

---

# **Bioverse**

***Release 1.0***

**Alex Bixel**

**May 13, 2023**



# OVERVIEW

<b>1</b>	<b>References &amp; Acknowledgements</b>	<b>3</b>
<b>2</b>	<b>Installation</b>	<b>5</b>
<b>3</b>	<b>Dependencies</b>	<b>7</b>
3.1	Introduction . . . . .	7
3.2	The Table class . . . . .	8
3.3	Generating planetary systems . . . . .	10
3.4	Simulating survey datasets . . . . .	12
3.5	Hypothesis testing . . . . .	15
3.6	Computing statistical power . . . . .	18
3.7	Object Editor . . . . .	21
3.8	Tutorial 1: Generating planetary systems . . . . .	24
3.9	Tutorial 2: Simulating survey datasets . . . . .	30
3.10	Tutorial 3: Calculating exposure times . . . . .	38
3.11	Example 1: Finding the habitable zone . . . . .	42
3.12	Example 2: Detecting the age-oxygen correlation . . . . .	47
3.13	bioverse.analysis module . . . . .	52
3.14	bioverse.classes module . . . . .	53
3.15	bioverse.constants module . . . . .	55
3.16	bioverse.custom module . . . . .	55
3.17	bioverse.functions module . . . . .	55
3.18	bioverse.generator module . . . . .	63
3.19	bioverse.hypothesis module . . . . .	65
3.20	bioverse.plots module . . . . .	70
3.21	bioverse.survey module . . . . .	72
3.22	bioverse.util module . . . . .	77
	<b>Python Module Index</b>	<b>81</b>
	<b>Index</b>	<b>83</b>



Bioverse is a Python package for simulating the results of a statistical survey of the properties of nearby terrestrial exoplanets via direct imaging or transit spectroscopy. An in-depth outline of the underlying statistical framework and examples of how it can be applied to astrophysics mission concepts is given in [Bixel & Apai \(2021\)](#). **Readers are strongly encouraged to review this paper before proceeding.** This documentation covers the Python implementation of Bioverse, but does not review many of its underlying statistical assumptions.

The [Overview](#) section describes the code's structure and primary classes and should be reviewed first. Following that, the [Examples](#) section offers step-by-step examples for producing some of the results published in the paper, as well as ways to modify and expand upon the code. Most of these examples are also available as interactive Jupyter notebooks in the Notebooks directory of the GitHub repository.



## REFERENCES & ACKNOWLEDGEMENTS

Papers making use of Bioverse should reference [Bixel & Apai \(2021\)](#). You should also include references to the [emcee](#) and [dynesty](#) packages, which are used for hypothesis testing and parameter fitting.

Bioverse was developed with support from the following grants and collaborations:

- [Alien Earths](#) & Earths in Other Solar Systems
- NASA Earth and Space Science Fellowship Program (grant No. 80NSSC17K0470)
- NASA's Nexus for Exoplanet System Science (NExSS)





## INSTALLATION

Bioverse can be cloned from its [GitHub repository](https://www.github.com/abixel/bioverse/):

```
git clone https://www.github.com/abixel/bioverse/
```

To install Bioverse, navigate to the directory containing `setup.py` and run:<sup>1</sup>

```
pip install .
```

---

<sup>1</sup> Bioverse will be added to PyPI in a future update.



## DEPENDENCIES

Bioverse is compatible with Python 3.7+. It has the following dependencies, all of which can be installed using pip:

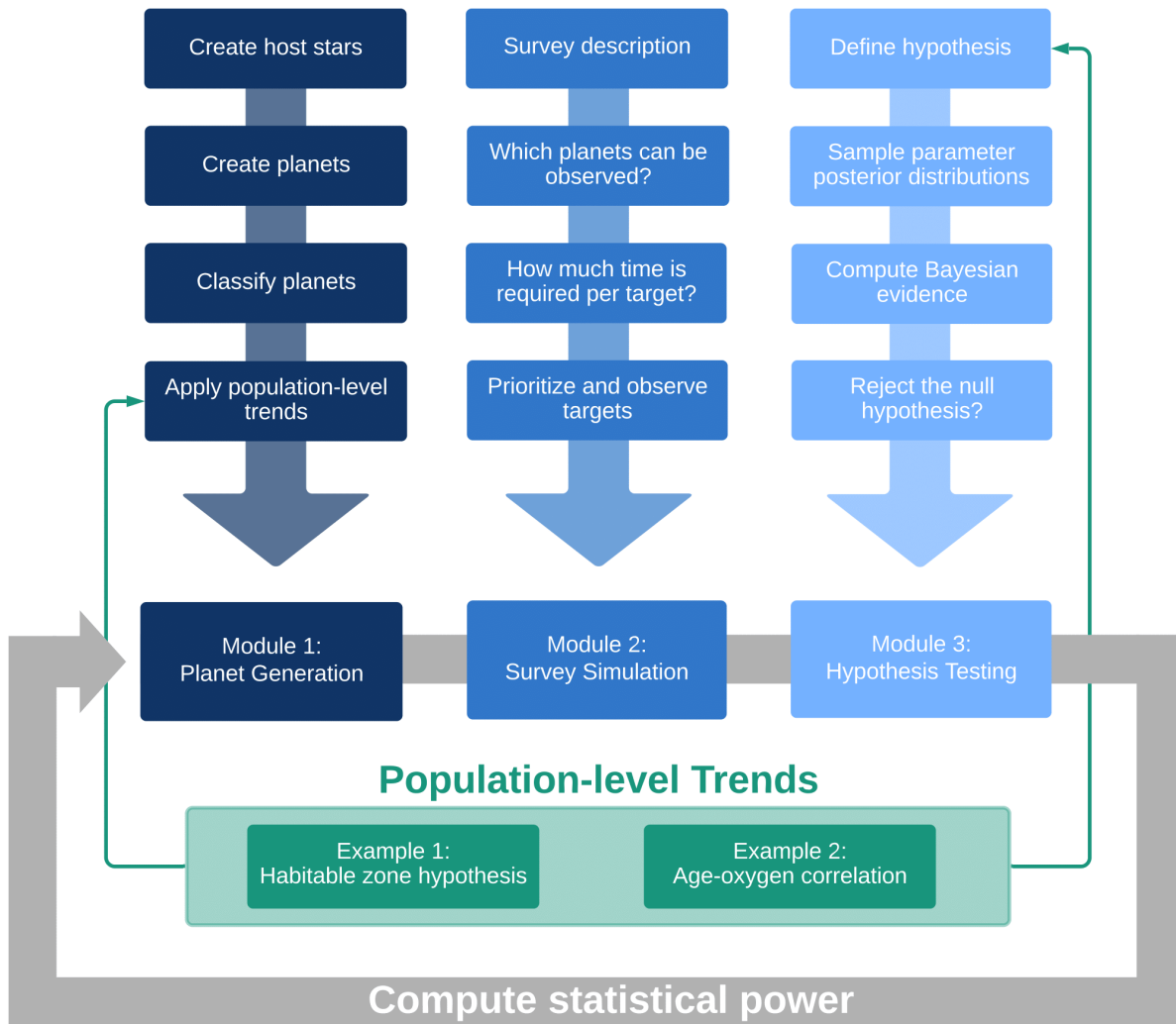
- `astroquery`
- `dynesty`
- `emcee`
- `matplotlib`
- `numpy`
- `scipy`
- `tqdm` (optional: provides a progress bar for long processes)
- `pandas` (optional: used for data visualization)
- `PyQt5` (optional: enables configuration GUI)

### 3.1 Introduction

Bioverse can be split into three primary components:

- The first module combines planet occurrence rates with empirical and hypothetical relationships between planet properties to produce a statistically realistic simulation of planetary systems in the solar neighborhood.
- The second module simulates a dataset (with uncertainties) that might be produced by an extensive direct imaging or transit spectroscopy survey of this exoplanet population.
- The third module applies statistical hypothesis tests to the simulated dataset in an attempt to uncover statistical relationships injected by the first module.

By iterating over these three modules as shown below, Bioverse can evaluate the potential of next-generation exoplanet surveys to answer key questions about exoplanet statistics.



## 3.2 The Table class

Bioverse uses the `Table` class to manage large simulated datasets across the code. Each row in the Table corresponds to a different simulated planet, while each column corresponds to a planetary parameter. Generally, rows correspond to indices while columns correspond to string keys. Some examples for selecting data in a table:

```

# Returns semi-major axis for every planet
table['a']

# Returns the mass of the tenth planet
table['M'][9]

# Returns all parameters for the first 50 planets
table[:50]

```

A Table is somewhat similar to a Pandas DataFrame. Indeed, if Pandas is installed, the Table will be displayed as one. To export a Table as a Pandas DataFrame, we can use the `to_pandas()` method:

```
table.to_pandas()

      d      M_st      R_st      L_st      T_eff_st SpT  ...  N_pl  order      R
      P      a      S
0  13.779  0.891276  0.912031  0.668409  5469.6313  G  ...  1      0  2.279597
680.071360  1.456531  0.315067
1  17.990  1.368140  1.285004  2.995380  6704.4496  F  ...  1      0  1.603131
4461.320947  5.887965  0.086402
2  21.648  1.384050  1.296945  3.119100  6741.3827  F  ...  1      0  3.114289
13.869061  0.125901  196.776529
3  17.565  0.892344  0.912906  0.671218  5472.7466  G  ...  1      0  1.246218
303.569497  0.851064  0.926700
4  3.563  0.397945  0.478474  0.039754  3729.2254  M  ...  4      0  2.560260
645.564506  1.075260  0.034384
...      ...      ...      ...      ...      ...  ..  ...  ...      ...
517  5.905  0.353798  0.435516  0.026342  3526.6437  M  ...  5      2  2.587812
128.221803  0.351970  0.212635
518  5.905  0.353798  0.435516  0.026342  3526.6437  M  ...  5      3  0.802393
165.421708  0.417119  0.151400
519  5.905  0.353798  0.435516  0.026342  3526.6437  M  ...  5      4  5.704183
311.476887  0.636036  0.065115
520  10.908  0.763220  0.805601  0.388396  5081.1400  K  ...  2      0  1.343379
1518.340450  2.362701  0.069576
521  10.908  0.763220  0.805601  0.388396  5081.1400  K  ...  2      1  0.513268
30.733272  0.175483  12.612595

[522 rows x 22 columns]
```

Each planet or stellar property is referred to throughout Bioverse by a unique string key. This formalism allows properties to be easily accessed across the code. The keys are not formally defined anywhere in the code, so creating a new property is as simple as adding it to a Table of planets:

```
# Assigns a random ocean covering fraction to every planet in the Table
table['f_ocean'] = np.random.uniform(0, 1, len(t))
```

This new column must have the same length as others in the Table. Some other examples of Table usage:

```
# Change the value of `f_ocean` to zero for planets that are not exo-Earth candidates
EEC = table['EEC'] # boolean array
table['f_ocean'][~EEC] = 0.

# Calculate planet densities in g/cm3
table['rho'] = 5.51 * table['M'] / table['R']**3

# List the definition of all keys in the table (found in legend.dat)
table.legend()

# Append one table to another in-place
table.append(table2, inplace=True)
```

See the [Table](#) documentation for a full list of its methods.

