Exoplanet Open-Source Imaging Mission Simulator (EXOSIMS) Interface Control Document

Daniel Garrett and Dmitry Savransky
Sibley School of Mechanical and Aerospace Engineering
Cornell University
Ithaca, NY 14853

ABSTRACT

This document describes the extensible, modular, open source software framework EXOSIMS. EXOSIMS creates end-to-end simulations of space-based exoplanet imaging missions using stand-alone software modules. These modules are split into input and simulation module groups. The input/output interfaces of each module and interactions of modules with each other are presented to give guidance on mission specific modifications to the EXOSIMS framework. Last Update: December 8, 2015

CONTENTS

Nomenclature

EXOSIMS Exoplanet Open-Source Imaging Mission Simulator
ICD Interface Control Document
MJD Modified Julian Day

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 imager can be almost as complicated as designing the mission. It is challenging to compare science simulation results and systematically test the effects of changing one aspect of the instrument or mission design.

EXOSIMS (Exoplanet Open-Source Imaging Mission Simulator) addresses this problem by generating ensembles of mission simulations for exoplanet direct imaging missions to estimate science yields. 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. This document describes the required input/output interfaces for the stand-alone modules to enable this flexibility.

1.1 Purpose and Scope

This Interface Control Document (ICD) provides an overview of the software framework of EXOSIMS and some details on its component parts. As the software is intended to be highly reconfigurable, operational aspects of the code are emphasized over implementational details. Specific examples are taken from the coronagraphic instrument under development for WFIRST-AFTA. The data inputs and outputs of each module are described. Following these guidelines will allow the code to be updated to accommodate new mission designs.

This ICD defines the input/output of each module and the interfaces between modules of the code. This document is intended to guide mission planners and instrument designers in the development of specific modules for new mission designs.

2 Overview

The terminology used to describe the software implementation is loosely based on the object-oriented Python framework upon which EXOSIMS is built. 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 of the prototype, or an instance of this object 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 with the larger framework, and thus allows flexibility and adaptability.

The overall framework of EXOSIMS is depicted in Fig. 1 where the component stand-alone software modules are classified as input modules and simulation modules. The input modules include the Optical System, Star Catalog, Planet Population, Observatory, Planet Physical Model, Time Keeping, and Post-Processing modules. These modules contain specific mission design parameters. The simulation modules include Target List, Simulated Universe, Survey Simulation, and Survey Ensemble modules. The simulation modules take information contained in the input 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. They are only constrained by their input and output specification contained in this document.

Figures 2 and 3 show schematic representations of the three different apsects of a module, for the Star Catalog and Observatory modules, respectively. Every module has a specific prototype that sets the input/output structure of the module and encodes any common functionality for all module class implementations. The various implementations inherit the prototype and perform whatever processing is necessary, limited only by the preset input/output scheme. 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 in exactly the same way in the downstream code, regardless of what implementation they are instances of, due to the strict interface defined in the class prototypes.

For the input modules, the input specification is much more loosely defined than the output specification, as different implementations may draw data from a wide variety of sources. For example, the star catalog 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 simulation modules, is entirely fixed so as to allow for generic use of all module objects in the simulation.

3 Global Specifications

Common references (units, frames of reference, etc.) are required to ensure interoperability between the modules of EXOSIM. All of the references listed below must be followed.

Common Epoch

J2000

Common Reference Frame

Heliocentric Equatorial (HE)

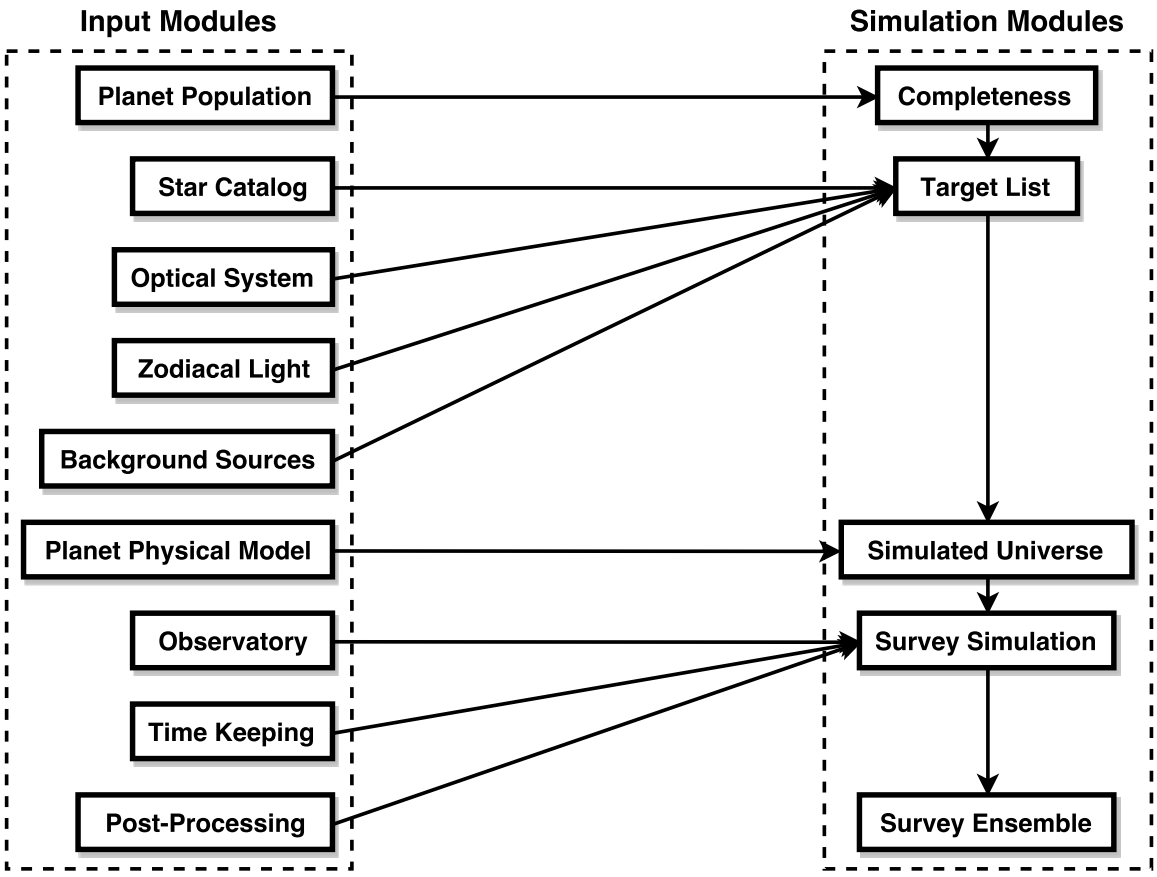

Fig. 1. EXOSIMS modules. Each box represents a component software module that interacts with other modules as indicated by the arrows. The simulation modules pass all input modules along with their own output. Thus, the Survey Ensemble module has access to all of the input modules and all of the upstream simulation modules.

3.1 Python Packages

EXOSIMS is an open source platform. As such, packages and modules may be imported and used for calculations within any of the stand-alone modules. The following commonly used Python packages are used for the WFIRST-specific implementation of EXOSIMS:

```
astropy
    astropy.constants
    astropy.coordinates
    astropy.time
    astropy.units
copy
importlib
numpy
    numpy.linalg
os
    os.path
pickle/cPickle
scipy
    scipy.io
    scipy.special
    scipy.interpolate
```

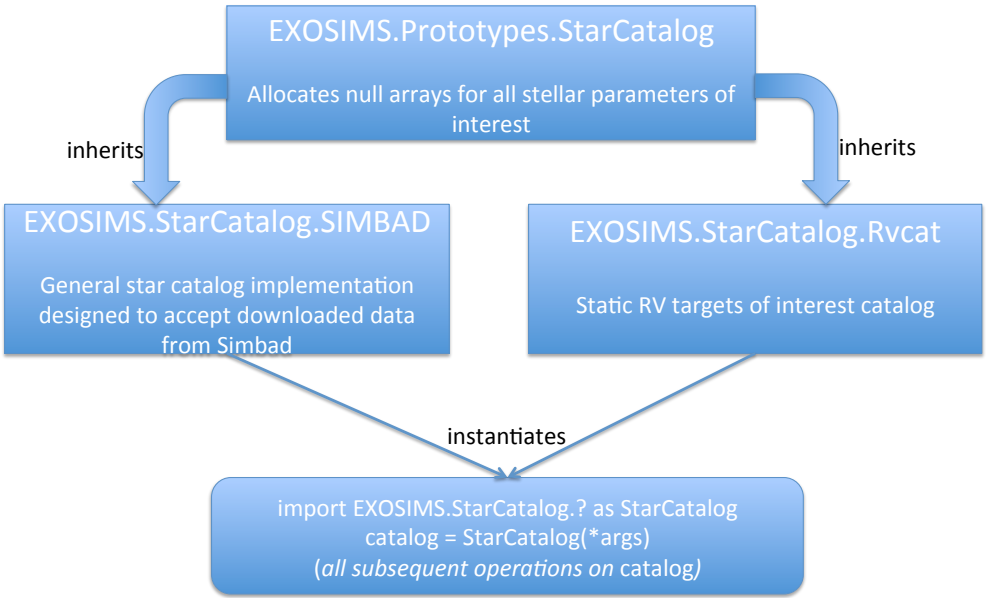

Fig. 2. Schematic of a sample implementation for the three module layers for the Star Catalog module. The Star Catalog prototype (top row) is immutable, specifies the input/output structure of the module along with all common functionality, and is inherited by all Star Catalog class implementations (middle 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 in the simulation (bottom row) is an instance of one of these classes, and can be used in exactly the same way in the rest of the code due to the common input/output scheme.

3.2 Coding Conventions

In order to allow for flexibility in using alternate or user-generated module implementations, the only requirement on any module is that it inherits (either directly or by inheriting another module implementation that inherits the prototype) the appropriate prototype.

3.2.1 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 return `Observatory`.

3.2.2 Callable Attributes

Certain module attributes must 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 optical system module. 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 even more simply:

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

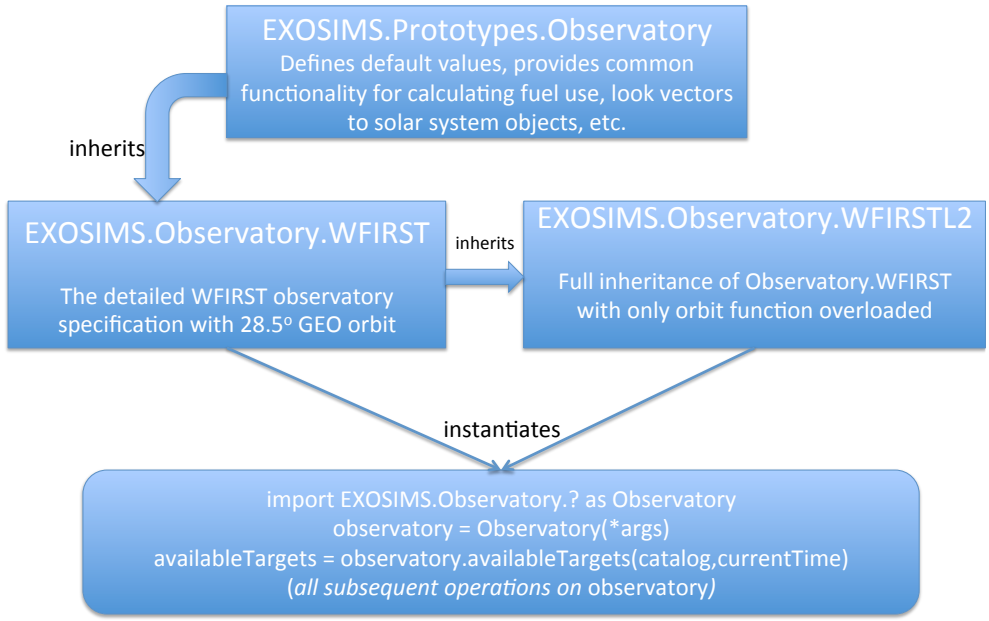

Fig. 3. Schematic of a sample implementation for the three module layers for the Observatory module. The Observatory prototype (top row) is immutable, specifies the input/output structure of the module along with all common functionality, and is inherited by all Observatory class implementations (middle 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 in the simulation (bottom row) is an instance of one of these classes, and can be used in exactly the same way in the rest of the code due to the common input/output scheme.

