrinsic Colors", Publications of the Astronomical Society of the Pacific, 100(1134), http://dx.doi.org/10.1086/132281

[Boyajian2013] Boyajian, T. S., von Braun, K., van Belle, G., Farrington, C., Schaefer, G., Jones, J., White, R., McAlister, H. A., ten Brummelaar, T. A., Ridgway, S., Gies, D., Sturmann, L., Sturmann, J., Turner, N. H., Goldfinger, P. J., & Vargas, N. (2013) "Stellar Diameters and Temperatures. III. Main-sequence A, F, G, and K Stars: Additional High-precision Measurements and Empirical Relations", The Astrophysical Journal, 771(40) http://dx.doi.org/10.1088/0004-637X/771/1/40

[Boyajian2014] Boyajian, T. S., van Belle, G., & von Braun, K. (2014) "Stellar Diameters and Temperatures. IV. Predicting Stellar Angular Diameters", The Astronomical Journal, 147(47), http://dx.doi.org/10.1088/0004-6256/147/3/47

[Brown2004] Brown, R. A. (2004) "New Information from Radial Velocity Data Sets", The Astrophysical Journal, 610(1079), http://dx.doi.org/10.1086/421896

[Brown2005] Brown, R. A. (2005) "Single-Visit Photometric and Obscurational Completeness", The Astrophysical Journal, 624(1010), http://dx.doi.org/10.1086/429124

[Chen2016] Chen, J., & Kipping, D. (2017) "Probabilistic Forecasting of the Masses and Radii of Other Worlds", The Astrophysical Journal, 834(17), http://dx.doi.org/10.3847/1538-4357/834/1/17

[HanburyBrown1974] Hanbury Brown, R., Davis, J., Lake, R. J. W., & Thompson, R. J. (1974) "The effects of limb darkening on measurements of angular size with an intensity interferometer", Monthly Notices of the Royal Astronomical Society, 167(475), http://dx.doi.org/10.1093/mnras/167.3.475

[Henry1993] Henry, T. J., & McCarthy, D. W. (1993) "The Mass-Luminosity Relation for Stars of Mass 1.0 to 0.08M(solar)", The Astronomical Journal, 106(773), http://dx.doi.org/10.1086/116685

[Garrett2016] Garrett, D., & Savransky, D. (2016) "Analytical Formulation of the Single-visit Completeness Joint Probability Density Function", The Astrophysical Journal, 828(20), http://dx.doi.org/10.3847/0004-637X/828/1/20

[Kasdin2006] Kasdin, N. J., & Braems, I. (2006) "Linear and Bayesian Planet Detection Algorithms for the Terrestrial Planet Finder", The Astrophysical Journal, 646(1260), http://dx.doi.org/10.1086/505017

[Keithly2020] Keithly, D. R., Savransky, D., Garrett, D., Delacroix, C., & Soto, G. (2020) "Optimal scheduling of exoplanet direct imaging single-visit observations of a blind search survey", Journal of Astronomical Telescopes, Instruments, and Systems, 6(027001), http://dx.doi.org/10.1117/1.JATIS.6.2.027001

[Keithly2021] Keithly, D. R., & Savransky, D. (2021) "The Solar System as an Exosystem: Planet Confusion", The Astrophysical Journal, 919(L11), http://dx.doi.org/10.3847/2041-8213/ac20cf

[Keithly2021b] Keithly, D., Savransky, D., & Spohn, C. (2021) "Integration time adjusted completeness", Journal of Astronomical Telescopes, Instruments, and Systems, 7(037002), http://dx.doi.org/10.1117/1.JATIS.7.3.037002

[Leinert1998] Leinert, C., Bowyer, S., Haikala, L. K., Hanner, M. S., Hauser, M. G., Levasseur-Regourd, A.-C., Mann, I., Mattila, K., Reach, W. T., Schlosser, W., Staude, H. J., Toller, G. N., Weiland, J. L., Weinberg, J. L., & Witt, A. N. (1998) "The 1997 reference of diffuse night sky brightness", Astronomy and Astrophysics Supplement Series, 127(1) http://dx.doi.org/10.1051/aas:1998105

[Nemati2014] Nemati, B. (2014) "Detector selection for the WFIRST-AFTA coronagraph integral field spectrograph", Space Telescopes and Instrumentation 2014: Optical, Infrared, and Millimeter Wave, 9143(91430Q), http://dx.doi.org/10.1117/12.2060321

[Rieke2008] Rieke, G. H., Blaylock, M., Decin, L., Engelbracht, C., Ogle, P., Avrett, E., Carpenter, J., Cutri, R. M., Armus, L., Gordon, K., Gray, R. O., Hinz, J., Su, K., and Willmer, C. N. A. (2008) Absolute Physical Calibration in the Infrared, AJ 135(2245), http://dx.doi.org/10.1088/0004-6256/135/6/2245

[Stark2014] Stark, C. C., Roberge, A., Mandell, A., & Robinson, T. D. (2014) "Maximizing the ExoEarth Candidate Yield from a Future Direct Imaging Mission", The Astrophysical Journal, 795(122), http://dx.doi.org/10.1088/0004-637X/795/2/122

[Stark2015] Stark, C. C., Roberge, A., Mandell, A., Clampin, M., Domagal-Goldman, S. D., McElwain, M. W., & Stapelfeldt, K. R. (2015) "Lower Limits on Aperture Size for an ExoEarth Detecting Coronagraphic Mission", The Astrophysical Journal, 808(149) http://dx.doi.org/10.1088/0004-637X/808/2/149

[StarkKrist2019] Stack, C. C. and Krist, J. (2019) "Standardized Coronagraph Parameters for Input into Yield Calculation", https://starkspace.com/yield_standards.pdf

[Savransky2010] Savransky, D., Kasdin, N. J., & Cady, E. (2010) "Analyzing the Designs of Planet-Finding Missions", Publications of the Astronomical Society of the Pacific, 122(401), http://dx.doi.org/10.1086/652181

[Savransky2015] Savransky, D., and Garrett, D., (2015) "WFIRST-AFTA coronagraph science yield modeling with EXOSIMS", JATIS 2(1); http://dx.doi.org/10.1117/1.JATIS.2.1.011006

[Sobolev1975] Sobolev, V. V. (1975) Light Scattering in Planetary Atmospheres, Pergamon Press: New York

[Traub2016] Traub, W. A., Breckinridge, J., Greene, T. P., Guyon, O., Jeremy Kasdin, N., & Macintosh, B. (2016) "Science yield estimate with the Wide-Field Infrared Survey Telescope coronagraph", Journal of Astronomical Telescopes, Instruments, and Systems, 2(011020), http://dx.doi.org/10.1117/1.JATIS.2.1.011020

PYTHON MODULE INDEX

e

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

N

P

Q

R

S

T

U

V

W

Z