### 3.2.1 List of properties

The following table lists all keys currently used in Bioverse and the properties they correspond to:

## 3.3 Generating planetary systems

### 3.3.1 The Generator class

Bioverse uses the *Generator* class to generate planetary systems in the solar neighborhood. A Generator object specifies a list of functions to be performed in sequential order onto a shared *Table*. For example, a simple generator might implement this algorithm:

- Function 1: Return the [Gaia DR2](#) catalog of all stars within 30 parsecs with effective temperatures above 4000 K.
- Function 2: Simulate one or more planets around each star according to the occurrence rate estimates in [Bergsten et al. 2022](#).
- Function 3: Evaluate the mass of each planet based on its radius and the mass-radius relationship published by [Wolfgang et al. \(2016\)](#).

The generator will feed the output of Function 1 into Function 2, then the output of Function 2 into Function 3, and finally will return the output of Function 3 (i.e. a table of planets with known masses, radii, orbital properties, and host star properties).

Bioverse “ships” with two Generators: one for transit mode, and the other for imaging mode. The primary difference between the two is that the former uses the [Chabrier \(2003\)](#) stellar mass function to generate host stars, while the latter uses an optimized host star catalog for the LUVOIR direct imaging mission (see the [LUVOIR Final Report](#)). The following code demonstrates how to simulate a sample of planets using the imaging mode Generator:

```
from bioverse.generator import Generator
generator = Generator('imaging')
sample = generator.generate()
```

We can inspect the Generator to see which functions it implements:

```
# List the generator's steps
generator

Generator with 11 steps:
0: Function 'read_stellar_catalog' with 5 keyword arguments.
1: Function 'create_planets_bergsten' with 7 keyword arguments.
2: Function 'assign_orbital_elements' with 1 keyword arguments.
3: Function 'impact_parameter' with 1 keyword arguments.
4: Function 'assign_mass' with no keyword arguments.
5: Function 'compute_habitable_zone_boundaries' with no keyword arguments.
6: Function 'classify_planets' with no keyword arguments.
7: Function 'geometric_albedo' with 2 keyword arguments.
8: Function 'effective_values' with no keyword arguments.
9: Function 'Example1_water' with 3 keyword arguments.
10: Function 'Example2_oxygen' with 2 keyword arguments.
```

Each of these functions is documented under the *functions* module.

### 3.3.2 Passing keyword arguments

Many of the functions in the Generator accept keyword arguments that affect the properties of the simulated sample. For example, the `create_planets_bergsten()` function scales the planet occurrence rates uniformly in response to its keyword argument `f_eta`. To change the value of `f_eta`, simply pass it to `generate()` as follows:

```
sample = generator.generate(f_eta=1.5)
```

Note that this value will be passed to any function in the generator for which `f_eta` is an argument. This can be useful for sharing arguments across multiple functions, but be careful not to accidentally use the same keywords for two different functions.

### 3.3.3 Transit mode

One of Bioverse’s main functions is to evaluate the sample size of a transiting exoplanet survey. However, most planets do not transit their stars, so simulating their properties would be inefficient. The argument `transit_mode` can be used to address this:

```
sample = generator.generate(transit_mode=True)
```

If True, then only planets that transit their stars are simulated.

### 3.3.4 Adding new functions

You can extend a generator by writing your own functions to simulate new planetary properties. Each function must accept a `Table` as its first and only required argument, can accept any number of keyword arguments, and must return a `Table` as its only return value.

For example, the following function will assign a random ocean covering fraction to Earth-sized planets in the habitable zone (exo-Earth candidates or “EECs”), while non-EECs will have no oceans.

```
def make_oceans(table, f_ocean_min=0.05, f_ocean_max=0.8):
    # f_ocean=0 for all planets
    table['f_ocean'] = np.zeros(len(table))

    # f_ocean_min < f_ocean < f_ocean_max for EECs
    EECs = table['EEC']
    table['f_ocean'][EECs] = np.random.uniform(f_ocean_min, f_ocean_max, EECs.sum())

    return table
```

Save this function in `custom.py` and insert it into the Generator as follows:

```
generator.insert_step('make_oceans')
```

You can then simulate a sample of planets with oceans for arbitrary values of `f_ocean_min` and `f_ocean_max`:

```
sample = generator.generate(f_ocean_min=0.3, f_ocean_max=0.7)
```

You might also want to replace an existing step in the Generator with your own alternative. For example, suppose we want to replace the function that assigns planet masses (step 4: `assign_mass()`) with one that implements the mass-radius relationship of Weiss & Marcy (2014). First, define a function `Weiss_Marcy_2014()` in `custom.py` that implements this relationship using the format above. Next, we can replace step 4 with the new function:

```
# Remove step 4 and replace it with the new mass-radius relationship
del generator.steps[4]
generator.insert_step('Weiss_Marcy_2014', 4)
```

Note that the function `Weiss_Marcy_2014()` should also compute the density and surface gravity of each planet as `assign_mass()` currently does.

### 3.3.5 Saving and loading

You can save the modified version of a Generator under a new name:

```
generator.save('imaging_with_oceans')
```

and load it as follows:

```
generator = Generator('imaging_with_oceans')
```

## 3.4 Simulating survey datasets

### 3.4.1 The Survey class

The output of `generate()` is a *Table* containing the values of several parameters for planets within the bounds of the simulation. However, only a subset of these will be detectable by a transit or direct imaging survey. For those planets, only a subset of their properties can be directly probed, and only with a finite level of precision. Module 2 captures these details by simulating the observing limits and measurement precision of a direct imaging or transit spectroscopy survey of the planet population.

The survey simulation module is implemented by the *Survey* class<sup>1</sup> and its children classes *ImagingSurvey* and *TransitSurvey*. The Survey describes several key components of an exoplanet survey including:

- **diameter**: the diameter of the telescope primary in meters (or the area-equivalent diameter for a telescope array)
- **t\_slew**: slew time between observations, in days
- *ImagingSurvey*
  - **inner\_working\_angle** and **outer\_working\_angle**: IWA/OWA of the coronagraphic imager
  - **contrast\_limit**: log-contrast limit (i.e. faintest detectable planet)
- *TransitSurvey*
  - **N\_obs\_max**: maximum allowable number of transit observations per target
  - **t\_max**: maximum amount of time across which to combine transit observations, in days
- **T\_st\_ref**, **R\_st\_ref**, and **d\_ref**: temperature (Kelvin), radius ( $R_{\odot}$ ), and distance (parsec) of the reference star (see *Exposure time calculations*)
- **D\_ref**: diameter of the reference telescope, in meters

Each type of survey “ships” with a default configuration:

---

<sup>1</sup> *Survey* should never be called directly; instead *ImagingSurvey* or *TransitSurvey* should be used.



```
from bioverse.survey import ImagingSurvey, TransitSurvey
survey_imaging = ImagingSurvey('default')
survey_transit = TransitSurvey('default')
```

The default imaging survey is modeled after [LUVOIR-A](#), with a coronagraphic imager and 15-meter primary aperture. The default transit survey is modeled after the Nautilus Space Observatory, with a 50-meter equivalent light-collecting area.

### 3.4.2 Which planets are detectable?

Given a simulated set of planets to observe, the Survey first determines which of these are detectable. For a [TransitSurvey](#), this set consists of all transiting planets, while for an [ImagingSurvey](#), it consists of all planets within the coronagraphic IWA/OWA and brighter than the limiting contrast. This can be invoked as follows

```
detected = survey_imaging.compute_yield(sample)
```

### 3.4.3 Conducting measurements

The Survey will conduct a series of measurements on the detectable planet sample, each defined by a [Measurement](#) object. A Measurement's parameters include:

- **key**: the name of the planet property to be measured
- **precision**: the relative or absolute precision of the measurement (e.g. 10% or 0.1 AU)
- **t\_ref**: the amount of time in days required to conduct this measurement for a typical target (see below)
- **t\_total**: the amount of survey time in days allocated toward this measurement
- **wl\_eff**: the effective wavelength of observation in microns
- **priority**: a set of rules describing how targets are prioritized (described below)

To conduct these measurements and produce a dataset:

```
data = survey_imaging.observe(detected)
```

### 3.4.4 Quick-run

In total, to produce a simulated sample of planets, determine which planets are detectable, and produce a mock dataset requires the following:

```
from bioverse.generator import Generator
from bioverse.survey import ImagingSurvey

generator = Generator('imaging')
survey = ImagingSurvey('default')

sample = generator.generate(eta_Earth=0.15)
detected = survey.compute_yield(sample)
data = survey.observe(detected)
```

The last three lines can be combined into the following:

```
sample, detected, data = survey.quickrun(generator, eta_Earth=0.15)
```

`quickrun()` will pass any keyword arguments to the `generate()` method, and will by default pass `transit_mode=True` for a `TransitSurvey`.

### 3.4.5 Exposure time calculations

Spectroscopic observations of exoplanets are time-consuming, and for some surveys the amount of time required to conduct them will be a limiting factor on sample size. To accommodate this, Bioverse calculates the exposure time  $t_i$  required to conduct the spectroscopic measurement for each planet, then prioritizes each planet according to  $t_i$  as well as its weight parameter (see [Target prioritization](#)). In the simulated dataset, planets that could not be observed within the total allotted time `t_total` will have nan values for the measured value.

A Measurement's "reference time", `t_ref`, is the exposure time required to perform the measurement for an Earth-like planet (receiving the same flux as Earth) orbiting a typical star (whose properties are defined by the Survey parameters `T_st_ref`, `R_st_ref`, and `d_ref`), with a telescope of diameter `D_ref`. For the default imaging survey, the typical target orbits a Sun-like star at a distance of 10 pc, while for the transit survey, the host star is a mid-M dwarf.

Bioverse uses `t_ref`, along the wavelength of observation `wl_eff`, to determine the exposure time `t_i` required for each individual planet with the following equation:

$$\frac{t_i}{t_{\text{ref}}} = f_i \left( \frac{d_i}{d_{\text{ref}}} \right)^2 \left( \frac{R_*}{R_{*,\text{ref}}} \right)^{-2} \left( \frac{B(\lambda_{\text{eff}}, T_{*,i})}{B(\lambda_{\text{eff}}, T_{*,\text{ref}})} \right)^{-1} \left( \frac{D}{D_{\text{ref}}} \right)^{-2}$$

$f_i$  encompasses the different factors affecting spectroscopic signal strength in imaging and transit mode:

$$f_i^{\text{imaging}} = \left( \frac{\zeta_i}{\zeta_{\oplus}} \right)^{-1}$$

$$f_i^{\text{transit}} = \left( \frac{h_i}{h_{\oplus}} \right)^{-2} \left( \frac{R_{p,i}}{R_{\oplus}} \right)^{-2} \left( \frac{R_{*,i}}{R_{*,\text{ref}}} \right)^4$$

Importantly, this calculation is conducted for each Measurement with a different value of `t_ref`. **Therefore, the same planet may have real values for one Measurement and ``nan`` for another.** This is particularly relevant for the transit survey, where the total number of transiting planets for which e.g. planet size and orbital period can be measured is much larger than the number that can be spectroscopically characterized. To return just the subset of detected planets that were observed for a given Measurement, use the `observed()` method:

```
observed = data.observed('has_02')
```

The determination of `t_ref` often relies on radiative transfer and instrument noise estimates that are generally not done in Bioverse. It can be accomplished by citing relevant studies in the literature or using third-party tools such as the [Planetary Spectrum Generator](#). One method of calculating `t_ref` for the transit survey is demonstrated in [Tutorial 3: Calculating exposure times](#).