4 Backbone

By default, the simulation execution will be performed via the backbone. This will consist of a limited set of functions that will primarily be tasked with parsing the input specification described below, and then creating the specified instances of each of the framework modules, detailed in §5. The backbone functionality will primarily be implemented in the MissionSimulation class, whose constructor will take the input script file (§4.1) and generate instances of all module objects, including the SurveySimulation (S6.4) and SurveyEnsemble modules, which will contain the functions to run the survey simulations. Any mission-specific execution variations will be introduced by method overloading in the inherited survey simulation implementation. Figure 4 provides a graphical description of the instantiation order of all module objects.

A simulation specification is a single JSON-formatted (`http://json.org/`) file that encodes user-settable parameters and module names. The backbone will contain a reference specification with *all* parameters and modules set via defaults in the constructors of each of the modules. In the initial parsing of the user-supplied specification, it will be merged with the reference specification such that any fields not set by the user will be assigned to their reference (default) values.

The backbone will also contain a specification parser that will check specification files for internal consistency. For example, if modules carry mutual dependencies, the specification parser will return an error if these are not met for a given specification. Similarly, if modules are selected with optional top level inputs, warnings will be generated if these are not set in the same specification files.

In addition to the specification parser, the backbone will contain a method for comparing two specification files and returning the difference between them. Assuming that the files specify all user-settable values, this will be equivalent to simply performing a `diff` operation on any POSIX system. The backbone diff function will add in the capability to automatically fill in unset values with their defaults. For every simulation (or ensemble), an output specification will be written to disk along with the simulation results with all defaults used filled in.

The backbone will also contain an interactive function to help users generate specification files via a series of questions.

4.1 Specification Format

The JSON specification file will contain a series of objects with members enumerating various user-settable parameters, top-level members for universal settings (such as the mission lifetime) and arrays of objects for multiple related specifications, such as starlight suppression systems and science instruments. The final array will contain module names (or paths on disk) for all modules.

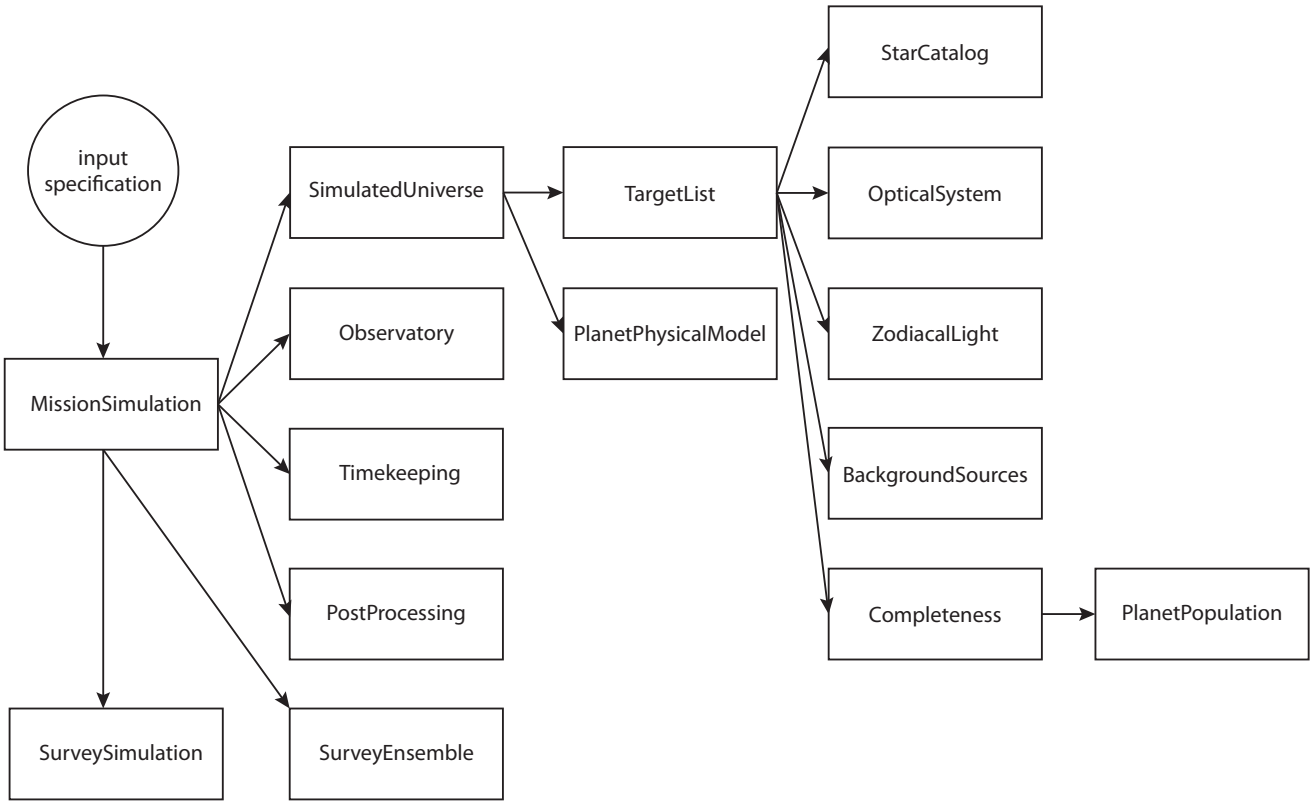

Fig. 4. Schematic depiction of the instantiation path of all simulation and input modules. The entry point to the backbone is the construction of a MissionSimulation object, which causes the instantiation of all other module objects. All objects are instantiated in the order shown here, with SurveySimulation and SurveyEnsemble constructed last. 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 at the end of construction, the MissionSimulation object has direct access to all other modules as its attributes.

```
{
  "universalParam1": value,
  "universalParam2": value,
  ...
  "NstarlightSuppressionSystems":2,
  "starlightSupressionSystems": [
    {
      "starlightSuppressionSystemNumber": 1,
      "type": "external",
      "detectionTimeMultiplier": value,
      "characterizationTimeMultiplier": value,
      "occulterDiameter": value,
      "NocculterDistances": 2,
      "occulterDistances: [
        {
         "occulterDistanceNumber": 1,
         "occulterDistance": value,
         "occulterBlueEdge": value,
         "occulterRedEdge": value,
         "IWA": value,
         "OWA": value,
         "PSFfile": "/data/mdo1_psf.fits",
         "throughputFile": "/data/mdo1_thru.fits",
         "contrastFile":  "/data1/mdo1_contrast.fits"
```

```
      },
      {
       "occulterDistanceNumber": 2,
       "occulterDistance": value,
       "occulterBlueEdge": value,
       "occulterRedEdge": value,
       "IWA": value,
       "OWA": value,
       "PSFfile": "/data/mdo1_psf.fits",
       "throughputFile": "/data/mdo1_thru.fits",
       "contrastFile":  "/data1/mdo1_contrast.fits"
      }
    ],
    "occulterWetMass": value,
    "occulterDryMass": value,
  },
  {
    "starlightSuppressionSystemNumber": 2,
    "type": "internal",
    "detectionTimeMultiplier": value,
    "characterizationTimeMultiplier": value,
    "IWA": value,
    "OWA": value,
    "PSFfile": "/data/coron1_psf.fits",
    "throughputFile": "/data/coron1_thru.fits",
    "contrastFile":  "/data1/coron1_contrast.fits"
  }
],
"NscienceInstruments": 2
"scienceInstruments": [
  {
    "scienceInstrumentNumber": 1,
    "type": "imager-EMCCD",
    "cic": value,
    "darkRate": value,
    "QE": value/filename
  },
  {
    "scienceInstrumentNumber": 2,
    "type": "IFS-CCD",
    "readNoise": value,
    "darkRate": value,
    "QE": value/filename,
    "readRate": value
  }
],
modules: {
  "PlanetPopulation": "HZEarthTwins",
  "StarCatalog": "exocat3",
  "OpticalSystem": "hybridOpticalSystem1",
  "ZodiacalLight": "10xSolZodi",
  "BackgroundSources": "besanconModel",
  "PlanetPhysicalModel": "fortneyPlanets",
  "Observatory": "WFIRSTGeo",
  "TimeKeeping": "UTCtime",
  "PostProcessing": "KLIPpost",
  "Completeness": "BrownCompleteness",
  "TargetList": "WFIRSTtargets",
```

```
    "SimulatedUniverse": "simUniverse1",
    "SurveySimulation": "backbone1",
    "SurveyEnsemble": "localIpythonEnsemble"
  }
}
```

4.2 Modules Specification

The final array in the input specification (`modules`) is a list of all the modules that define a particular simulation. This is the only part of the specification that will not be filled in by default if a value is missing - each module must be explicitly specified. The order of the modules in the list is arbitrary, so long as they are all present.

If the module implementations are in the appropriate subfolder in the EXOSIMS tree, then they can be specified by the module name. However, if you wish to use an implemented module outside of the EXOSISM directory, then you need to specify it via its full path in the input specification.

All modules, regardless of where they are stored on disk must inherit the appropriate prototype.

4.3 Universal Parameters

These parameters apply to all simulations, and are described in detail in their specific module definitions:

`missionLifetime`	The total mission lifetime in years. When the mission time is equal or greater to this value, the mission simulation stops.
`missionPortion`	The portion of the mission dedicated to exoplanet science, given as a value between 0 and 1. When the total integration time plus observation overhead time is equal to the missionLifetime $\times$ missionPortion the mission simulation stops.
`lambda`	Central or detection wavelength.
`deltaLambda`	Detection bandwidth.
`shapeFac`	Telescope aperture shape factor.

5 Input Modules

The input modules include Planet Population, Star Catalog, Optical System, Zodiacal Light, Background Sources, Planet Physical Model, Observatory, Time Keeping, and Post-Processing. These modules encode and/or generate all of the information necessary to perform mission simulations. The specific mission design determines the functionality of each input module, while inputs and outputs of these modules remain the same (in terms of data type and variable representations). This section defines the functionality, major tasks, input, output, and interface of each of these modules.

5.1 Planet Population

The Planet Population module encodes the density functions of all required planetary parameters, both physical and orbital. These include semi-major axis, eccentricity, orbital orientation, radius, mass, and geometric albedo (see 5.1.2). Certain parameter models may be empirically derived while others may come from analyses of observational surveys. This module also encodes the limits on all parameters to be used for sampling the distributions and determining derived cutoff values such as the maximum target distance for a given instrument's IWA.

The coordinate system of the simulated exosystems is defined as in Figure 5. The observer looks at the target star along the $\mathbf{s}_3$ axis, located at a distance $-d\mathbf{s}$ from the target at the time of observation. The argument of periapse, inclination, and longitude of the ascending node (ω, I, Ω) are defined as a 3-1-3 rotation about the unit vectors defining the $\mathcal{S}$ reference frame. This rotation defines the standard Equinoctial reference frame $(\hat{\mathbf{e}}, \hat{\mathbf{q}}, \hat{\mathbf{h}})$, with the true anomaly (ν) measured from $\hat{\mathbf{e}}$). The planet-star orbital radius vector $\mathbf{r}_{P/S}$ is projected into the $\mathbf{s}_1, \mathbf{s}_2$ plane as the projected separation vector $\mathbf{s}$, with magnitude s, and the phase (star-planet-observer) angle (β) is closely approximated by the angle between $\mathbf{r}_{P/s}$ and its projection onto $\mathbf{s}_3$.

The Planet Population module does not model the physics of planetary orbits or the amount of light reflected or emitted by a given planet, but rather encodes the statistics of planetary occurrence and properties.

5.1.1 Planet Population Object Attribute Initialization Input/Output Description

Inputs

User specification

Information from simulation specification JSON file organized into a Python dictionary. If the below key: value pairs are missing from the dictionary, the Planet Population object attributes will be assigned the default values listed.