Bioverse can calculate `t_ref` given two simulated spectra files - one with and one without the targeted absorption feature - both of which contain measurements for wavelength, flux, and flux uncertainty as the first three columns. You must also specify the simulated exposure time and the minimum and maximum wavelengths for the absorption feature. The `compute_t_ref()` function will then determine the exposure time required for a 5-sigma detection (in the same units as the input exposure time).

```
from bioverse.util import compute_t_ref

# Scales from simulated spectra for a combined 100 hr exposure time, targeting the 03_
# feature near 0.6 microns.
```

(continues on next page)

(continued from previous page)

```
t_ref = compute_t_ref(filenamees=('spectrum_03.dat', 'spectrum_no03.dat'), t_exp=100, wl_min=0.4, wl_max=0.8)
print("Required exposure time: {:.1f} hr".format(t_ref))
```

Output: Required exposure time: 73.9 hr

Finally, change the `t_ref` and `wl_eff` attributes of the associated Measurement object, using units of days and microns respectively:

```
survey = TransitSurvey('default')
survey.measurements['has_02'].t_ref = 73.9/24
survey.measurements['has_02'].wl_eff = 0.6
```

### 3.4.6 Target prioritization

For measurements where `t_total` is finite and `t_ref` is non-zero, targets must be prioritized in case there is insufficient time to characterize all of them. In Bioverse, target prioritization depends both on the target's scientific interest (quantified by the weight parameter `w_i`) and the amount of time `t_i` required to properly characterize it. Each target's priority is calculated as follows:

$$p_i = w_i / t_i$$

Bioverse will observe targets in order of decreasing `p_i` until `t_total` has been exhausted. The resulting dataset will fill in `nan` values for any targets that were not observed.

By default, `w_i = 1` for all targets, but it can be raised or lowered for planets that meet certain criteria. For example, to assign `w_i = 5` for targets with radii between 1-2  $R_{\oplus}$ :

```
m = survey.measurement['has_02']
m.set_weight('R', weight=5, min=1, max=2)
```

To exclude a set of targets, set `w_i = 0`. For example, to restrict a measurement to exo-Earth candidates only:

```
m.set_weight('EEC', weight=0, value=False)
```

In transit mode, targets are weighted by  $a/R_*$  to correct the detection bias toward shorter period planets. To disable this feature:

```
m.debias = False
```

## 3.5 Hypothesis testing

### 3.5.1 The Hypothesis class

The combined result of the first two modules is a simulated dataset representing the output of the exoplanet survey. The third module addresses the power of that dataset for testing statistical hypotheses. The first step in this exercise involves defining the hypotheses you want to test, and in Bioverse this is done via a *Hypothesis* object. To define a Hypothesis requires:

- a set of dependent variable(s)  $X$ , called features
- a set of independent variable(s)  $Y$ , called labels

- a set of parameters `theta`
- a Python function describing the quantitative relationship between `X` and `Y` in terms of `theta`
- the prior distribution of values of `theta`
- an alternative (or null) hypothesis against which to test

For example, consider the hypothesis that planet mass and radius can be related by a simple power law:  $M(R|M_0, \alpha) = M_0 R^\alpha$ . In this case, `X = R`, `Y = M`, and `theta = (M_0, alpha)`.

The first step in defining this hypothesis is to write out the function `Y = f(X | theta)`:

```
def f(theta, X):
    M_0, alpha = theta
    R, = X
    return M_0 * R ** alpha

params = ('M_0', 'alpha')
features = ('R',)
labels = ('M',)
```

The tuples `features` and `labels` tell the code which parameters to extract from the simulated dataset. In this case, planet radius (`R`) and mass (`M`) will be extracted from the simulated dataset as the features and labels.

We must define the bounds on values for `M_0` and `alpha` - conservative constraints might be  $0.1 < M_0 < 10$  and  $2 < \alpha < 5$ . We will also choose a log-uniform distribution for `M_0`, as its bounds span a few orders of magnitude.

```
bounds = np.array([[0.1, 10], [2, 5]])
log = (True, False)
```

Next, we can initialize the Hypothesis:

```
from bioverse.hypothesis import Hypothesis
h_mass_radius = Hypothesis(f, bounds, params=params, features=features, labels=labels,
    log=log)
```

### 3.5.2 The null hypothesis

In order to test the evidence in favor of `h_mass_radius`, we must define an alternative (or “null”) hypothesis<sup>1</sup>. In this case, the hypothesis states that planetary mass is independent of radius, and ranges from 0.01 and 100 `M_Earth` (with average value `M_random`):

```
def f_null(theta, X):
    shape = (np.shape(X)[0], 1)
    return np.full(shape, theta)

bounds_null = np.array([[0.01, 100]])
h_mass_radius.h_null = Hypothesis(f_null, bounds, params=('M_random',), log=(True,))
```

<sup>1</sup> Note that `bioverse.hypothesis.f_null()` provides the same function as `f_null()` above but for an arbitrary number of parameters, features, and labels.

### 3.5.3 Testing the hypothesis

Next, we can test `h_mass_radius` using a dataset from the previous examples:

```
results = h_mass_radius.fit(data)
```

The `fit()` method will pull the measured values of ‘R’ and ‘M’ and test them using one or more of the following methods (set by the `method` keyword):

- `method = dynesty` (default) Uses nested sampling to sample the parameter space of `theta` and compute the Bayesian evidence for both the Hypothesis and the null hypothesis. Implemented by `dynesty`.
- `method = emcee` Uses Markov Chain Monte Carlo to sample the parameter space of `theta`. Implemented by `emcee`.
- `method = mannwhitney` Assuming `X` to be a single continuous variable and `Y` a single boolean, reports the probability that `X[Y]` and `X[~Y]` are drawn from the same parent distribution. Implemented by `scipy`.

By default, nested sampling is used to estimate the Bayesian evidence in favor of the Hypothesis in comparison to the null hypothesis.

### 3.5.4 Likelihood functions

Both `dynesty` and `emcee` require a Bayesian likelihood function to be defined. The likelihood function is proportional to the probability that `Y` would be drawn given `X` and a set of values for `theta`. Currently, two likelihood functions are supported:

- `binomial`: If `Y` is a single boolean parameter (e.g., ‘has\_H2O’) then `f` is interpreted as the likelihood that `Y == 1` given `X`. In this case the likelihood function is:

$$\ln \mathcal{L} = \sum_i \ln (Y_i f(X|\theta) + (1 - Y_i) f(X|\theta))$$

- `multivariate`: If `Y` is one or more continuous variables then `f` is interpreted as the expectation values of `Y` given `X`. In this case the likelihood function is the multivariate Gaussian:

$$\ln \mathcal{L} = \sum_i \left[ -(Y_i - f(X|\theta))^2 / (2\sigma_i^2) \right]$$

### 3.5.5 Prior distributions

The prior distributions of the parameters `theta` can be set to either uniform or log-uniform functions *or* defined by the user<sup>2</sup>. For uniform and log-uniform, only the boundaries of these distributions must be given:

```
# For theta = (M_0, alpha)
bounds = np.array([[0.1, 10], [2, 5]])

# Log-uniform distribution for M_0, uniform distribution for alpha
h_mass_radius = Hypothesis(f, bounds, log=(True, False))
```

Non-uniform prior distributions can be defined by the user, but they must be given in the proper format for both `dynesty` and `emcee`:

```
h_mass_radius = Hypothesis(f, bounds, tfprior_function=tfprior, lnprior_function=lnprior)
```

For more details on how to define `tfprior()` and `lnprior()`, see the documentation for `dynesty` and `emcee` respectively.

<sup>2</sup> Documentation for user-defined priors will be added in a future update.

### 3.5.6 Posterior distributions

When using `dynesty` or `emcee`, the `results` object will contain summary statistics of the posterior distributions for the values of `theta`, including the mean, median, and lower and upper 95% confidence intervals. Alternatively, by passing `return_chains = True` to the `fit()` method, the entire chain of sampled values will be returned. Given enough time, the distribution of these values will converge onto the posterior distribution. In general, `emcee` converges much more efficiently and should be used to estimate (for example) the precision with which model parameters can be constrained.

## 3.6 Computing statistical power

Consider the “habitable zone hypothesis”, which proposes that habitable planets with atmospheric water vapor will be more common within the semi-major axis range  $a_{\text{inner}} < a_{\text{eff}} < a_{\text{outer}}$  (see Section 6 of the paper and Example 1 for more details). In Bioverse, this effect is injected into the simulated sample by the `Example1_water()` function, and tested using the `h_HZ` Hypothesis. To test this hypothesis using a LUVOIR-like direct imaging survey:

```
from bioverse.generator import Generator
from bioverse.survey import ImagingSurvey
from bioverse.hypothesis import h_HZ

# Load the Generator and Survey objects
generator = Generator('imaging')
survey = ImagingSurvey('default')

# Generate a set of planetary systems and a simulated dataset as observed by an imaging_
# survey
# Assume 50% of EECs are habitable (f_water_habitable=0.5)
# Assume 1% of non-habitable planets have water vapor (f_water_nonhabitable=0.01)
sample, detected, data = survey.quickrun(generator, f_water_habitable=0.5, f_water_
# nonhabitable=0.01)

# Test the habitable zone hypothesis from this dataset
results = h_HZ.fit(data)

print("The evidence in favor of the habitable zone hypothesis is {:.1f}.".format(results[
# 'dlnZ']))
```

Output: The evidence in favor of the habitable zone hypothesis is 6.4.

This result corresponds to a “p-value” of  $\sim 1.7\text{E-}3$ . However, this represents only one possible realization of the survey. Due to Poisson uncertainty, another equivalent survey might detect fewer habitable planets and thus be less capable of testing the hypothesis. To capture this, we can repeat the survey several times and average their results. The `analysis` module enables this through its `test_hypothesis_grid()` function.

```
from bioverse import analysis

# Repeat the hypothesis test N=30 times with the same assumptions as above
results = analysis.test_hypothesis_grid(h_HZ, generator, survey, N=30, f_water_
# habitable=0.5,
# f_water_nonhabitable=0.01, processes=8)

# Determine the statistical power assuming a significance threshold of dlnZ > 3
```

(continues on next page)

(continued from previous page)

```
power = analysis.compute_statistical_power(results, method='dlnZ', threshold=3)

print("The statistical power of the survey is {:.1f}%".format(100*power))
```

Output: The statistical power of the survey is 75.0%.

Under the assumptions that 50% of exo-Earth candidates are habitable **and** 1% of non-habitable planets have H<sub>2</sub>O in their atmospheres, it is 75% likely that a LUVOIR-like survey would be able to detect the overabundance of H<sub>2</sub>O in the habitable zone.

### 3.6.1 Parameter grids

Of course, those assumptions are highly uncertain, and a more thorough analysis should investigate how this result depends on key model parameters - such as `f_water_habitable` or `eta_Earth`. This can be done by passing an array of values for these parameters to the `test_hypothesis_grid()` function:

```
# Vary the fraction of EECs with water vapor from 1% to 100% (log spacing)
f_water_habitable = np.logspace(-2, 0, 5)

# Vary eta Earth from 7.5% to 30% (linear spacing)
eta_Earth = np.linspace(0.075, 0.3, 5)

# Test the hypothesis N=30 times for each parameter combination
results = analysis.test_hypothesis_grid(h_HZ, generator, survey, N=30, f_water_
↳habitable=f_water_habitable,
                                eta_Earth=eta_Earth, f_water_nonhabitable=0.01,
↳processes=8)

# Compute the statistical power for each parameter combination
power = analysis.compute_statistical_power(results, method='dlnZ', threshold=3)
```

`power` will be a 5x5 array containing the statistical power for each parameter combination. The axis order depends on the order in which arguments are passed to `test_hypothesis_grid()`; in this case, `f_water_habitable` will correspond to the first axis and `eta_Earth` to the second.

### 3.6.2 Plotting the results

The `plot_power_grid()` function can be used to plot the statistical power over a 2-dimensional grid. Starting from the above example:

```
from bioverse.plots import plot_power_grid

# Specify which parameters to plot on the x and y axes
axes = ('f_water_habitable', 'eta_Earth')

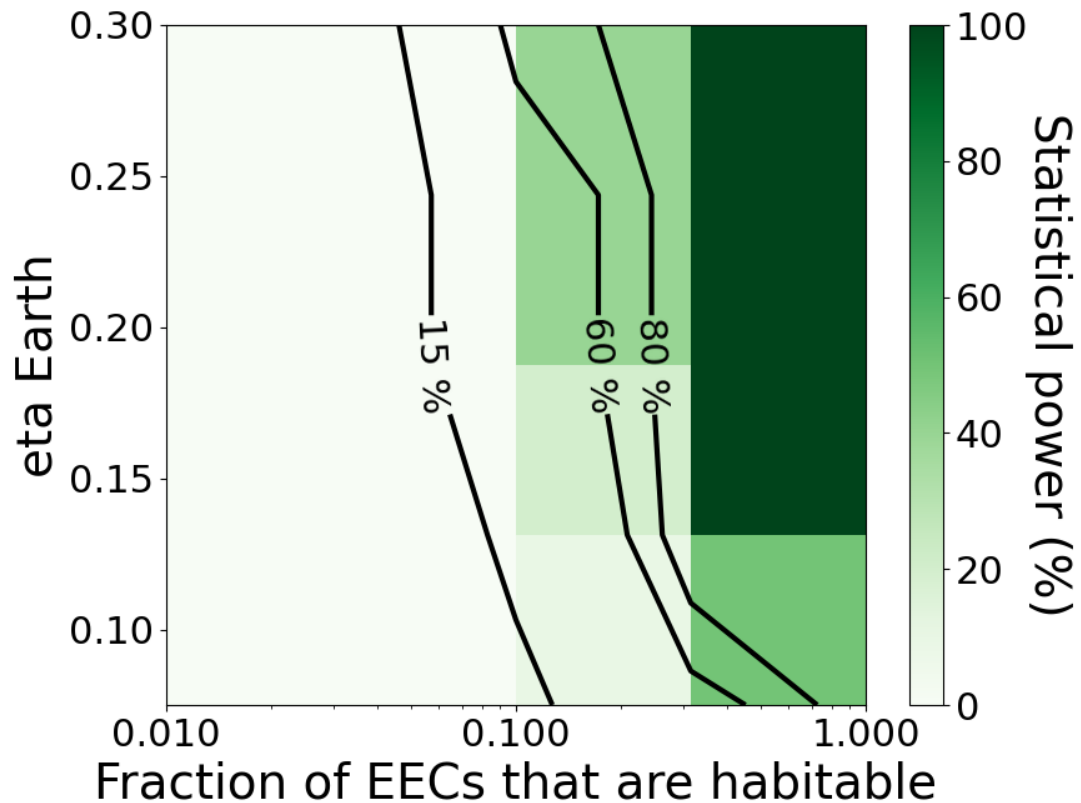
# Set the axis labels
labels = ('Fraction of EECs that are habitable', 'eta Earth')

# Set log-scale for the x axis
log = (True, False)
```

(continues on next page)

(continued from previous page)

```
# Create the plot
plot_power_grid(results, axes=axes, labels=labels, log=log)
```



The number and percentage values of the contour lines can be set with the `levels` argument, or set `levels=None` to disable them. To create a higher resolution plot with smoother contour lines, simply run `test_hypothesis_grid()` over a finer grid of parameter values.

### 3.6.3 Multiprocessing

To compute the statistical power for a 20x20 parameter grid with  $N=50$  simulations in each cell requires 20,000 simulations, or approximately 5-6 hours for the example above. Fortunately, these simulations are entirely independent of each other, making parallel processing an effective solution. You can use the `processes` argument of `test_hypothesis_grid()` to indicate how many processes to run in parallel. Note that Bioverse can be memory-intensive, so large values of `processes` (e.g. greater than 10) can have diminishing returns or lead to a crash.



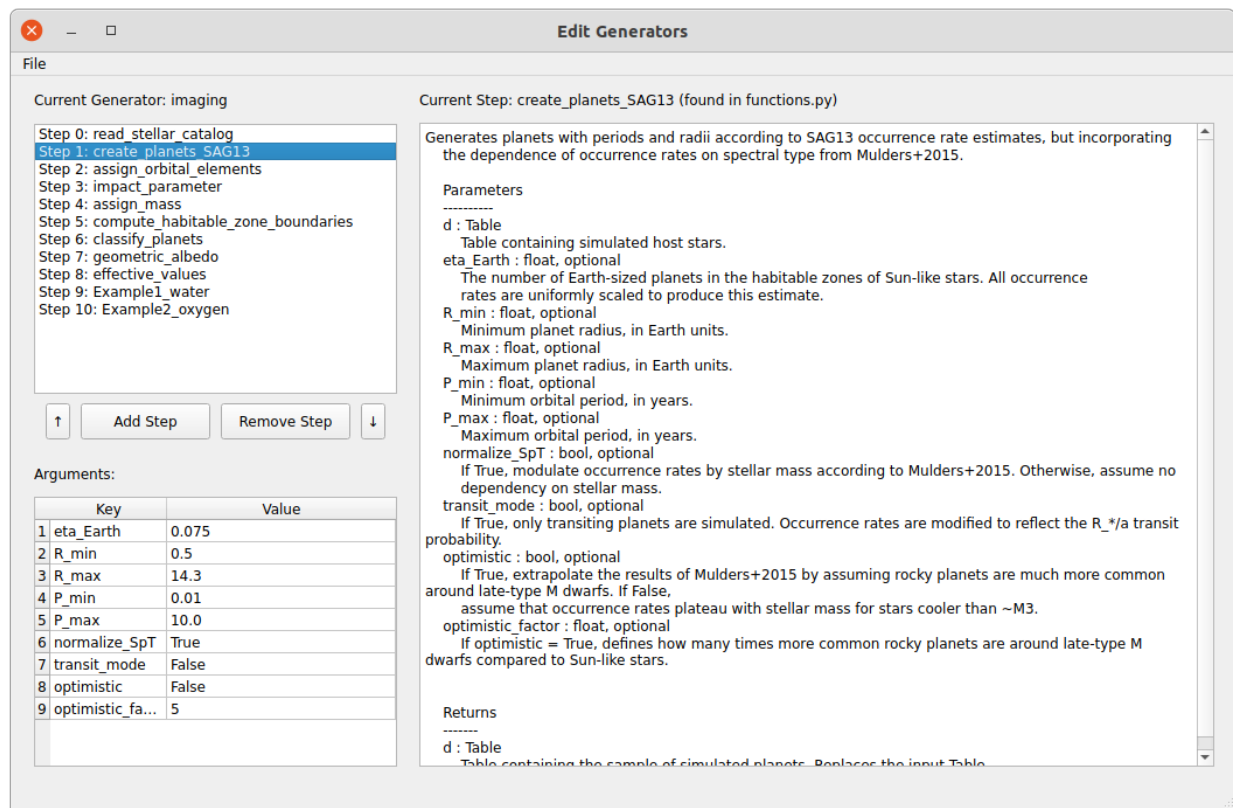
## 3.7 Object Editor

The object editor is a GUI that allows the user to edit the default *Generator* and *Survey* objects and save new configurations. It can be opened as follows:

```
from bioverse import gui
gui.show()
```

### 3.7.1 Editing Generators

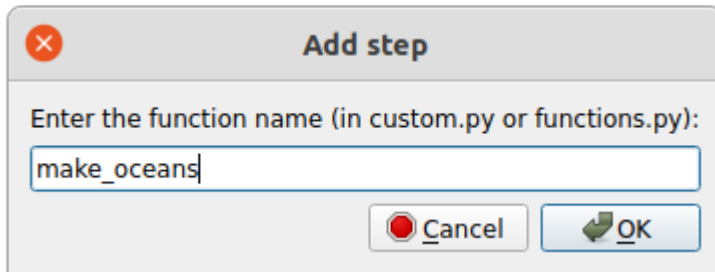
To make changes to a Generator object, first load its associated .pkl file using File > Load. This will load the list of steps performed by the Generator into the upper left box. You can click through each step to see a description of each function and its arguments on the right:



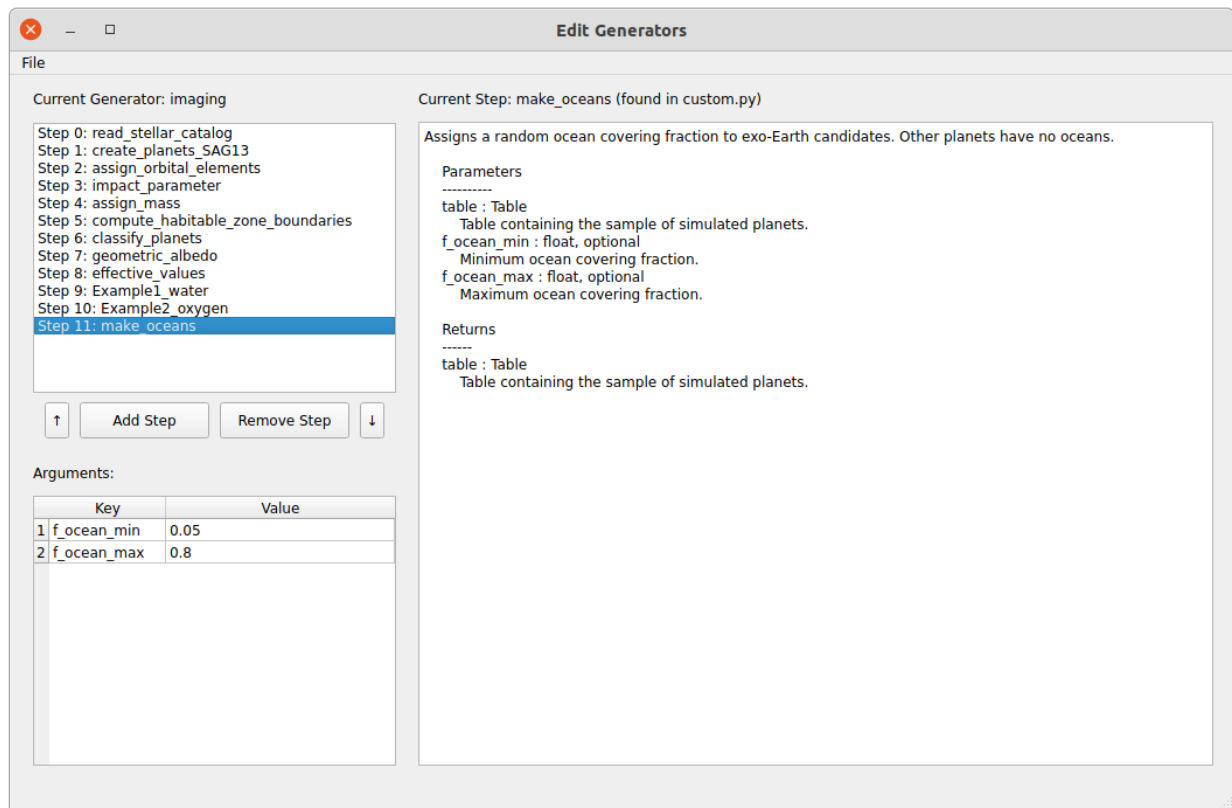
The lower-left table shows the keyword arguments for the currently-selected function along with their current default values. To edit the default value, simply double-click and enter a new value. Note that this will update the argument value for *all* steps that accept this argument.