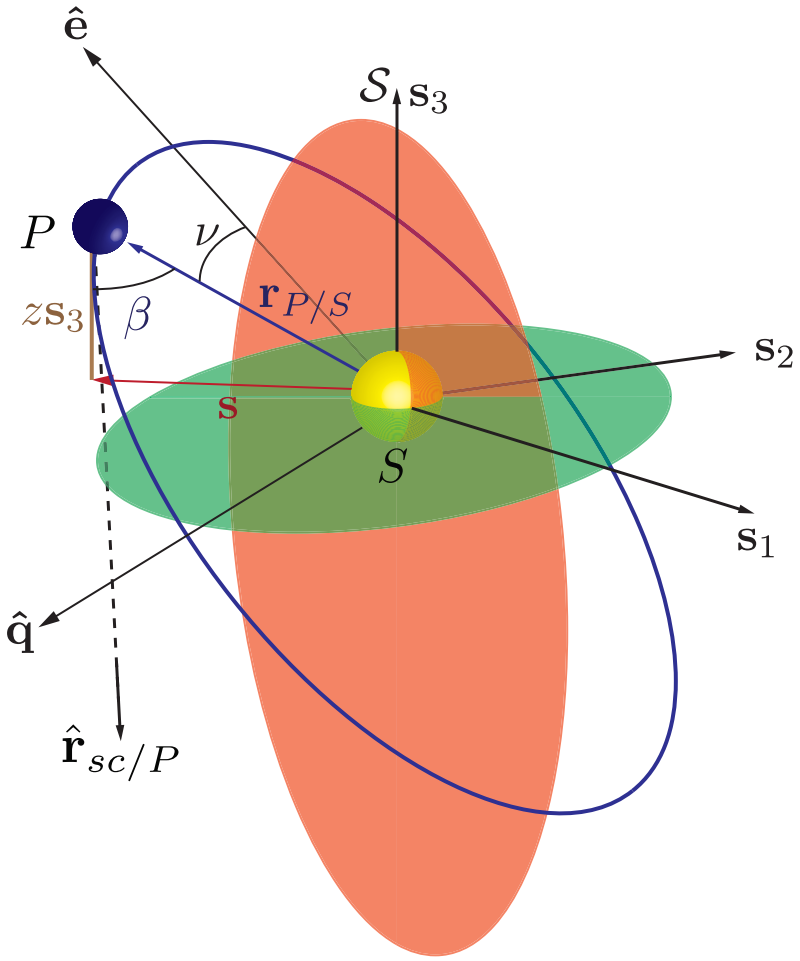

Fig. 5. Definition of reference frames and coordinates of simulated exosystems. The observer lies along the negative $\mathbf{s}_3$ axis so that the observer-star unit vector is $+\mathbf{s}_3$.

specs["a_min"]
Minimum semi-major axis in *AU*. Default value is 0.01.
specs["a_max"]
Maximum semi-major axis in *AU*. Default value is 10.0.
specs["e_min"]
Minimum eccentricity. Default value is 0.001.
specs["e_max"]
Maximum eccentricity. Default value is 0.8.
specs["w_min"]
Minimum argument of perigee in degrees. Default value is 0.
specs["w_max"]
Maximum argument of perigee in degrees. Default value is 360.
specs["O_min"]
Minimum right ascension of the ascending node. Default value is 0.
specs["O_max"]
Maximum right ascension of the ascending node. Default value is 360.
specs["p_min"]
Minimum planetary geometric albedo. Default value is 0.0004.
specs["p_max"]
Maximum planetary geometric albedo. Default value is 0.6.
specs["I_min"]
Minimum inclination in degrees. Default value is 0.
specs["I_max"]

Maximum inclination in degrees. Default value is 180.

specs["R_min"]
Minimum planetary radius in *km*. Default value is 1930.

specs["R_max"]
Maximum planetary radius in *km*. Default value is 145844.

specs["Mp_min"]
Minimum planetary mass in *kg*. Default value is 1.196e+23.

specs["Mp_max"]
Maximum planetary mass in *kg*. Default value is 5.411e+28.

specs["scaleOrbits"]
Boolean where True means planetary orbits are scaled by the square root of stellar luminosity. Default value is False.

Outputs

PlanetPopulation.arange
Semi-major axis range defined as numpy.array([a_min, a_max]) (astropy Quantity object initially set in *AU*)

PlanetPopulation.erange
Eccentricity range defined as numpy.array([e_min, e_max])

PlanetPopulation.wrange
Argument of perigee range defined as numpy.array([w_min, w_max])

PlanetPopulation.Orange
Right ascension of the ascending node range defined as numpy.array([O_min, O_max])

PlanetPopulation.prange
Planetary geometric albedo range defined as numpy.array([p_min, p_max])

PlanetPopulation.Irange
Planetary orbital inclination range defined as numpy.array([I_min, I_max])

PlanetPopulation.Rrange
Planetary radius range defined as numpy.array([R_min, R_max]) (astropy Quantity object or astropy constant objects, e.g., astropy.constant.R_earth initially set in *km*)

PlanetPopulation.Mprange
Planetary mass range defined as numpy.array([Mp_min, Mp_max]) (astropy Quantity object initially set in *kg*)

PlanetPopulation.scaleOrbits
Boolean where True means planetary orbits are scaled by the square root of stellar luminosity

PlanetPopulation.rrange
Planetary orbital radius range defined as numpy.array([r_min, r_max]) derived from PlanetPopulation.arange and PlanetPopulation.erange (astropy Quantity object initially set in *km*)

5.1.2 Planet Population Probability Density Functions

Each of the probability density functions contained in the Planet Population module are implemented as functions. The input to these functions is the quantity of interest and the output is value of the probability density function. These functions are provided so that the Simulated Universe module may sample these quantities from the probability density functions and create a simulated universe of synthetic planets. The required functions are given as follows:

semi_axis
Probability density function for semi-major axis

eccentricity
Probability density function for eccentricity

arg_perigee
Probability density function for argument of perigee

RAAN
Probability density function for right ascension of the ascending node

radius
Probability density function for planetary radius

mass
Probability density function for planetary mass

albedo
Probability density function for planetary geometric albedo

inclination
Probability density function for orbital inclination

5.2 Star Catalog

The Star Catalog module includes detailed information about potential target stars drawn from general databases such as SIMBAD, mission catalogs such as Hipparcos, or from existing curated lists specifically designed for exoplanet imaging missions. Information to be stored, or accessed by this module will include target positions and proper motions at the reference epoch, catalog identifiers (for later cross-referencing), bolometric luminosities, stellar masses, and magnitudes in standard observing bands. Where direct measurements of any value are not available, values are synthesized from ancillary data and empirical relationships, such as color relationships and mass-luminosity relations.

This module does not provide any functionality for picking the specific targets to be observed in any one simulation, nor even for culling targets from the input lists where no observations of a planet could take place. This is done in the Target List module as it requires interactions with the Planet Population (to determine the population of interest), Optical System (to define the capabilities of the instrument), and Observatory (to determine if the view of the target is unobstructed) modules.

5.2.1 Star Catalog Object Attribute Initialization Input/Output Description

The Star Catalog prototype creates empty 1D NumPy ndarrays for each of the output quantities listed below. Specific Star Catalog modules populate the values as appropriate.

Inputs

star catalog information
Information from an external star catalog

Outputs

StarCatalog.Name
1D NumPy ndarray of star names
StarCatalog.Type
1D NumPy ndarray of star types
StarCatalog.Spec
1D NumPy ndarray of spectral types
StarCatalog.parx
1D NumPy ndarray of parallax in milliarcseconds
StarCatalog.Umag
1D NumPy ndarray of U magnitude
StarCatalog.Bmag
1D NumPy ndarray of B magnitude
StarCatalog.Vmag
1D NumPy ndarray of V magnitude
StarCatalog.Rmag
1D NumPy ndarray of R magnitude
StarCatalog.Imag
1D NumPy ndarray of I magnitude
StarCatalog.Jmag
1D NumPy ndarray of J magnitude
StarCatalog.Hmag
1D NumPy ndarray of H magnitude
StarCatalog.Kmag
1D NumPy ndarray of K magnitude
StarCatalog.dist
1D NumPy ndarray of distance in parsecs
StarCatalog.BV
1D NumPy ndarray of B-V Johnson magnitude
StarCatalog.MV
1D NumPy ndarray of absolute V magnitude
StarCatalog.BC
1D NumPy ndarray of bolometric correction

StarCatalog.L
1D NumPy ndarray of stellar luminosity in Solar luminosities
StarCatalog.coords
Astropy SkyCoord object containing list of star positions (e.g., right ascension and declination)
StarCatalog.pmra
1D NumPy ndarray of proper motion in right ascension in milliarcseconds/year
StarCatalog.pmdec
1D NumPy ndarray of proper motion in declination in milliarcseconds/year
StarCatalog.rv
1D NumPy ndarray of radial velocity in kilometers/second
StarCatalog.Binary_Cut
Boolean 1D NumPy ndarray where True is companion star closer than 10 arcseconds

5.3 Optical System

The Optical System module contains all of the necessary information to describe the effects of the telescope and starlight suppression system on the target star and planet wavefronts. This requires encoding the design of both the telescope optics and the specific starlight suppression system, whether it be an internal coronagraph or an external occulter. The encoding can be achieved by specifying Point Spread Functions (PSF) for on- and off-axis sources, along with angular separation and wavelength dependent contrast and throughput definitions. At the opposite level of complexity, the encoded portions of this module may be a description of all of the optical elements between the telescope aperture and the imaging detector, along with a method of propagating an input wavefront to the final image plane. Intermediate implementations can include partial propagations, or collections of static PSFs representing the contributions of various system elements. The encoding of the optical train will allow for the extraction of specific bulk parameters including the instrument inner working angle (IWA), outer working angle (OWA), and mean and max contrast and throughput.

Finally, the Optical System must also include a description of the science instrument. The baseline instrument is assumed to be an imaging spectrometer. The encoding must provide the spatial and wavelength coverage of the instrument as well as sampling for each, along with detector details such as read noise, dark current, and readout cycle.

The Optical System module has two tasks used in simulation. `calc_maxintTime` is called from the Target List module to calculate the maximum integration time for each star in the target list (see 5.3.2). `calc_intTime` is called from the Survey Simulation module to calculate integration times for a target system (see 5.3.3). The inputs and outputs for the Optical System tasks are depicted in Fig. 6.

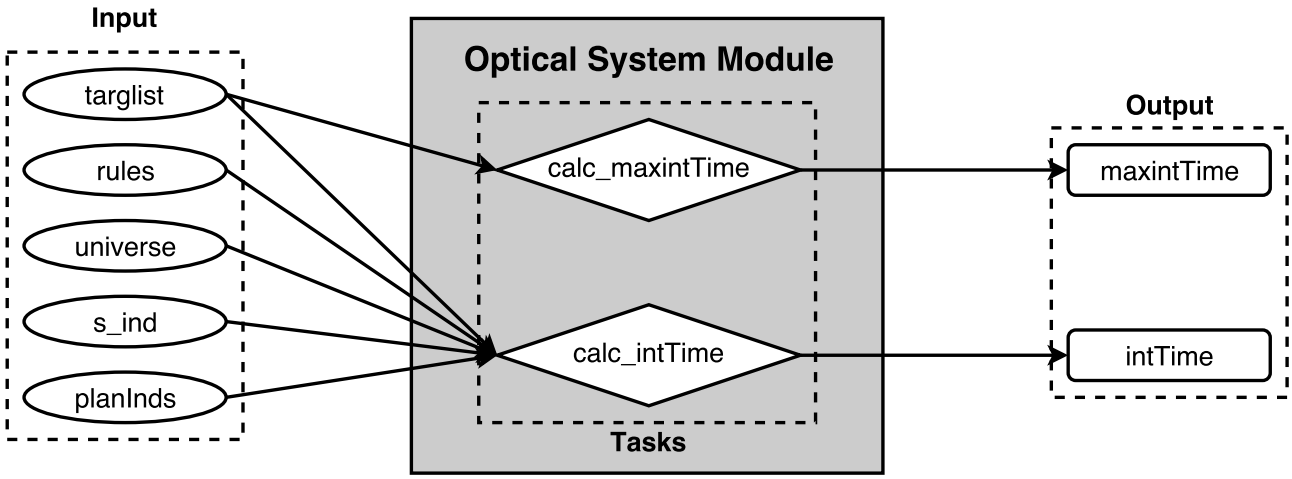

Fig. 6. Depiction of Optical System module tasks including inputs and outputs (see 5.3.2 and 5.3.3).

5.3.1 Optical System Object Attribute Initialization Input/Output Description

The specific set of inputs to this module will vary based on the simulation approach used. Here we define the specification for the case where static PSF(s), derived from external diffraction modeling, are used to describe the system. Note that

some of the inputs are coronagraph or occulter specific, and will be expected based on the "internal" or "external" starlight suppression system keyword, respectively.

Inputs

User specification

Information from simulation specification JSON file organized into a Python dictionary. For multiple systems, there will be an array of dictionaries. If the below key: value pairs are missing from the input specification, the Optical System object attributes will be assigned the default values listed.

specs["lambda"]
: Central detection wavelength in *nm*. Default value is 500.

specs["deltaLambda"]
: Detection bandwidth $\Delta\lambda$ in *nm*. Default value is 100.

specs["shapeFac"]
: Shape factor (also known as fill factor) so that $shapeFac \times diameter^2 = Area$. Default value is $\frac{\pi}{4}$.

specs["pupilArea"]
: Entrance pupil area in m^2. Default value is 4π.

specs["SNchar"]
: Signal to Noise Ratio for characterization. Default value is 11.

specs["pixelArea"]
: Pixel area in m^2. Default value is 1e-10.

specs["focalLength"]
: Focal length in *m*. Default value is 240.

specs["IWA"]
: Inner Working Angle in *arcseconds*. Default value is 0.075.

specs["OWA"]
: Outer Working Angle in *arcseconds*. Set to Inf for no OWA. Default value is Inf.

specs["throughput"]
: Coronagraph throughput: either a scalar for constant throughput, a two-column array for angular separation-dependent throughput, where the first column contains the separations in arcseconds, or a 2D array for angular separation- and wavelength- dependent throughput, where the first column contains the angular separation values in as and the first row contains the wavelengths in nm. The ranges on all parameters must be consistent with the values for the IWA, OWA, lambda and deltaLambda inputs. May be data or FITS filename. Default is scalar 0.5.

specs["dMagLim"]
: Limiting Δmag (difference in magnitude between star and planet). Default value is minimum of contrast.

specs["contrast"]
: Optical system contrast: either a scalar for constant contrast, a two-column array for angular separation-dependent contrast, where the first column contains the separations in arcseconds, or a 2D array for angular separation- and wavelength- dependent contrast, where the first column contains the angular separation values in as and the first row contains the wavelengths in nm. The ranges on all parameters must be consistent with the values for the IWA, OWA, lambda and deltaLambda inputs. May be data or FITS filename. Default is scalar 1e-10.

specs["dr"]
: Detector dark current rate per pixel in units of electrons/second. Default value is 0.001.

specs["sigma_r"]
: Detector read noise in electrons/read. Default value is 3.

specs["t_exp"]
: Exposure time per read in *s*. Default value is 1000.

specs["QE"]
: Detector quantum efficiency: either a scalar for constant QE, or a two-column array for wavelength-dependent QE, where the first column contains the wavelengths in nm. The ranges on all parameters must be consistent with the values for lambda and deltaLambda inputs. May be data or FITS filename. Default is scalar 0.5.

specs["attenuation"]
: Non-coronagraph attenuation, equal to the throughput of the optical system without the coronagraph elements. Default value is 0.57.

specs["PSF"]
: Instrument point spread function. Either a 2D array of a single-PSF, or a 3D array of wavelength-dependent

PSFs. May be data or FITS filename. Default is numpy.ones((3,3)).

specs["PSFsampling"]
Sampling of the PSF in arcsec/pixel. Default value is 10.

specs["specLambda"]
Spectral wavelength of interest in nm. Default value is equal to lambda.

specs["Rspec"]
(Specific to spectrometers) Spectral resolving power defined as $\lambda/\Delta\lambda$. Default value is 70.

specs["optical_oh"]
Optical system overhead time in *days*. Default value is 1 day. This is the (assumed constant) amount of time required to set up the optical system (i.e., dig the dark hole or do fine alignment with the occulter). It is added to every observation, and is separate from the observatory overhead defined in the observatory module, which represents the observatory's settling time. Both overheads are added to the integration time to determine the full duration of each detection observation.

specs["intCutoff"]
Maximum allowed integration time in *days*. Default value is 50. No integrations will be started that would take longer than this value.

specs["detectionTimeMultiplier"]
Duty cycle of a detection observation. If only a single integration is required for the initial detection observation, then this value is 1. Otherwise, it is equal to the number of discrete integrations needed to cover the full field of view (i.e., if a shaped pupil with a dark hole that covers 1/3 of the field of view is used for detection, this value would equal 3).

specs["characterizationTimeMultiplier"]
Characterization duty cycle. If only a single integration is required for the initial detection observation, then this value is 1. Otherwise, it is equal to the number of discrete integrations needed to cover the full wavelength band and all required polarization states. For example, if the band is split into three sub-bands, and there are two polarization states that must be measured, and each of these must be done sequentially, then this value would equal 6. However, if the three sub-bands could be observerd at the same time (e.g., by separate detectors) then the value would by two (for the two polarization states).

specs["occulterDiameter"]
Occulter diameter in *m*. Measured petal tip-to-tip.

specs["NocculterDistances"]
Number of telescope separations the occulter operates over (number of occulter bands). If greater than 1, then the occulter description is an array of dicts.

specs["occulterDistance"]
Telescope-occulter separation in *km*.

specs["occulterBlueEdge"]
Occulter blue end of wavelength band in *nm*.

specs["occulterRedEdge"]
Occulter red end of wavelength band in *nm*.

For all values that may be either scalars or interpolants, in the case where scalar values are given, the optical system module will automatically wrap them in lambda functions so that they become callable (just like the interpolant) but will always return the same value for all arguments. The inputs for interpolants may be filenames with tabulated data, or NumPy ndarrays of argument and data (in that order in rows so that input[0] is the argument and input[1] is the data).

Outputs

OpticalSystem.lambda
Detection wavelength (astropy Quantity object initially set in *nm*)

OpticalSystem.deltaLambda
Detection bandwidth $\Delta\lambda$ (astropy Quantity object initially set in *nm*)

OpticalSystem.shapeFac
Shape factor so that $shapeFac \times diameter^2 = Area$

OpticalSystem.pupilArea
Entrance pupil area (astropy Quantity object initially set in m^2)

OpticalSystem.SNchar
Signal to Noise Ratio for characterization

OpticalSystem.pixelArea
Pixel area (astropy Quantity object initially set in m^2

OpticalSystem.focalLength
Focal length (astropy Quantity object initially set in *m*)
OpticalSystem.IWA
Inner Working Angle (astropy Quantity object initially set in *arcseconds*)
OpticalSystem.OWA
Outer Working Angle (astropy Quantity object initially set in *as*)
OpticalSystem.dMagLim
Limiting Δmag (difference in magnitude between star and planet)
OpticalSystem.throughput
Optical system throughput. Callable object with arguments of wavelength, valid between the lowest and highest wavelengths to be observed, and angular separation, valid between the IWA and OWA. For a fixed throughput, the callable object may always return a single value.
OpticalSystem.contrast
Optical system contrast. Callable object with arguments of wavelength, valid between the lowest and highest wavelengths to be observed, and angular separation, valid between the IWA and OWA. For a fixed contrast, may always return a single value.
OpticalSystem.dr
Detector dark-current rate per pixel (astropy Quantity object initially set in $\frac{1}{s}$)
OpticalSystem.sigma_r
Detector read noise in electrons/read
OpticalSystem.t_exp
Exposure time per read (astropy Quantity object initially set in *s*)
OpticalSystem.QE
Detector quantum efficiency. Callable object with argument of wavelength, valid between the lowest and highest wavelengths to be observed (as defined by lambda and deltaLambda), inclusive, and must assume an argument in nanometers.
OpticalSystem.attenuation
Non-coronagraph attenuation, equal to the throughput of the optical system without the coronagraph elements.
OpticalSystem.PSF
Instrument point spread function. Callable object with arguments of wavelength, valid between the lowest and highest wavelengths to be observed, and angular separation, valid between the IWA and OWA. Returns a 2D NumPy ndarray. The core of the PSF is normalized to 1, so that all throughput terms are combined in the througput variable above.
OpticalSystem.PSFSampling
Sampling of PSF in arcsec/pixel (astropy Quantity object initially set in arcseconds)
OpticalSystem.telescopeKeepout
Telescope keepout angle in degrees
OpticalSystem.specLambda
Spectral wavelength of interest (astropy Quantity object initially set in *nm*)
OpticalSystem.Rspec
Spectral resolving power
OpticalSystem.intCutoff
Maximum allowed integration time (astropy Quantity object initially set in *days*)
OpticalSystem.detectionTimeMultiplier
Duty cycle of a detection observation.
OpticalSystem.characterizationTimeMultiplier
Characterization duty cycle.
OpticalSystem.occulterDiameter
Occulter diameter. (astropy Quantity object initially set in *m*)
OpticalSystem.occulterDistance
Telescope-occulter separation. (astropy Quantity object initially set in *km*)
OpticalSystem.occulterBlueEdge
Occulter blue end of wavelength band. (astropy Quantity object initially set in *nm*)
OpticalSystem.occulterRedEdge
Occulter red end of wavelength band. (astropy Quantity object initially set in *nm*)

5.3.2 calc_maxintTime Task Input/Output Description

The `calc_maxintTime` task calculates the maximum integration time for each star in the target list. This task is called from the Target List module.

Inputs

targlist
Instantiated Target List object from Target List module see 6.2 for definition of available attributes

Output

maxintTime
Maximum integration time for each target star as 1D NumPy ndarray with astropy Quantity units of time attached

5.3.3 calc_intTime Task Input/Output Description

The `calc_intTime` task calculates the integration time required for a specific target system. This task is called from the Survey Simulation module.

Inputs

targlist
Instantiated Target List object from Target List module see 6.2 for definition of available attributes

universe
Instantiated Simulated Universe object from Simulated Universe module see 6.3 for definition of available attributes

s_ind
Index of target star from Target List module

planInds
Index of planets belonging to target star

Output

intTime
Integration time for each of the planets indexed by planInds as 1D NumPy ndarray with astropy Quantity units of time attached

5.4 Zodiacal Light

The Zodiacal Light module contains the `fzodi` task. This task calculates the contribution of both local and exozodiacal light levels for each planet. The inputs and outputs for the Zodiacal Light task are depicted in Fig. 7.

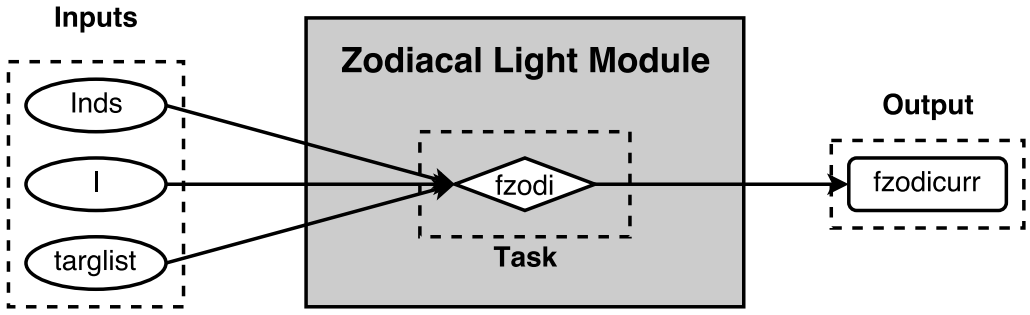

Fig. 7. Depiction of Zodiacal Light module task including inputs and outputs (see 5.4.2).