## Editing functions

To add a new function to the Generator, first define it in `custom.py` as described in [Adding new functions](#), then enter the name of the function using the Add Step button:



The GUI will add your function to the end of the list:



Use the arrow keys to edit the order in which functions are called. Finally, to remove a function from the Generator, use the Remove Step button.

Once you have made your changes to the Generator, you can save it using `File > Save as...` and use it in Bioverse:

```
# Loads my_generator.pkl
from bioverse.generator import Generator
generator = Generator('my_generator')
```

### 3.7.2 Editing Surveys

To make changes to an ImagingSurvey or TransitSurvey object, first load its associated .pkl file using File > Load. You can then edit the Survey's properties in the top left and add, remove, or re-order measurements in the bottom left.

The 'Edit Surveys' window is divided into two main panels. The left panel shows the 'Current ImagingSurvey: default' with a table of parameters and a list of measurements. The right panel shows the 'Current Measurement: has\_O2' with a table of parameters and a list of priorities.

**Current ImagingSurvey: default**

	Parameter	Value
1	diameter	15.0
2	t_max	3652.5
3	t_slew	0.1
4	T_st_ref	5788.0
5	R_st_ref	1.0
6	D_ref	15.0
7	d_ref	10.0
8	inner_workin...	3.5
9	outer_workin...	64.0
10	contrast_limit	-10.6

**Measurements:**

- Measurement 0: L\_st
- Measurement 1: R\_st
- Measurement 2: T\_eff\_st
- Measurement 3: d
- Measurement 4: contrast
- Measurement 5: a
- Measurement 6: has\_H2O
- Measurement 7: age
- Measurement 8: EEC
- Measurement 9: has\_O2**

**Current Measurement: has\_O2**

	Parameter	Value
1	precision	0.0
2	t_total	None
3	t_ref	0.1
4	t_min	0.0
5	wl_eff	0.7
6	debias	True

**Priority:**

	Condition	Weight
1	0.00 < age < 1.00	10
2	1.00 < age < 2.00	5
3	2.00 < age < 10.00	1
4	EEC == False	0

Buttons at the bottom left: ↑, Add Measurement, Remove Measurement, ↓. Buttons at the bottom right: Add Prioritization, Remove Prioritization.

Select a measurement in the bottom left to view its details on the right side of the interface, including its key parameters (top right) and prioritization scheme (bottom right).

## Target prioritization

Each measurement assigns a weight to each potential target that determines the order in which it is observed. By default, all planets have weight = 1. In the image above, for the measurement of 'has\_O2', planets with younger ages have higher weight while planets that are not exo-Earth candidates are not observed (weight = 0).

The Add Prioritization button allows you to define a new condition for assigning weight. In the following, we assign weight = 5 to planets with host star temperatures higher than 3000 K:

## 3.8 Tutorial 1: Generating planetary systems

In this tutorial, we will review how to use the Generator class to generate a sample of planetary systems, including how to add or replace steps in the process.

### 3.8.1 Setup

Let's start by importing the necessary module from Bioverse.

```
[1]: # Import numpy
import numpy as np

# Import the Generator class
from bioverse.generator import Generator
from bioverse.constants import ROOT_DIR

# Import pyplot (for making plots later) and adjust some of its settings
from matplotlib import pyplot as plt
%matplotlib inline
plt.rcParams['font.size'] = 20.
```

### 3.8.2 Loading the Generator

The first step in using Bioverse is to generate a simulated sample of planetary systems. This is accomplished with a Generator object, of which two come pre-installed. Let's open one of them and examine its contents.

```
[2]: # Open the transit mode generator and display its properties
generator = Generator('transit')
generator
```

```
[2]: Generator with 12 steps:
0: Function 'create_stars_Gaia' with 6 keyword arguments.
1: Function 'create_planets_SAG13' with 9 keyword arguments.
2: Function 'assign_orbital_elements' with 1 keyword arguments.
3: Function 'geometric_albedo' with 2 keyword arguments.
4: Function 'impact_parameter' with 1 keyword arguments.
5: Function 'assign_mass' with no keyword arguments.
6: Function 'compute_habitable_zone_boundaries' with no keyword arguments.
7: Function 'compute_transit_params' with no keyword arguments.
8: Function 'classify_planets' with no keyword arguments.
9: Function 'scale_height' with no keyword arguments.
10: Function 'Example1_water' with 3 keyword arguments.
11: Function 'Example2_oxygen' with 2 keyword arguments.
```

The list above shows each of the steps the Generator runs through in producing the sample of planetary systems. For more information about an individual step, we can display it based on its index:

```
[3]: # Show more about the 'create_planets_SAG13' step
generator.steps[1]
```

```
[3]: Function 'create_planets_SAG13' with 9 keyword arguments.
```

Description:

Generates planets with periods and radii according to SAG13 occurrence rate, estimates, but incorporating the dependence of occurrence rates on spectral type from Mulders+2015.

Parameters

-----

d : Table

Table containing simulated host stars.

eta\_Earth : float, optional

The number of Earth-sized planets in the habitable zones of Sun-like stars. All occurrence rates are uniformly scaled to produce this estimate.

R\_min : float, optional

Minimum planet radius, in Earth units.

R\_max : float, optional

Maximum planet radius, in Earth units.

P\_min : float, optional

Minimum orbital period, in years.

P\_max : float, optional

Maximum orbital period, in years.

normalize\_SpT : bool, optional

If True, modulate occurrence rates by stellar mass according to Mulders+2015. Otherwise, assume no

(continues on next page)

(continued from previous page)

```

    dependency on stellar mass.
    transit_mode : bool, optional
        If True, only transiting planets are simulated. Occurrence rates are modified to
        ↪reflect the  $R_*/a$  transit probability.
    optimistic : bool, optional
        If True, extrapolate the results of Mulders+2015 by assuming rocky planets are
        ↪much more common around late-type M dwarfs. If False,
        assume that occurrence rates plateau with stellar mass for stars cooler than ~M3.
    optimistic_factor : float, optional
        If optimistic = True, defines how many times more common rocky planets are
        ↪around late-type M dwarfs compared to Sun-like stars.

Returns
-----
d : Table
    Table containing the sample of simulated planets. Replaces the input Table.

Argument values:
    eta_Earth      = 0.075
    R_min          = 0.5
    R_max          = 14.3
    P_min          = 0.01
    P_max          = 10.0
    normalize_SpT  = True
    transit_mode   = True
    optimistic     = False
    optimistic_factor = 5

```

Here we see a short description of the planet generation step as well as its keyword arguments, including `eta_Earth`. This specifies the average number of Earth-sized planets in the habitable zones of Sun-like stars, and it is currently set to 7.5% (following Pascucci et al. 2020). Let's change this to a more optimistic value:

```
[4]: # Set eta_Earth = 15%
generator.set_arg('eta_Earth', 0.15)
```

Some arguments are shared by multiple functions - for example, the `transit_mode` argument tells a couple of functions that only transiting planets are being simulated. By default, it is set to `False`, but we can change that like so:

```
[5]: # Set transit_mode = True for all functions that use it
generator.set_arg('transit_mode', True)

# We can also check on an argument's value as follows:
val = str(generator.get_arg('transit_mode'))
print("\nThe current value of transit_mode is {}".format(val))
```

The current value of `transit_mode` is `True`

### 3.8.3 Running the Generator

Now, let's use this Generator object to produce an ensemble of transiting planets within 100 parsecs.

```
[6]: sample = generator.generate(d_max=100)
print("Generated a sample of {:d} transiting planets including {:d} exo-Earth candidates.
↪".format(len(sample), sample['EEC'].sum()))
```

Generated a sample of 17081 transiting planets including 1115 exo-Earth candidates.

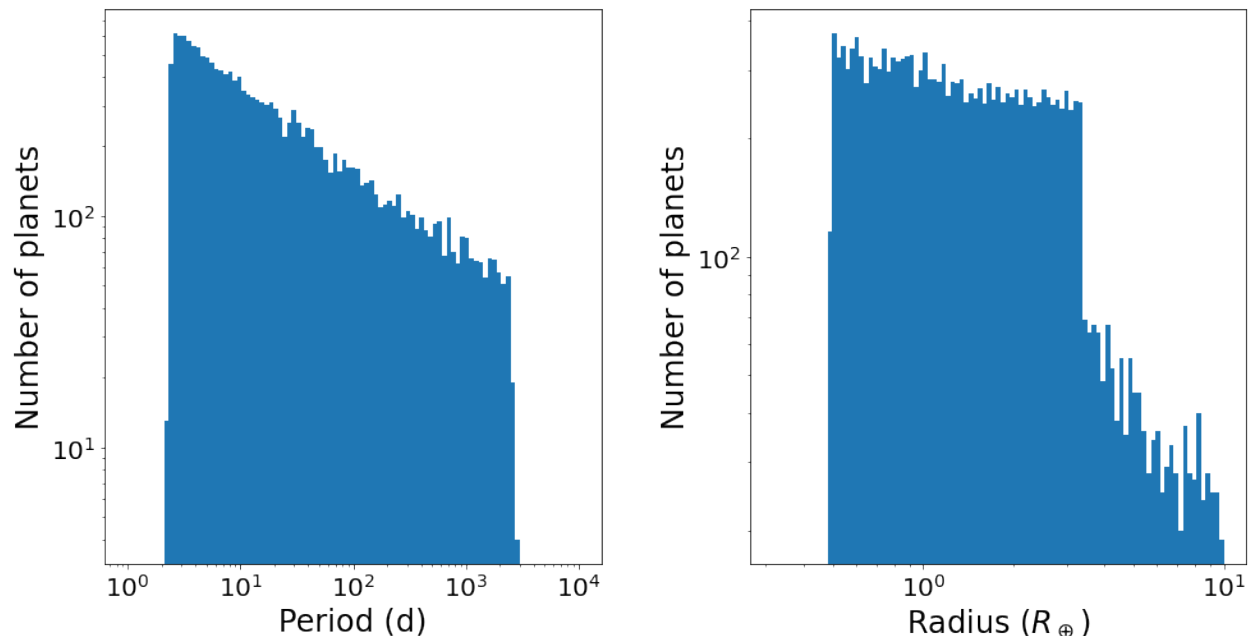
Let's plot the period and radius distribution of this planet sample:

```
[7]: fig, ax = plt.subplots(ncols=2, figsize=(16,8))

# Period histogram
P = sample['P']
bins = np.logspace(0, 4, 100)
ax[0].hist(P, bins=bins)
ax[0].set_xscale('log')
ax[0].set_yscale('log')
ax[0].set_xlabel('Period (d)', fontsize=24)
ax[0].set_ylabel('Number of planets', fontsize=24)

# Radius histogram
R = sample['R']
bins = np.logspace(-0.5, 1, 100)
ax[1].hist(R, bins=bins)
ax[1].set_xscale('log')
ax[1].set_yscale('log')
ax[1].set_xlabel('Radius ($R_{\oplus}$)', fontsize=24)
ax[1].set_ylabel('Number of planets', fontsize=24)

plt.subplots_adjust(wspace=0.3)
```