5.4.1 Zodiacal Light Object Attribute Initialization Input/Output Description

Input

User specification

Information from simulation specification JSON file organized into a Python dictionary. If the below key: value pairs are missing from the dictionary, the Zodiacal Light object attributes will be assigned the default values listed.

specs[“exozodi”]
Exo-zodi level in zodi
specs[“exozodiVar”]
Exo-zodi variation (variance of log-normal distribution)

Output

ZodiacalLight.exozodi

Exo-zodi level in zodi

ZodiacalLight.exozodiVar

Exo-zodi variation (variance of log-normal distribution)

5.4.2 fzodi Task Input/Output Description

The `fzodi` task returns exozodi levels for planetary systems. This functionality is used by the Simulated Universe module.

Inputs

Inds

1D NumPy ndarray of indices mapping planets to stars contained in the target list

I

1D NumPy ndarray of planet inclination in degrees

targlist

Instantiated Target List object from Target List module see 6.2 for description of functionality and attributes

Outputs

fzodicurr

1D NumPy ndarray containing exozodiacal light levels in zodi

5.5 Background Sources

The Background Sources module will provide density of background sources for a given target based on its coordinates and the integration depth. This will be used in the post-processing module to determine false alarms based on confusion. The prototype module has no inputs and only a single function: `dNbackground`.

5.5.1 dNbackground

Inputs

coords

Astropy SkyCoord object containing list of star positions (e.g., right ascension and declination), typically passed from target list.

intDepths

Array-like (list or ndarray) of floating point values equal to absolute magnitude (in the detection band) of dark hole to be produced for each target. Dimension must match the length of the coords input.

Outputs

dN

Numpy ndarray matching size of inputs, containing number densities of background sources for given targets in number per square arcminute.

5.6 Planet Physical Model NEEDS UPDATING

The Planet Physical Model module contains models of the light emitted or reflected by planets in the wavelength bands under investigation by the current mission simulation. It takes as inputs the physical quantities sampled from the distributions in the Planet Population module and generates synthetic spectra (or band photometry, as appropriate). The specific implementation of this module can vary greatly, and can be based on any of the many available planetary geometric albedo, spectra and phase curve models.

Planet Physical Model TASKS:

5.7 Observatory

The Observatory module contains all of the information specific to the space-based observatory not included in the Optical System module. The module has two main tasks: `orbit` and `keepout`, which are implemented as functions within the module.

The observatory orbit plays a key role in determining which of the target stars may be observed for planet finding at a specific time during the mission lifetime. The Observatory module's `orbit` task takes the current mission time as input and outputs the observatory's position vector. The position vector is standardized throughout the modules to be referenced to a heliocentric equatorial frame at the J2000 epoch. The observatory's position vector is used in the `keepout` task and Target List module to determine which of the stars are observable at the current mission time.

The `keepout` task determines which target stars are observable at a specific time during the mission simulation and which are unobservable due to bright objects within the field of view such as the sun, moon, and solar system planets. The keepout volume is determined by the specific design of the observatory and, in certain cases, by the starlight suppression system. The `keepout` task takes the current mission time and Star Catalog or Target List module output as inputs and outputs a list of the target stars which are observable at the current time. It constructs position vectors of the target stars and bright objects which may interfere with observations with respect to the observatory. These position vectors are used to determine if bright objects are in the field of view for each of the potential stars under exoplanet finding observation. If there are no bright objects obstructing the view of the target star, it becomes a candidate for observation in the Survey Simulation module. The solar keepout is typically encoded as allowable angle ranges for the spacecraft-star unit vector as measured from the spacecraft-sun vector.

In addition to these tasks, the observatory definition can also encode finite resources used by the observatory throughout the mission. The most important of these is the fuel used for stationkeeping and repointing, especially in the case of occulters which must move significant distances between observations. Other considerations could include the use of other volatiles such as cryogens for cooled instruments, which tend to deplete solely as a function of mission time. This module also allows for detailed investigations of the effects of orbital design on the science yield, e.g., comparing the baseline geosynchronous 28.5° inclined orbit for WFIRST-AFTA with an L2 halo orbit proposed for other exoplanet imaging mission concepts.

The inputs, outputs, and updated attributes of the required Observatory module tasks are depicted in Fig. 8.

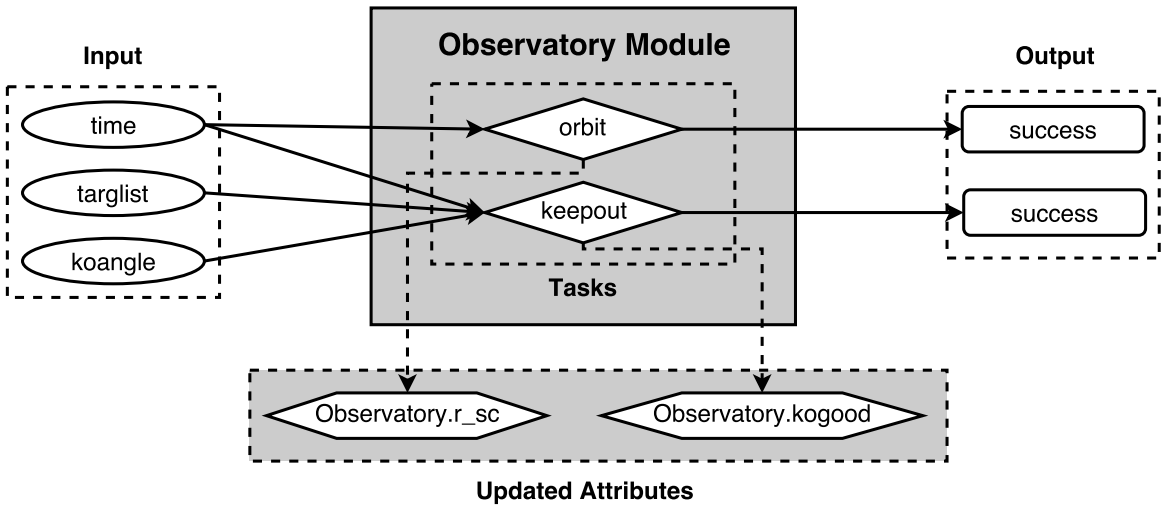

Fig. 8. Depiction of Observatory module tasks including inputs, outputs, and updated attributes (see 5.7.2 and 5.7.3).

5.7.1 Observatory Object Attribute Initialization Input/Output Description

Inputs

User specification

Information from simulation specification JSON file organized into a Python dictionary. If the below key: value pairs are missing from the dictionary, the Observatory object attributes will be assigned the default values listed.

specs["settling_time"]
: Amount of time needed for observatory to settle after a repointing in *days*. Default value is 1.

specs["thrust"]
: Occulter slew thrust in *mN*. Default value is 450.

specs["slewIsp"]
: Occulter slew specific impulse in *s*. Default value is 4160.

specs["sc_mass"]
: Occulter (maneuvering spacecraft) initial wet mass in *kg*. Default value is 6000.

specs["dryMass"]
: Occulter (maneuvering spacecraft) dry mass in *kg*. Default value is 3400.

specs["coMass"]
: Telescope (or non-maneuvering spacecraft) mass in *kg*. Default value is 5800.

specs["occulterSep"]
: Occulter-telescope distance in *km*. Default value is 55000.

specs["skIsp"]
: Specific impulse for station keeping in *s*. Default value is 220.

specs["defburnPortion"]
: Default burn portion for slewing. Default value is 0.05

Outputs

Observatory.settling_time
: Amount of time needed for observatory to settle after a repointing (astropy Quantity object initially set in days)

Observatory.thrust
: Occulter slew thrust (astropy Quantity object initially set in *mN*)

Observatory.slewIsp
: Occulter slew specific impulse (astropy Quantity object initially set in *s*)

Observatory.sc_mass
: Occulter (maneuvering spacecraft) initial wet mass (astropy Quantity object initially set in *kg*)

Observatory.dryMass
: Occulter (maneuvering spacecraft) dry mass (astropy Quantity object initially set in *kg*)

Observatory.coMass
: Telescope (or non-maneuvering spacecraft) mass (astropy Quantity object initially set in *kg*)

Observatory.kogood
: 1D NumPy ndarray of Boolean values where True is a target unobstructed and observable in the keepout zone. Initialized to an empty array. This attribute is updated to the current mission time through the keepout task (see 5.7.3).

Observatory.r_sc
: Observatory orbit position in HE reference frame. Initialized to NumPy ndarray as numpy.array([0., 0., 0.]) and associated with astropy Quantity object in *km*. This attribute is updated to the orbital position of the observatory at the current mission time through the orbit task (see 5.7.2).

Observatory.occulterSep
: Occulter-telescope distance (astropy Quantity object initially set in *km*)

Observatory.skIsp
: Specific impulse for station keeping (astropy Quantity object initially set in *s*)

Observatory.defburnPortion
: Default burn portion for slewing

Observatory.currentSep
: Current occulter separation (astropy Quantity object initially set in *km*

Observatory.flowRate
: Slew flow rate derived from Observatory.thrust and Observatory.slewIsp (astropy Quantity object initially set in kg/day)

5.7.2 orbit Task Input/Output Description

The `orbit` task finds the heliocentric equatorial position vector of the observatory spacecraft.

Inputs

time

astropy Time object which may be `TimeKeeping.currenttimeAbs` from Time Keeping module see 5.8.1 for definition

Outputs

success

Boolean indicating if orbit was successfully calculated

Updated Object Attributes

Observatory.r_sc

Observatory orbit position in HE reference frame at current mission time (astropy Quantity object defined in *km*)

5.7.3 keepout Task Input/Output Description

The `keepout` task determines which stars in the target list are observable at the given input time.

Inputs

time

astropy Time object which may be `TimeKeeping.currenttimeAbs` (see 5.8.1 for definition)

targlist

Instantiated Target List object from Target List module. See 6.2 for definition of available attributes

koangle

Telescope keepout angle in degrees - `OpticalSystem.telescopeKeepout`

Outputs

success

Boolean indicating if orbit was successfully calculated

Updated Object Attributes

Observatory.kogood

1D NumPy ndarray of Boolean values for each target at given time where True is a target unobstructed in the keepout zone and False is a target unobservable due to obstructions in the keepout zone

5.8 Time Keeping

The Time Keeping module is responsible for keeping track of the current mission time. It encodes only the mission start time, the mission duration, and the current time within a simulation. All functions in all modules requiring knowledge of the current time call functions or access parameters implemented within the Time module. Internal encoding of time is implemented as the time from mission start (measured in days). The Time Keeping module also provides functionality for converting between this time measure and standard measures such as Julian Day Number and UTC time.

The Time Keeping module contains the `update_times` and `duty_cycle` tasks. These task updates the mission time during a survey simulation. The duty cycle determines when during the mission timeline the observatory is allowed to perform planet-finding operations. The duty cycle function takes the current mission time as input and outputs the next available time when exoplanet observations may begin or resume, along with the duration of the observational period. The outputs of this task are used in the Survey Simulation module to determine when and how long exoplanet finding and characterization observations occur. The inputs and updated attributes for the Time Keeping tasks are depicted in Fig. 9.

5.8.1 Time Keeping Object Attribute Initialization Input/Output Description

Inputs

User specification

Information from simulation specification JSON file organized into a Python dictionary. If the below key: value pairs are missing from the dictionary, the Time object attributes will be assigned the default values listed.

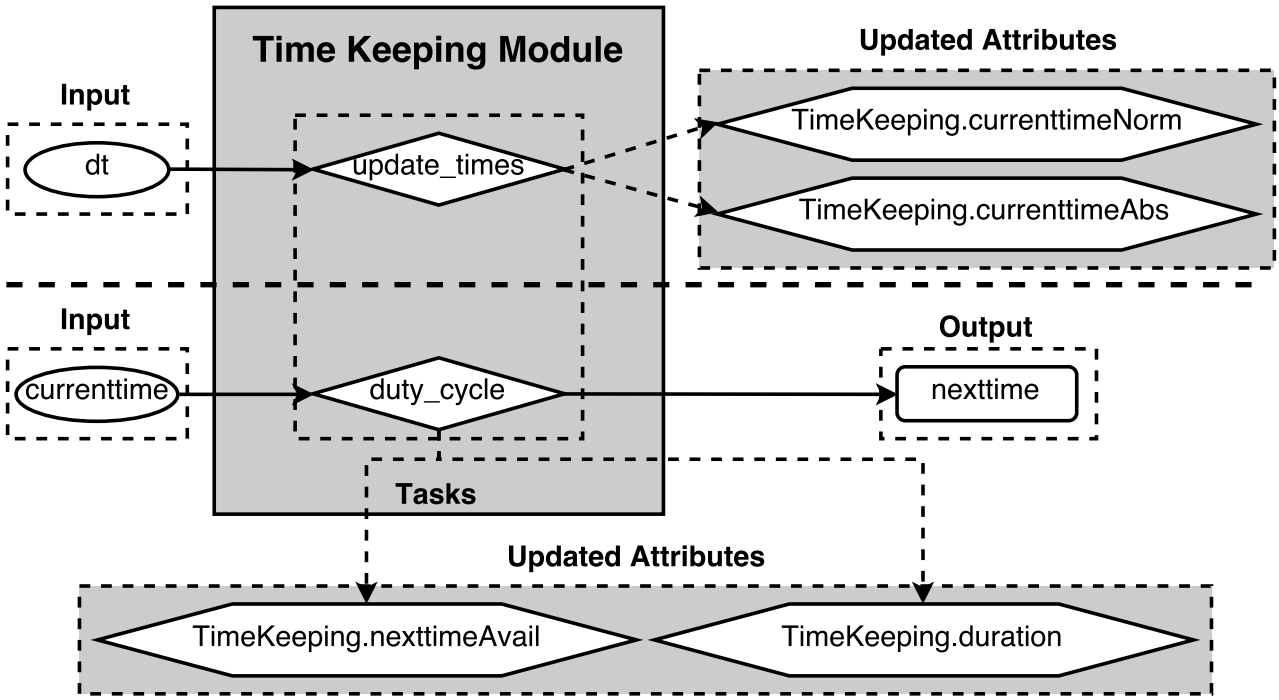

Fig. 9. Depiction of Time Keeping module task including input and updated attributes (see 5.8.2 and 5.8.3).

specs[“missionStart”]
Mission start time in *MJD*. Default value is 60634.
specs[“missionLife”]
Total length of mission in *years*. Default value is 6.
specs[“extendedLife”]
Extended mission time in *years*. Default value is 0.
specs[“missionPortion”]
Portion of mission time devoted to planet-finding. Default value is 1/6.
specs[“duration”]
Duration of planet-finding operations in *days*. Default value is 14.
specs[“nexttimeAvail”]
Next time available for planet-finding in *days*. Default value is 0.

Outputs

TimeKeeping.missionStart
Mission start time (astropy Time object initially defined in *MJD*)
TimeKeeping.missionLife
Mission lifetime (astropy Quantity object initially set in *years*)
TimeKeeping.extendedLife
Extended mission time (astropy Quantity object initially set in *years*)
TimeKeeping.missionPortion
Portion of mission time devoted to planet-finding
TimeKeeping.duration
Duration of planet-finding operations (astropy Quantity object initially set in *days*)
TimeKeeping.nexttimeAvail
Next time available for planet-finding (astropy Quantity object initially set in *days*)
TimeKeeping.currenttimeNorm
Current mission time normalized so that start date is 0 (astropy Quantity object initially set in *days*)
TimeKeeping.currenttimeAbs

Current absolute mission time (astropy Time object initially defined in *MJD*)

TimeKeeping.missionFinishNorm

Mission finish time (astropy Quantity object initially set in *days*)

TimeKeeping.missionFinishAbs

Mission completion date (astropy Time object initially defined in *MJD*)

5.8.2 update_times Task Input/Output Description

The `update_times` task updates the relevant mission times.

Inputs

dt

Time increment (astropy Quantity object with units of time)

Updated Object Attributes

TimeKeeping.currenttimeNorm

Current mission time normalized so that start date is 0 (astropy Quantity object with units of time)

TimeKeeping.currenttimeAbs

Current absolute mission time (astropy Time object)

5.8.3 duty_cycle Task Input/Output Description

The `duty_cycle` task updates the available time and duration for planet-finding operations.

Inputs

currenttime

Current time in mission simulation (astropy Quantity object with units of time often `TimeKeeping.currenttimeNorm`)

Outputs

nexttime

Next available time for planet-finding (astropy Quantity object with units of time)

Updated Object Attributes

TimeKeeping.nexttimeAvail

Next time available for planet-finding (astropy Quantity object with units of time)

TimeKeeping.duration

Duration of planet-finding operations (astropy Quantity object with units of time)

5.9 Post-Processing

The Post-Processing module encodes the effects of post-processing on the data gathered in a simulated observation, and the effects on the final contrast of the simulation. The Post-Processing module is also responsible for determining whether a planet detection has occurred for a given observation, returning one of four possible states—true positive (real detection), false positive (false alarm), true negative (no detection when no planet is present) and false negative (missed detection). These can be generated based solely on statistical modeling or by processing simulated images.

The Post-Processing module contains the `det_occur` task. This task determines if a planet detection occurs for a given observation. The inputs and outputs for this task are depicted in Fig. 10.

5.9.1 Post-Processing Object Attribute Initialization Input/Output Description

Inputs

User specification

Information from simulation specification JSON file organized into a Python dictionary. If the below key: value pairs are missing from the dictionary, the Time object attributes will be assigned the default values listed.

specs[“FAP”]

Detection false alarm probability. Default value is 0.01/1000.

specs[“MDP”]

Missed detection probability. Default value is 0.001.

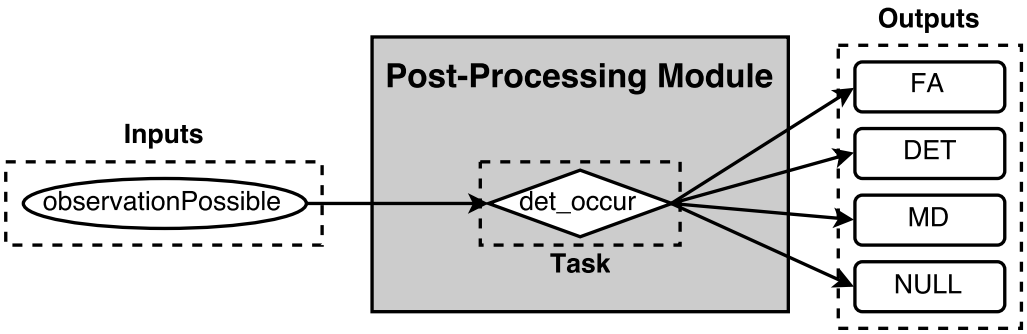

Fig. 10. Depiction of Post-Processing module task including inputs and outputs (see 5.9.2).

Outputs

PostProcessing.FAP

Detection false alarm probability

PostProcessing.MDP

Missed detection probability

5.9.2 det_occur Task Input/Output Description

The `det_occur` task determines if a planet detection has occurred.

Inputs

observationPossible

1D NumPy ndarray of booleans signifying if a planet in the system being observed is observable

Outputs

FA

Boolean where True means False Alarm

DET

Boolean where True means DETection

MD

Boolean where True means Missed Detection

NULL

Boolean where True means Null Detection

6 Simulation Modules

The simulation modules include Completeness, Target List, Simulated Universe, Survey Simulation and Survey Ensemble. These modules perform tasks which require inputs from one or more input modules as well as calling function implementations in other simulation modules.

6.1 Completeness

The Completeness module takes in information from the Planet Population module to determine initial completeness and update completeness values for target list stars when called upon.

The Completeness module contains the following tasks: `target_completeness` and `completeness_update`. `target_completeness` generates initial completeness values for each star in the target list (see 6.1.2). `completeness_update` updates the completeness values following an observation (see 6.1.3).

6.1.1 Completeness Object Attribute Initialization Input/Output Description

Input

User specification

Information from simulation specification JSON file organized into a Python dictionary. If the below key: value

pairs are missing from the dictionary, the Completeness object attributes will be assigned the default values listed.

specs[“minComp”]
Minimum completeness level for detection

PlanetPopulation
Output of Planet Population module (see 5.1)

Output

Completeness.PlanetPopulation
Output of Planet Population module (see 5.1)
Completeness.minComp
Minimum completeness level for detection

6.1.2 target_completeness Task Input/Output Description

The `target_completeness` task generates completeness values for each star in the target list.

Inputs

targlist
Instantiated Target List object from Target List module see 6.2 for definition of functionality and attributes

Outputs

comp0
1D NumPy ndarray containing completeness values for each star in the target list

6.1.3 completeness_update Task Input/Output Description

The `completeness_update` task updates the completeness values for each star in the target list following an observation.

Inputs

s_ind
index of star in target list just observed
targlist
Instantiated Target List object from Target List module see 6.2 for definition of functionality and attributes
obsbegin
Mission time when the observation of `s_ind` began (astropy Quantity object with units of time)
obsend
Mission time when the observation of `s_ind` ended (astropy Quantity object with units of time)
nexttime
Mission time of next observational period (astropy Quantity object with units of time)

Output

comp0
1D NumPy ndarray of updated completeness values for each star in the target list

6.2 Target List

The Target List module takes in information from the Optical System, Star Catalog, Planet Population, and Observatory input modules and Completeness simulation module to generate the target list for the simulated survey. This list can either contain all of the targets where a planet with specified parameter ranges could be observed or a list of pre-determined targets such as in the case of a mission which only seeks to observe stars where planets are known to exist from previous surveys. The final target list encodes all of the same information as is provided by the Star Catalog module.

6.2.1 Target List Object Attribute Initialization Input/Output Description

Inputs

User specification
Information from simulation specification JSON file organized into a Python dictionary. If key: value pairs are missing from the dictionary, the Target List object attributes will be assigned the default values.

StarCatalog
Output of Star Catalog module (see 5.2)

OpticalSystem
Output of Optical System module (see 5.3)

PlanetPopulation
Output of Planet Population module (see 5.1)

ZodiacalLight
Output of Zodiacal Light module (see 5.4)

Completeness
Output of Completeness module (see 6.1)

Outputs

TargetList.(StarCatalog values)
Mission specific filtered star catalog values from Star Catalog module (see 5.2)

TargetList.OpticalSystem
Output of Optical System module (see 5.3)

TargetList.PlanetPopulation
Output of Planet Population module (see 5.1)

TargetList.ZodiacalLight
Output of Zodiacal Light module (see 5.4)

TargetList.Completeness
Output of Completeness module (see 6.1)

TargetList.maxintTime
1D NumPy ndarray of maximum integration time for each target star found from `OpticalSystem.calc_maxintTime` 5.3.2 (astropy Quantity object with units of time)

TargetList.comp0
1D NumPy ndarray of completeness value for each target star found from `Completeness.target_completeness` 6.1.2

TargetList.MsEst
Approximate stellar mass in M_{sun}

TargetList.MsTrue
Stellar mass with an error component included in M_{sun}

6.3 Simulated Universe

The Simulated Universe module takes as input the outputs of the Target List simulation module to create a synthetic universe composed of the systems in the target list. For each target, a planetary system is generated based on the statistics encoded in the Planet Population module, so that the overall planet occurrence and multiplicity rates are consistent with the provided distribution functions. Physical parameters for each planet are similarly sampled from the input density functions. This universe is encoded as a list where each entry corresponds to one element of the target list, and where the list entries are arrays of planet physical parameters. In cases of empty planetary systems, the corresponding list entry contains a null array.

The Simulated Universe module also takes as input the Planet Physical Model module instance, so that it can return the specific spectra due to every simulated planet at an arbitrary observation time throughout the mission simulation.

The Simulated Universe module contains the following tasks: `planet_to_star`, `planet_a`, `planet_e`, `planet_w`, `planet_O`, `planet_radii`, `planet_masses`, `planet_albedos`, `planet_inclinations`, `planet_pos_vel`, and `prop_system`. `planet_pos_vel` finds initial position and velocity vectors for each planet (see 6.3.9). `prop_system` propagates planet position and velocity vectors in time (see 6.3.12). The rest of the tasks assign orbital or physical quantities to each planet (see 6.3.2, 6.3.3, 6.3.4, 6.3.5, 6.3.6, 6.3.7, 6.3.8, 6.3.10, and 6.3.11. The inputs and outputs for these tasks are depicted in Fig. 11.