### 3.8.4 Adding new steps

Finally, suppose you want to simulate a new planetary property in Bioverse. For example, suppose you want to assign an ocean covering fraction to each planet ( $f_{\text{ocean\_min}} < f_{\text{ocean}} < f_{\text{ocean\_max}}$  for exo-Earth candidates and  $f_{\text{ocean}} = 0$  for others). We can accomplish this like so:

```
[8]: # Define a function that accepts and returns a Table object
def oceans(d, f_ocean_min=0, f_ocean_max=1):
    # First, assign zero for all planets
    d['f_ocean'] = np.zeros(len(d))

    # Next, assign a random non-zero value for exo-Earth candidates
    EEC = d['EEC']
    d['f_ocean'][EEC] = np.random.uniform(f_ocean_min, f_ocean_max, size=EEC.sum())

    # Finally, return the Table with its new column
    return d

# Insert this function at the end of the Generator
generator.insert_step(oceans)

# Run the generator with f_ocean_min = 0.3 and f_ocean_max = 0.7
sample = generator.generate(d_max=100, f_ocean_min=0.3, f_ocean_max=0.7)
```

Now, let's plot the distribution of ocean covering fractions for EECs and non-EECs:

```
[9]: fig, ax = plt.subplots(ncols=2, figsize=(16,8))

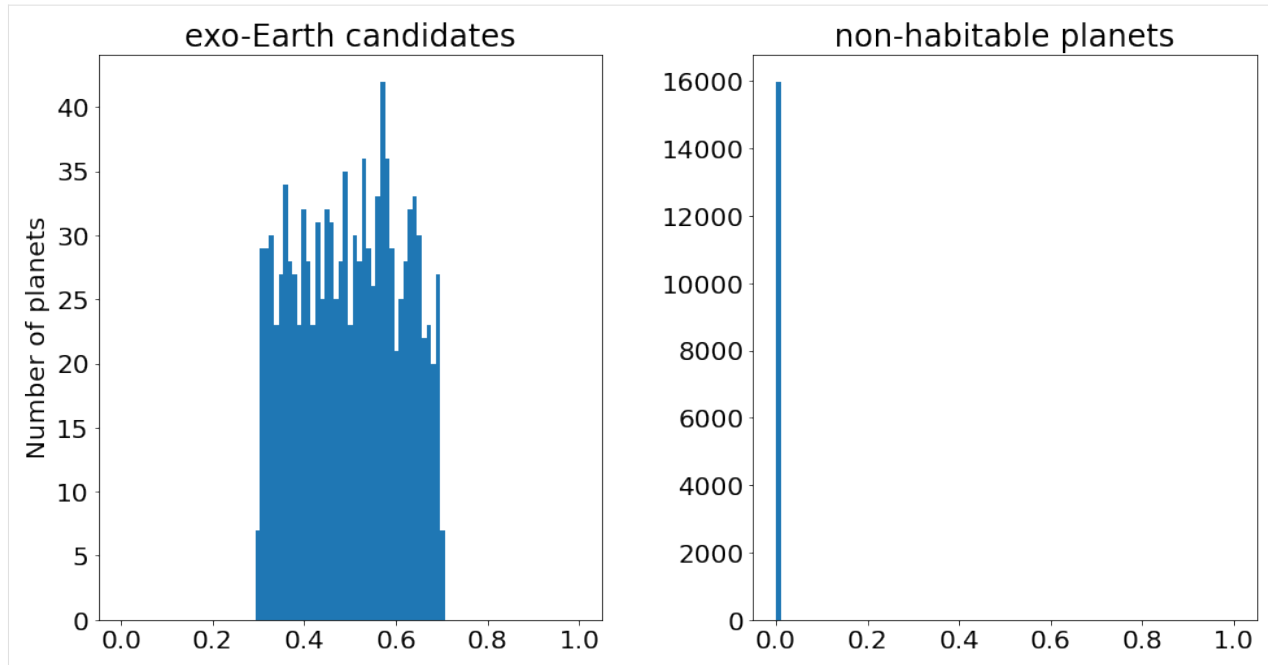
f_ocean = sample['f_ocean']
EEC = sample['EEC']

# EECs
bins = np.linspace(0., 1, 100)
ax[0].hist(f_ocean[EEC], bins=bins)
ax[0].set_title('exo-Earth candidates')
ax[0].set_ylabel('Number of planets')

# Non-EECs
ax[1].hist(f_ocean[~EEC], bins=bins)
ax[1].set_title('non-habitable planets')

plt.subplots_adjust(wspace=0.3)
```





Of course, you may wish to change the `oceans` function later, and it would be tiring to add it to the Generator again each time. Instead, we can point the Generator to a Python file containing your function, which must be saved under `bioverse/functions/`.

```
[10]: # Reload the generator to erase the previously-added step
generator = Generator('transit')

# Write the function definition as a string
func = """
# Define a function that accepts and returns a Table object
def oceans(d, f_ocean_min=0, f_ocean_max=1):
    # First, assign zero for all planets
    d['f_ocean'] = np.zeros(len(d))

    # Next, assign a random non-zero value for exo-Earth candidates
    EEC = d['EEC']
    d['f_ocean'][EEC] = np.random.uniform(f_ocean_min, f_ocean_max, size=EEC.sum())

    # Finally, return the Table with its new column
    return d
"""

# Save the function to a .py file
with open(ROOT_DIR+'example_oceans.py', 'w') as f:
    f.write(func)

# Insert this function into the Generator and specify the filename
generator.insert_step('oceans', filename='example_oceans.py')

# Verify that the new step has been added
generator
```

```
[10]: Generator with 13 steps:
      0: Function 'create_stars_Gaia' with 6 keyword arguments.
      1: Function 'create_planets_SAG13' with 9 keyword arguments.
      2: Function 'assign_orbital_elements' with 1 keyword arguments.
      3: Function 'geometric_albedo' with 2 keyword arguments.
      4: Function 'impact_parameter' with 1 keyword arguments.
      5: Function 'assign_mass' with no keyword arguments.
      6: Function 'compute_habitable_zone_boundaries' with no keyword arguments.
      7: Function 'compute_transit_params' with no keyword arguments.
      8: Function 'classify_planets' with no keyword arguments.
      9: Function 'scale_height' with no keyword arguments.
     10: Function 'Example1_water' with 3 keyword arguments.
     11: Function 'Example2_oxygen' with 2 keyword arguments.
     12: Function 'oceans' with 2 keyword arguments.
```

Now, any changes you make to the oceans function under `example_oceans.py` will automatically be applied. Finally, you will want to save this Generator under a new name, so that you don't have to re-add the new step every time you load Bioverse:

```
[11]: # Save the new Generator
      generator.save('transit_oceans')

      # Reload it
      generator = Generator('transit_oceans')
```

The following lines of code will clean up the files created during this exercise:

```
[12]: import os
      trash = [ROOT_DIR+'/Objects/Generators/transit_oceans.pkl', ROOT_DIR+'/example_oceans.py',
      ↪]
      for filename in trash:
          if os.path.exists(filename):
              os.remove(filename)
```

The next example will translate this simulated sample of planetary systems into a dataset from a transit spectroscopy survey.

## 3.9 Tutorial 2: Simulating survey datasets

In this tutorial, we will review how to use the Survey class to simulate an exoplanet survey, as well as how to configure the Survey's parameters.

### 3.9.1 Setup

Let's start by importing the necessary module from Bioverse.

```
[1]: # Import numpy
import numpy as np

# Import the Generator, ImagingSurvey, and TransitSurvey classes
from bioverse.generator import Generator
from bioverse.survey import ImagingSurvey, TransitSurvey

# Import pyplot (for making plots later) and adjust some of its settings
from matplotlib import pyplot as plt
%matplotlib inline
plt.rcParams['font.size'] = 20.
```

### 3.9.2 Loading the Survey object

Bioverse uses ImagingSurvey and TransitSurvey objects to simulate datasets from a direct imaging or transit spectroscopy exoplanet survey. Each has a 'default' option. Let's take a look at the default imaging survey:

```
[2]: # Load the 'default' imaging survey
survey = ImagingSurvey('default')

# Display some key properties
print("Telescope diameter: {:.1f} meters".format(survey.diameter))
print("Inner working angle: {:.1f} lambda/D".format(survey.inner_working_angle))
print("Outer working angle: {:.1f} lambda/D".format(survey.outer_working_angle))
print("Faintest detectable contrast: {:.1E}".format(10**survey.contrast_limit))

Telescope diameter: 15.0 meters
Inner working angle: 3.5 lambda/D
Outer working angle: 64.0 lambda/D
Faintest detectable contrast: 2.5E-11
```

Next, the default transit survey:

```
[3]: # Load the transit survey
survey = TransitSurvey('default')

# Display some key properties
print("Effective telescope diameter: {:.1f} meters".format(survey.diameter))
print("Typical target star temperature: {:.0f} K".format(survey.T_st_ref))
print("Maximum survey lifetime: {:.1f} years".format(survey.t_max/365.25))

Effective telescope diameter: 50.0 meters
Typical target star temperature: 3300 K
Maximum survey lifetime: 10.0 years
```

Each survey is capable of conducting a set of measurements on the planets it observes. Let's take a look at the transit survey:

```
[4]: print(survey)
```

TransitSurvey with the following parameters:

```
label: default
diameter: 50.0
t_max: 3652.5
t_slew: 0.0208
T_st_ref: 3300.0
R_st_ref: 0.315
D_ref: 50.0
d_ref: 50.0
N_obs_max: 1000
mode: transit
```

Conducts the following measurements

- (0) Measures parameter 'L\_st'
- (1) Measures parameter 'R\_st' with 5% precision
- (2) Measures parameter 'M\_st' with 5% precision
- (3) Measures parameter 'T\_eff\_st' with 25.0 precision
- (4) Measures parameter 'd'
- (5) Measures parameter 'H'
- (6) Measures parameter 'age' with 30% precision
- (7) Measures parameter 'depth'
- (8) Measures parameter 'T\_dur'
- (9) Measures parameter 'P' with 0.001 precision
- (10) Measures parameter 'has\_H2O'  
Average time required: 7.5 d
- (11) Measures parameter 'EEC'
- (12) Measures parameter 'has\_O2'  
Average time required: 3.1 d

Each measurement is conducted in the order it is listed. Some have special properties:

**Uncertainty:** Some values are measured with approximately zero uncertainty (e.g. distance to the star), while others are measured with poorer precision (e.g.  $\pm 30\%$  for the system's age).

**Conditions:** Not every measurement is applied to every planet. For example, only approximately terrestrial-sized planets orbiting between 0.1 to 10 AU (adjusted for stellar luminosity) are probed for the presence of H<sub>2</sub>O.

**Total allocated time:** Some characterizations require a large amount of time to complete. For example, for the typical EEC target, the survey requires  $\sim 3$  days to detect the presence of ozone in the atmosphere.

### 3.9.3 Producing simulated datasets

The first step in a simulated Survey is to determine which simulated planets can actually be detected. Let's start by generating a sample of planets orbiting stars that might be targeted by LUVOIR. Note that the Generator used here does *not* draw stars from a stellar mass function; instead, it inherits the sophisticated coronagraph yield modeling performed by Stark et al. (2019).

```
[5]: # Generate a sample of planetary systems
generator = Generator('imaging')
sample = generator.generate()
```

Next, we use the imaging Survey object to determine which of these planets can be detected by LUVOIR.

```
[6]: survey = ImagingSurvey('default')
detected = survey.compute_yield(sample)

print("Detected {:d} planets including {:d} exo-Earth candidates.".format(len(detected),
    detected['EEC'].sum()))

Detected 360 planets including 25 exo-Earth candidates.
```

Finally, we simulate a dataset from observations of all of the detected planets over the course of 10 years.

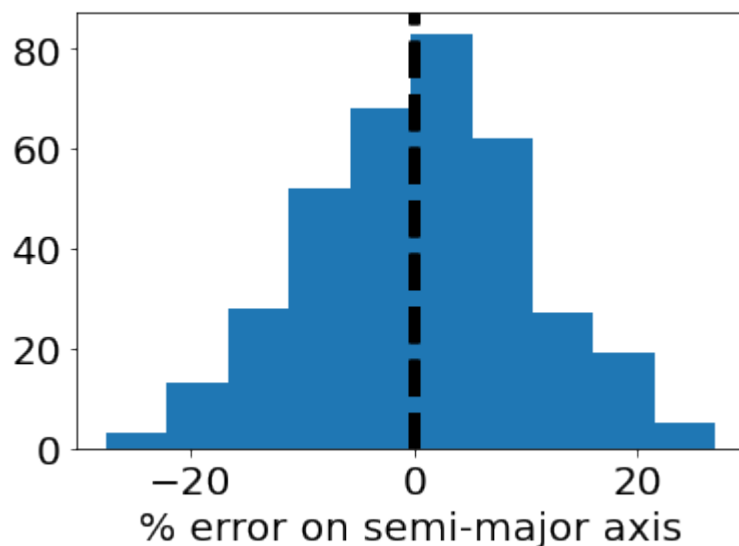
```
[7]: data = survey.observe(detected, t_total=10*365.25)
```

The values in data are imprecise measurements of the true values in detected. For example, let's look at the fractional measurement error on the semi-major axis.

```
[8]: error = (data['a']-detected['a'])/detected['a']
std = np.std(error)

plt.hist(error*100, bins=10)
plt.xlabel('% error on semi-major axis')
plt.axvline(-std, c='black', lw=5, linestyle='dashed')
plt.axvline(std, c='black', lw=5, linestyle='dashed')

[8]: <matplotlib.lines.Line2D at 0x7fa6e9cb3c10>
```



The standard deviation, highlighted by black lines, is about 10%. This is the estimated uncertainty in semi-major axis determination following ~3 direct imaging revisits (Guimond & Cowan 2019).

Let's repeat this exercise for the transit survey. This time we will use the stellar mass function to generate host stars, and the `quickrun()` function to consolidate the steps.

```
[9]: # Load the Generator and Survey
generator = Generator('transit')
survey = TransitSurvey('default')

# Instead of this:
# sample = generator.generate(transit_mode=True)
```

(continues on next page)

(continued from previous page)

```
# detected = survey.compute_yield(generator)
# data = survey.observe(sample, t_total=10*365.25)

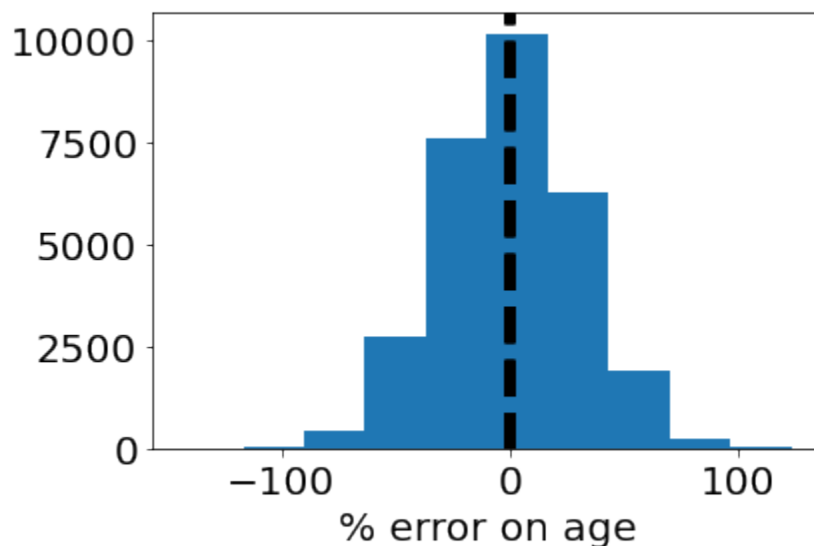
# Use this:
sample, detected, data = survey.quickrun(generator, t_total=10*365.25) # note that
↳transit_mode = True is automatically passed for transit surveys
```

Because the transit survey mostly observes low-mass stars, planet ages are difficult to constrain. The state of the art for low-mass stellar age constraints is around 30% (for TRAPPIST-1; see Burgasser & Mamajek 2017)

```
[10]: error = (data['age']-detected['age'])/detected['age']
std = np.std(error)

plt.hist(error*100, bins=10)
plt.xlabel('% error on age')
plt.axvline(-std, c='black', lw=5, linestyle='dashed')
plt.axvline(std, c='black', lw=5, linestyle='dashed')
```

```
[10]: <matplotlib.lines.Line2D at 0x7fa6e5c0caf0>
```



### 3.9.4 Time constraints

In principle, the transit survey is capable of detecting and characterizing any transiting planet, provided it can observe enough transits to build up the spectroscopic signal-to-noise ratio. In reality, the total number of transits it can observe is limited by survey duration.

For example, a key observable parameter for terrestrial planets is the presence (or absence) of H<sub>2</sub>O in the atmosphere. However, water clouds obscure almost all absorption from water vapor along the same sightlines. Using the Planetary Spectrum Generator with sophisticated GCM models that include the effects of clouds, we have estimated that the transit survey will require approximately ~7.5 days of in-transit observing time for a typical star, assuming a 50-meter effective diameter. We incorporate this assumption into the ‘has\_H<sub>2</sub>O’ measurement:

```
[11]: # Retrieve the time required for the has_H2O measurement
t_ref = survey.measurements['has_H2O'].t_ref
```

(continues on next page)

(continued from previous page)

```
# Properties of the reference star
T_st_ref = survey.T_st_ref
R_st_ref = survey.R_st_ref

print("Reference star properties: R = {:.1f} R_sun and T_eff = {:.0f} K".format(R_st_ref,
↪ T_st_ref))
print("Time required to characterize an Earth-like planet around this star: {:.1f} d".
↪ format(t_ref))
```

```
Reference star properties: R = 0.3 R_sun and T_eff = 3300 K
Time required to characterize an Earth-like planet around this star: 7.5 d
```

`t_ref` is scaled according to planet size, stellar size, stellar distance, atmospheric scale height, etc. to produce an estimate for the observing time required for each individual planet.

Plotted below are the exposure times (and number of transit observations) required to find H<sub>2</sub>O in the atmospheres of transiting EECs. The dashed lines indicate a generous upper limit of 1,000 combined transit observations of ~1 hr duration.

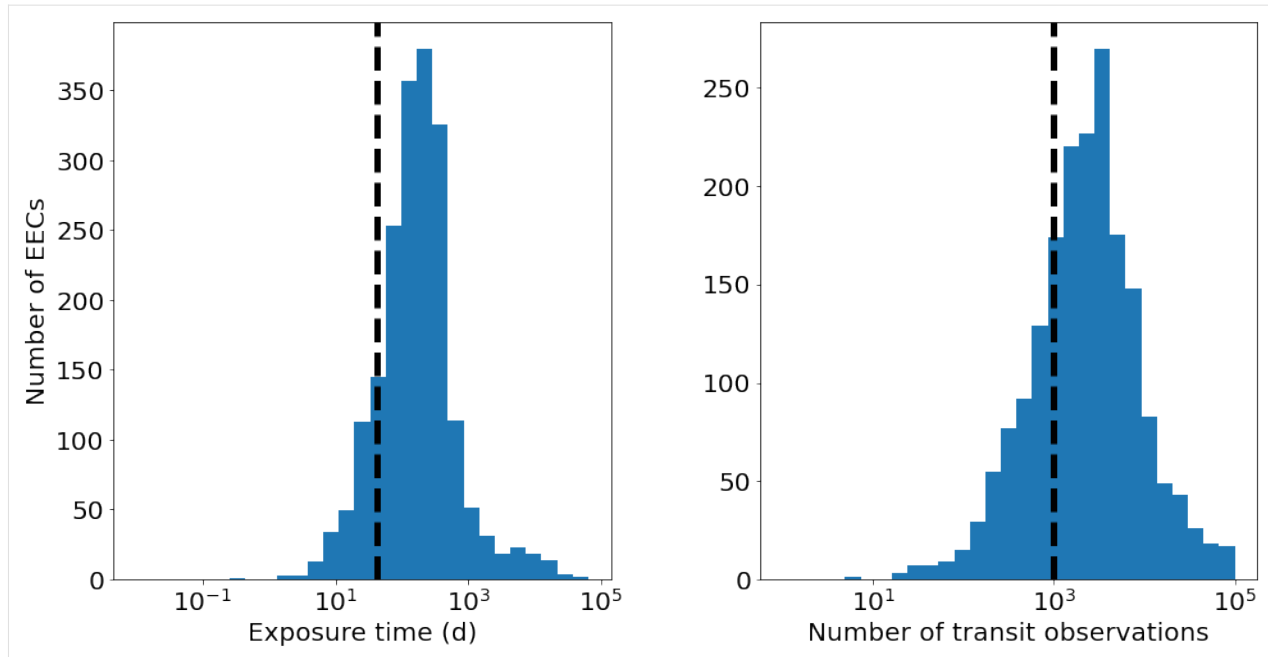
```
[12]: sample, detected, data = survey.quickrun(generator, t_total=10*365.25)
t_exp, N_obs = survey.measurements['has_H2O'].compute_exposure_time(data[detected['EEC
↪ ']])

fig, ax = plt.subplots(ncols=2, figsize=(16,8))

bins = np.logspace(np.log10(0.01), np.log10(np.amax(t_exp)), 30)
ax[0].hist(t_exp, bins=bins)
ax[0].set_xscale('log')
ax[0].set_xlabel('Exposure time (d)')
ax[0].set_ylabel('Number of EECs')
ax[0].axvline(1000/24, linestyle='dashed', lw=5, c='black')

bins = np.logspace(0, 5, 30)
ax[1].hist(N_obs, bins=bins)
ax[1].set_xscale('log')
ax[1].set_xlabel('Number of transit observations')
ax[1].axvline(survey.N_obs_max, linestyle='dashed', lw=5, c='black')

plt.subplots_adjust(wspace=0.3)
```