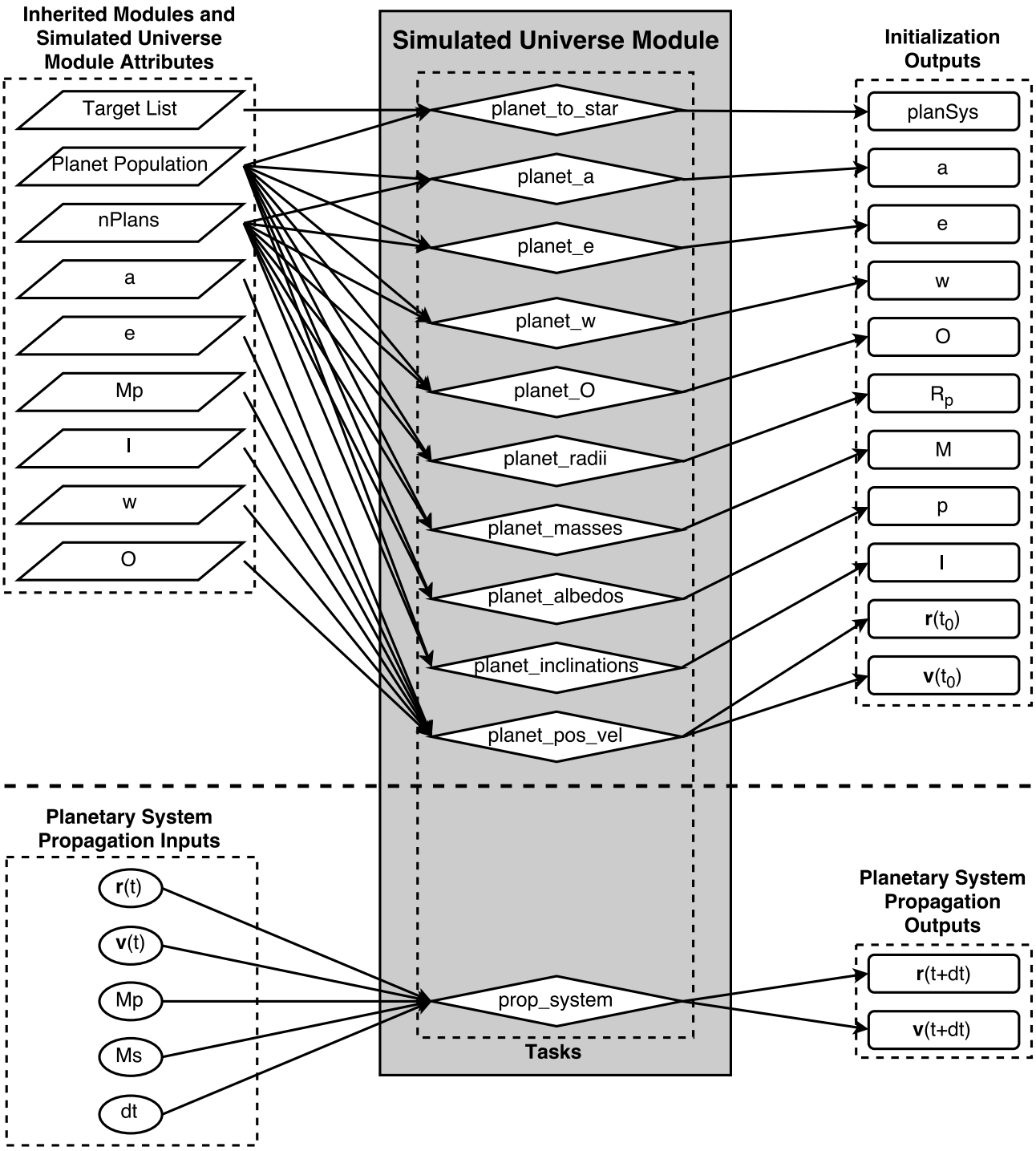

Fig. 11. Depiction of Simulated Universe module tasks including inputs and outputs (see 6.3.2, 6.3.3, 6.3.4, 6.3.5, 6.3.6, 6.3.7, 6.3.8, 6.3.9, 6.3.10, 6.3.11, and 6.3.12).

6.3.1 Simulated Universe Object Attribute Initialization Input/Output Description

Inputs

User specification

Information from simulation specification JSON file organized into a Python dictionary. If the below key: value pairs are missing from the dictionary, the Simulated Universe object attributes will be assigned the default values listed.

OpticalSystem
Output of Optical System module inherited from Target List module (see 5.3)
PlanetPopulation
Output of Planet Population module inherited from Target List module (see 5.1)
ZodiacalLight
Output of Zodiacal Light module inherited from Target List module (see 5.4)
Completeness
Output of Completeness module inherited from Target List module (see 6.1)
TargetList
Output of Target List module given by above specs[“targlistname”] (see 6.2)
PlanetPhysicalModel
Output of Planet Physical Model module (see 5.6)

Outputs
SimulatedUniverse.OpticalSystem
Output of Optical System module (see 5.3)
SimulatedUniverse.pop
Output of Planet Population module (see 5.1)
SimulatedUniverse.ZodiacalLight
Output of Zodiacal Light module (see 5.4)
SimulatedUniverse.Completeness
Output of Completeness module (see 6.1)
SimulatedUniverse.TargetList
Output of Target List module (see 6.2)
SimulatedUniverse.PlanetPhysicalModel
Output of Planet Physical Model module (see 5.6)
SimulatedUniverse.planInds
1D NumPy ndarray containing indices (referenced to exoplanet list) mapping each planet to its star determined from `planet_to_star()` 6.3.2
SimulatedUniverse.nPlans
Number of planets (determined from `len(SimulatedUniverse.planInds)`)
SimulatedUniverse.sysInds
1D NumPy ndarray containing indices (referenced to target star list) of target stars with planets
SimulatedUniverse.a
1D NumPy ndarray containing semi-major axis for each planet determined from `planet_a()` 6.3.3
SimulatedUniverse.e
1D NumPy ndarray containing eccentricity values for each planet determined from `planet_e()` 6.3.4
SimulatedUniverse.w
1D NumPy ndarray containing argument of perigee in degrees for each planet determined from `planet_w()` 6.3.5
SimulatedUniverse.O
1D NumPy ndarray containing right ascension of the ascending node in degrees for each planet determined from `planet_O()` 6.3.6
SimulatedUniverse.Mp
1D NumPy ndarray containing masses of each planet determined from `planet_masses()` 6.3.7
SimulatedUniverse.Rp
1D NumPy ndarray containing radii of each planet determined from `planet_radii()` 6.3.8
SimulatedUniverse.r
1D NumPy ndarray containing planet position vectors relative to host stars determined from `planet_pos_vel()` 6.3.9
SimulatedUniverse.v
1D NumPy ndarray containing planet velocity vectors relative to host stars determined from `planet_pos_vel()` 6.3.9
SimulatedUniverse.p
1D NumPy ndarray containing planet geometric albedos determined from `planet_albedos()` 6.3.10
SimulatedUniverse.I
1D NumPy ndarray containing list of inclination of planetary systems in degrees determined from `planet_inclinations()`

6.3.11

SimulatedUniverse.fzodicurr

1D NumPy ndarray containing list of exozodi levels for systems with planets determined from `ZodiacalLight.fzodi` 5.4.2

6.3.2 planet_to_star Task Input/Output Description

The `planet_to_star` task determines how many planets belong to each star in the target list. It returns a 1D NumPy ndarray containing the indices of the target star to which each planet belongs. The length of the ndarray is the total number of planets in the simulated universe.

Inputs

This task does not take any explicit inputs. It uses the inherited Target List and Planet Population objects.

Outputs

planSys

1D NumPy ndarray containing the indices of each target star to which each planet (each element of the array) belongs

6.3.3 planet_a Task Input/Output Description

The `planet_a` task assigns each planet a semi-major axis with astropy Quantity units of distance (AU). The prototype samples the probablity density function `PlanetPopulation.semi_axis(x)` 5.1.2 and returns appropriate output.

Inputs

This task does not take any explicit inputs. It uses the inherited Planet Population object and total number of planets (`SimulatedUniverse.nPlans`).

Outputs

a

1D NumPy ndarray containing semi-major axis for each planet (astropy Quantity object with units of *AU*)

6.3.4 planet_e Task Input/Output Description

The `planet_e` task assigns each planet an eccentricity value. The prototype samples the probablity density function `PlanetPopulation.eccentricity(x)` 5.1.2 and returns appropriate output.

Inputs

This task does not take any explicit inputs. It uses the inherited Planet Population object and total number of planets (`SimulatedUniverse.nPlans`).

Outputs

e

1D NumPy ndarray containing eccentricity for each planet

6.3.5 planet_w Task Input/Output Description

The `planet_w` task assigns each planet argument of perigee in degrees. The prototype samples the probablity density function `PlanetPopulation.arg_perigee(x)` 5.1.2 and returns appropriate output.

Inputs

This task does not take any explicit inputs. It uses the inherited Planet Population object and total number of planets (`SimulatedUniverse.nPlans`).

Outputs

w

1D NumPy ndarray containing argument of perigee in degrees for each planet

6.3.6 planet_O Task Input/Output Description

The `planet_O` task assigns each planet right ascension of the ascending node in degrees. The prototype samples the probablity density function `PlanetPopulation.RAAN(x)` 5.1.2 and returns appropriate output.

Inputs

This task does not take any explicit inputs. It uses the inherited Planet Population object and total number of planets (`SimulatedUniverse.nPlans`).

Outputs

O

1D NumPy ndarray containing right ascension of the ascending node in degrees for each planet

6.3.7 planet_masses Task Input/Output Description

The `planet_masses` task assigns each planet mass with astropy Quantity units of mass (default is *kg*). The prototype samples the probablity density function `PlanetPopulation.mass(x)` 5.1.2 and returns appropriate output.

Inputs

This task does not take any explicit inputs. It uses the inherited Planet Population object and total number of planets (`SimulatedUniverse.nPlans`).

Outputs

M

1D NumPy ndarray containing mass for each planet (astropy Quantity object with default units of *kg*)

6.3.8 planet_radii Task Input/Output Description

The `planet_radii` task assigns each planet a radius with astropy Quantity units of distance (default is *km*). The prototype samples the probablity density function `PlanetPopulation.radius(x)` 5.1.2 and returns appropriate output.

Inputs

This task does not take any explicit inputs. It uses the inherited Planet Population object and total number of planets (`SimulatedUniverse.nPlans`).

Outputs

Rp

1D NumPy ndarray containing radius for each planet (astropy Quantity object with default units of *km*)

6.3.9 planet_pos_vel Task Input/Output Description

The `planet_pos_vel` task assigns each planet an initial position and velocity vector with appropriate astropy Quantity units attached.

Inputs

This task does not take any explicit inputs. It uses the following attributes assigned before calling this task:

```
SimulatedUniverse.a
SimulatedUniverse.e
SimulatedUniverse.Mp
SimulatedUniverse.I
SimulatedUniverse.w
SimulatedUniverse.O
```

Outputs

r

NumPy ndarray containing initial position vector for each planet (astropy Quantity object with default units of *km*)

v

NumPy ndarray containing initial velocity vector for each planet (astropy Quantity object with default units of km/s)

6.3.10 planet_albedos Task Input/Output Description

The `planet_albedos` task assigns each planet a geometric albedo value. The prototype samples the probablity density function `PlanetPopulation.albedo(x)` 5.1.2 and returns appropriate output.

Inputs

This task does not take any explicit inputs. It uses the inherited Planet Population object and total number of planets (`SimulatedUniverse.nPlans`).

Outputs

p

1D NumPy ndarray containing geometric albedo for each planet

6.3.11 planet_inclinations Task Input/Output Description

The `planet_inclinations` task assigns each planet an inclination in degrees. The prototype samples the probablity density function `PlanetPopulation.inclination(x)` 5.1.2 and returns appropriate output.

Inputs

This task does not take any explicit inputs. It uses the inherited Planet Population object and total number of planets (`SimulatedUniverse.nPlans`).

Outputs

I

1D NumPy ndarray containing inclination for each planet in degrees

6.3.12 prop_system Task Input/Output Description

The `prop_system` task propagates planet state vectors (position and velocity) in time.

Inputs

r

NumPy ndarray containing current planet position vectors relative to host star (astropy Quantity object with units of distance)

v

NumPy ndarray containing current planet velocity vectors relative to host star (astropy Quantity object with units of distance/time)

Mp

1D NumPy ndarray containing planet masses (astropy Quantity object with units of mass)

Ms

1D NumPy ndarray containing target star mass in M_{sun}

dt

Time increment to propagate system (astropy Quantity object with units of time)

Outputs

rnew

NumPy ndarray of propagated position vectors relative to host star (astropy Quantity object with units of distance)

vnew

NumPy ndarray of propagated velocity vectors relative to host star (astropy Quantity object with units of distance/time)

6.4 Survey Simulation

The Survey Simulation module takes as input the output of the Simulated Universe simulation module and the Time Keeping, and Post-Processing input modules. This is the module that performs a specific simulation based on all of the input parameters and models. This module returns the mission timeline - an ordered list of simulated observations of various targets on the target list along with their outcomes. The output also includes an encoding of the final state of the simulated universe (so that a subsequent simulation can start from where a previous simulation left off) and the final state of the observatory definition (so that post-simulation analysis can determine the percentage of volatiles expended, and other engineering metrics).

Survey Simulation TASKS: `run_sim()` - perform survey simulation 6.4.2

Survey Simulation SUBTASKS: `initial_target()` - find initial target star 6.4.3

`observation_detection(pInds, s_ind, DRM, planPosTime)` - finds if planet detections are possible and returns relevant information 6.4.4

`det_data(s, dMag, Ip, DRM, FA, DET, MD, s_ind, pInds, observationPossible, observed)` - determines detection status 6.4.5

`observation_characterization(observationPossible, pInds, s_ind, spectra, s, Ip, DRM, FA, t_` - finds if characterizations are possible and returns relevant information 6.4.6

`next_target(s_ind, revisit_list, extended_list, DRM)` - find next target (scheduler) 6.4.7

6.4.1 Survey Simulation Object Attribute Initialization Input/Output Description

Inputs

User specification
Information from simulation specification JSON file organized into a Python dictionary. If the below key: value pairs are missing from the dictionary, the Survey Simulation object attributes will be assigned the default values listed.

OpticalSystem
Output of Optical System module inherited from Simulated Universe module (see 5.3)

PlanetPopulation
Output of Planet Population module inherited from Simulated Universe module (see 5.1)

ZodiacalLight
Output of Zodiacal Light module inherited from Simulated Universe module (see 5.4)

Completeness
Output of Completeness module inherited from Simulated Universe module (see 6.1)

TargetList
Output of Target List module inherited from Simulated Universe module (see 6.2)

PlanetPhysicalModel
Output of Planet Physical Model module inherited from Simulated Universe module (see 5.6)

SimulatedUniverse
Output of Simulated Universe module (see 6.3)

Observatory
Output of Observatory module (see 5.7)

TimeKeeping
Output of Time Keeping module (see 5.8)

PostProcessing
Output of Post-Processing module (see 5.9)

Outputs

SurveySimulation.OpticalSystem
Output of Optical System module (see 5.3)

SurveySimulation.PlanetPopulation
Output of Planet Population module (see 5.1)

SurveySimulation.ZodiacalLight
Output of Zodiacal Light module (see 5.4)

SurveySimulation.Completeness
Output of Completeness module (see 6.1)

SurveySimulation.TargetList
Output of Target List module (see 6.2)

SurveySimulation.PlanetPhysicalModel
Output of Planet Physical Model module (see 5.6)
SurveySimulation.SimulatedUniverse
Output of Simulated Universe module (see 6.3)
SurveySimulation.Observatory
Output of Observatory module (see 5.7)
SurveySimulation.TimeKeeping
Output of Time Keeping module (see 5.8)
SurveySimulation.PostProcessing
Output of Post-Processing module (see 5.9)
SurveySimulation.DRM
Contains the results of survey simulation

6.4.2 run_sim Task Input/Output Description

The `run_sim` task performs the survey simulation and populates the results in `SurveySimulation.DRM`.

Inputs

This task does not take any explicit inputs. It uses the inherited modules to generate a survey simulation.

Updated Object Attributes

SurveySimulation.DRM
Python list where each entry contains a dictionary of survey simulation results for each observation. The dictionary may include the following key:value pairs (from the prototype):

'target_ind'
Index of star in target list observed
'arrival_time'
Days since mission start when observation begins
'sc_mass'
Maneuvering spacecraft mass (if simulating an occulter system)
'dF_lateral'
Lateral disturbance force on occulter in *N* if simulating an occulter system
'dF_axial'
Axial disturbance force on occulter in *N* if simulating an occulter system
'det_dV'
Detection station-keeping ΔV in m/s if simulating an occulter system
'det_mass_used'
Detection station-keeping fuel mass used in *kg* if simulating an occulter system
'det_int_time'
Detection integration time in *days*
'det_status'
Integer or list where

1 = detection
0 = null detection
-1 = missed detection
-2 = false alarm

'det_WA'
Detection WA in *milliarcseconds*
'det_dMag'
Detection Δmag
'char_1_time'
Characterization integration time in *days*
'char_1_dV'
Characterization station-keeping ΔV in m/s if simulating an occulter system
'char_1_mass_used'
Characterization station-keeping fuel mass used in *kg* if simulating an occulter system

'char_1_success'
Characterization success where value which may be:
1 - successfull characterization
effective wavelength found during characterization in *nm*

'slew_time'
Slew time to next target in *days* if simulating an occulter system

'slew_dV'
Slew ΔV in m/s if simulating an occulter system

'slew_mass_used'
Slew fuel mass used in *kg* if simulating an occulter system

'slew_angle'
Slew angle to next target in *rad*

6.4.3 initial_target Sub-task Input/Output Description

The `initial_target` sub-task is called from the `run_sim` task to determine the index of the initial target star in the target list.

Inputs

This sub-task does not take any explicit inputs. It may use any of the inherited modules to generate the initial target star index.

Outputs

s_ind
Index of the initial target star

6.4.4 observation_detection Sub-task Input/Output Description

The `observation_detection` sub-task is called from the `run_sim` task to determine if planets may be detected and calculate information needed later in the simulation.

Inputs

pInds
1D NumPy ndarray of indices of planets belonging to the target star (used to get relevant attributes from the `SimulatedUniverse` module)

s_ind
Index of target star in target list

DRM
Python dictionary containing survey simulation results of current observation as key:value pairs

planPosTime
1D NumPy ndarray containing the times at which the planet positions and velocities contained in `SimulatedUniverse.r` and `SimulatedUniverse.v` are current (astropy Quantity object with units of time)

Outputs

observationPossible
1D NumPy ndarray (length is number of planets in the system under observation) containing boolean values where True is an observable planet

t_int
Integration time (astropy Quantity object with units of time)

DRM
Python dictionary containing survey simulation results of current observation as key:value pairs

s
1D NumPy ndarray (length is number of planets in the system under observation) containing apparent separation of planets (astropy Quantity object with units of distance)

dMag
1D NumPy ndarray (length is number of planets in the system under observation) containing Δmag for each planet

Ip
1D NumPy ndarray (length is number of planets in the system under observation) containing irradiance (astropy Quantity object with units of $\frac{1}{m^2 \cdot nm \cdot s}$)

6.4.5 det_data Sub-task Input/Output Description

The `det_data` sub-task is called from the `run_sim` task to assign a detection status to the dictionary of current observation results.

Inputs

s
1D NumPy array (length is number of planets in the system under observation) containing apparent separation of planets (astropy Quantity object with units of distance)

dMag
1D NumPy ndarray (length is number of planets in the system under observation) containing Δmag for each planet

Ip 1D NumPy ndarray (length is number of planets in the system under observation) containing irradiance (astropy Quantity object with units of $\frac{1}{m^2 \cdot nm \cdot s}$)

DRM
Python dictionary containing survey simulation results of current observation as key:value pairs

FA
Boolean where True is False Alarm

DET
Boolean where True is DETection

MD
Boolean where True is Missed Detection

s_ind
Index of target star in target list

pInds
1D NumPy ndarray of indices of planets belonging to the target star (used to get relevant attributes from the `SimulatedUniverse` module)

observationPossible
1D NumPy ndarray (length is number of planets in the system under observation) containing boolean values where True is an observable planet

observed
1D NumPy ndarray which contains the number of observations for each planet in the simulated universe

Outputs

s
1D NumPy array (length is number of planets in the system under observation) containing apparent separation of planets (astropy Quantity object with units of distance)

dMag
1D NumPy ndarray (length is number of planets in the system under observation) containing Δmag for each planet

Ip
1D NumPy ndarray (length is number of planets in the system under observation) containing irradiance (astropy Quantity object with units of $\frac{1}{m^2 \cdot nm \cdot s}$)

DRM
Python dictionary containing survey simulation results of current observation as key:value pairs

observed
1D NumPy ndarray which contains the number of observations for each planet in the simulated universe

6.4.6 observation_characterization Sub-task Input/Output Description

The `observation_characterization` sub-task is called by the `run_sim` task to determine if characterizations are to be performed and calculate relevant characterization information to be used later in the observation simulation.

Inputs

observationPossible
1D NumPy ndarray (length is number of planets in the system under observation) containing boolean values where True is an observable planet

pInds
1D NumPy ndarray of indices of planets belonging to the target star (used to get relevant attributes from the `SimulatedUniverse` module)

s_ind
Index of target star in target list

spectra
NumPy ndarray where 1 denotes spectra for a planet that has been captured, 0 denotes spectra for a planet that has not been captured

s
1D NumPy array (length is number of planets in the system under observation) containing apparent separation of planets (astropy Quantity object with units of distance)

Ip
1D NumPy ndarray (length is number of planets in the system under observation) containing irradiance (astropy Quantity object with units of $\frac{1}{m^2 \cdot nm \cdot s}$)

DRM
Python dictionary containing survey simulation results of current observation as key:value pairs

FA
Boolean where True is False Alarm

t_int
Integration time (astropy Quantity object with units of time)

Outputs

DRM
Python dictionary containing survey simulation results of current observation as key:value pairs

FA
Boolean where True is False Alarm

spectra
NumPy ndarray where 1 denotes spectra for a planet that has been captured, 0 denotes spectra for a planet that has not been captured

6.4.7 next_target Sub-task Input/Output Description

The `next_target` sub-task is called from the `run_sim` task to determine the index of the next star from the target list for observation.

Inputs

s_ind
Index of current star from the target list

targlist
Target List module (see 6.2)

revisit_list
NumPy ndarray containing index of target star and time in days of target stars from the target list to revisit

extended_list
1D NumPy ndarray containing the indices of stars in the target list to consider if in extended mission time

DRM
Python dictionary containing survey simulation results of current observation as key:value pairs

Outputs

new_s_ind
Index of next target star in the target list

DRM
Python dictionary containing survey simulation results of current observation as key:value pairs

6.5 Survey Ensemble NEEDS UPDATING

The Survey Ensemble module's only task is to run multiple simulations. While the implementation of this module is not at all dependent on a particular mission design, it can vary to take advantage of available parallel-processing resources. As the generation of a survey ensemble is an embarrassingly parallel task—every survey simulation is fully independent and can be run as a completely separate process—significant gains in execution time can be achieved with parallelization. The baseline implementation of this module contains a simple looping function that executes the desired number of simulations sequentially, as well as a locally parallelized version based on IPython Parallel.

Depending on the local setup, the Survey Ensemble implementation could also potentially save time by cloning survey module objects and reinitializing only those sub-modules that have stochastic elements (i.e., the simulated universe).

Another possible implementation variation is to use the Survey Ensemble module to conduct investigations of the effects of varying any normally static parameter. This could be done, for example, to explore the impact on yield in cases where the non-coronagraph system throughput, or elements of the propulsion system, are mischaracterized prior to launch. This SE module implementation would overwrite the parameter of interest given in the input specification for every individual survey executed, and saving the true value of the parameter used along with the simulation output.