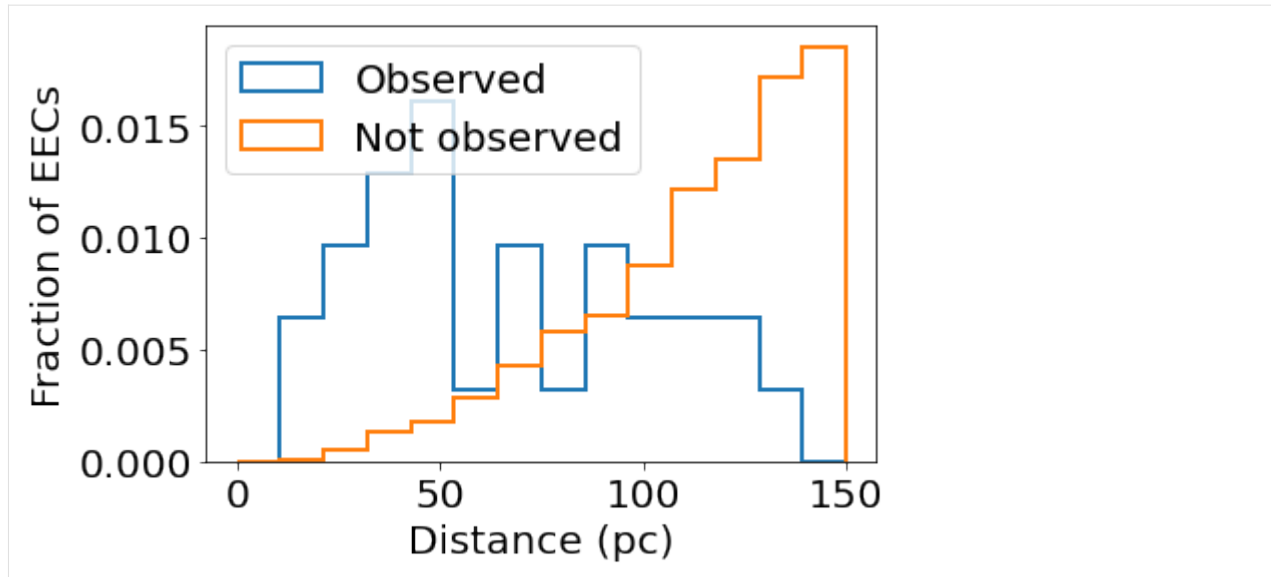
As a result, most exo-Earth candidates cannot be observed during the survey. Since the required exposure time scales with distance<sup>2</sup>, the successfully observed targets tend to be much closer to Earth.

```
[13]: # Determines which EECs were observed vs not observed for atmospheric H2O
obs = ~np.isnan(data['has_H2O'])
EEC = detected['EEC']

bins = np.linspace(0, 150, 15)
plt.hist(data['d'][obs&EEC], density=True, histtype='step', lw=2, bins=bins, label=
    ↳ 'Observed')
plt.hist(data['d'][~obs&EEC], density=True, histtype='step', lw=2, bins=bins, label='Not_
    ↳ observed')
plt.xlabel('Distance (pc)')
plt.ylabel('Fraction of EECs')
plt.legend(loc='upper left')

plt.show()
```



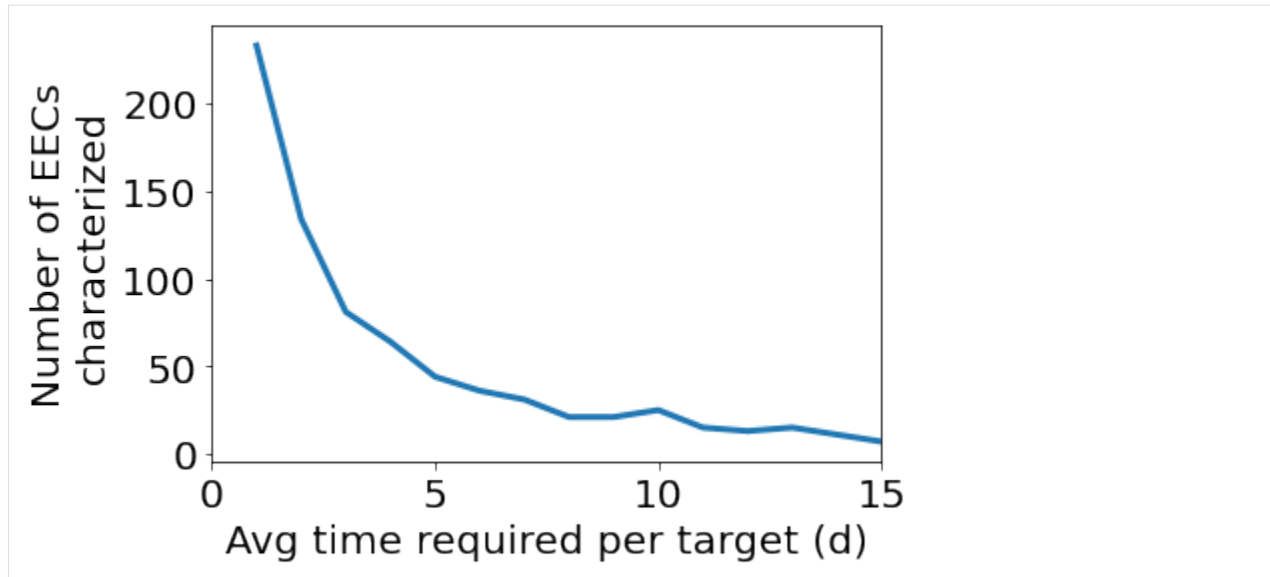


If we choose more optimistic assumptions about cloud cover, we can reduce `t_ref` and therefore increase the number of observable EECs:

```
[14]: t_ref = np.arange(1, 16)
N_EEC = np.zeros(len(t_ref))
for i, tr in enumerate(t_ref):
    survey.measurements['has_H2O'].t_ref = tr
    sample, detected, data = survey.quickrun(generator, t_total=10*365.25)
    EEC = detected['EEC']
    N_EEC[i] = np.sum(~np.isnan(data['has_H2O'][EEC]))

plt.plot(t_ref, N_EEC, lw=3)
plt.xlim([0, 15])
plt.xlabel('Avg time required per target (d)')
plt.ylabel('Number of EECs\nncharacterized')

plt.show()
```



Note that these exposure time considerations apply equally to imaging surveys. However, given low values of  $\eta$  Earth assumed ( $\sim 7.5\%$ ), even a LUVOIR-like survey will likely be volume-limited in the number of EECs it can characterize. For a given observatory configuration, the number of EECs it can spectroscopically characterize is mostly a function of  $\eta$  Earth. For highly time-intensive observations (e.g. rotational mapping), this may not be the case.

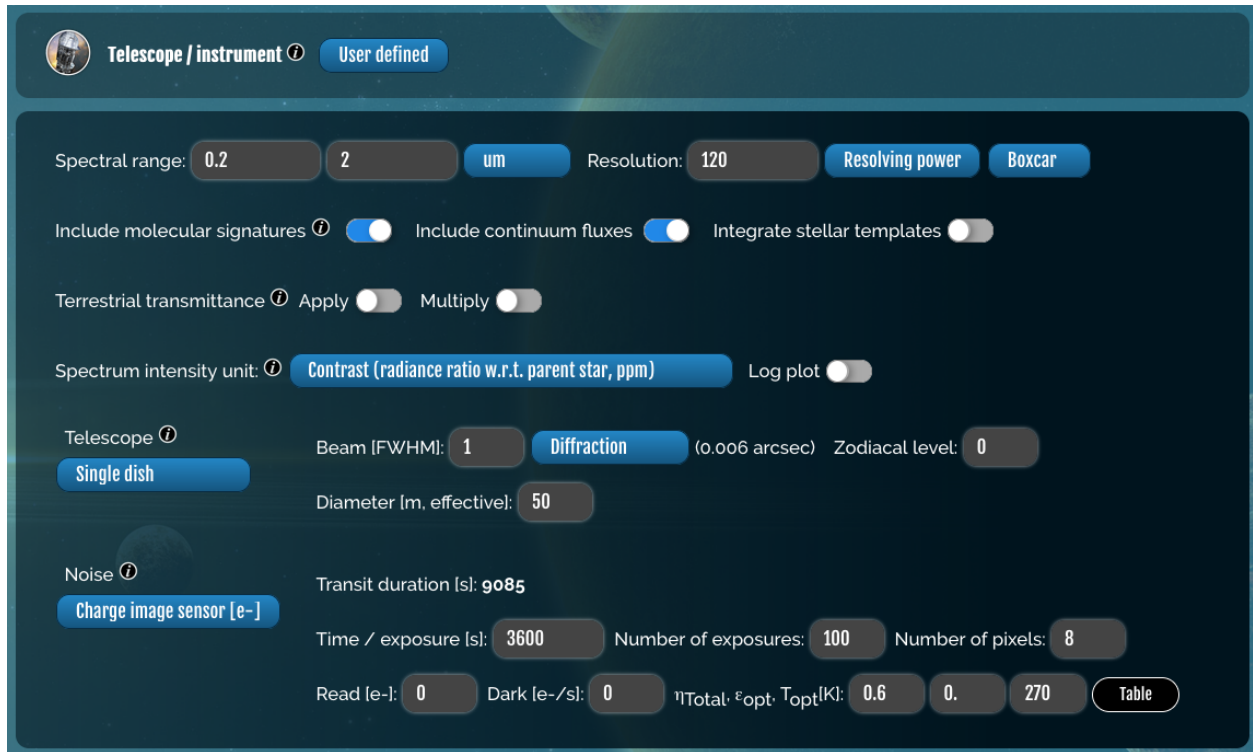
### 3.10 Tutorial 3: Calculating exposure times

The sample size of a simulated survey is often limited by the amount of observing time allotted for time-consuming spectroscopic measurements. To impose this limit, Bioverse requires a realistic estimate for the exposure time needed to conduct the measurement for a typical survey target, referred to as  $t_{\text{ref}}$  (see [Exposure time calculations](#) for details). For the time being,  $t_{\text{ref}}$  must be calculated separately using third-party tools, though future updates may bring this functionality into Bioverse.

This example will demonstrate one method of determining  $t_{\text{ref}}$  for the default transit survey, for which the typical target orbits a mid-M dwarf at  $\sim 50$  parsecs distance. We will use the [Planetary Spectrum Generator \(PSG\)](#) to produce simulated spectra (and noise estimates) for our desired telescope configuration.

### 3.10.1 Telescope and target properties

To begin, navigate to PSG and load the following template containing the telescope and target parameters: `Templates/transit_100hr.cfg`. Next, click ‘Change Instrument’ to navigate to the instrument parameters interface:



**Telescope / instrument** User defined

Spectral range:    Resolution:  Resolving power Boxcar

Include molecular signatures ☒ Include continuum fluxes ☒ Integrate stellar templates ☐

Terrestrial transmittance ☒ Apply ☐ Multiply ☐

Spectrum intensity unit: Contrast (radiance ratio w.r.t. parent star, ppm) Log plot ☐

Telescope Single dish

Beam [FWHM]:  Diffraction (0.006 arcsec) Zodiacal level:

Diameter [m, effective]:

Noise Charge image sensor [e-]

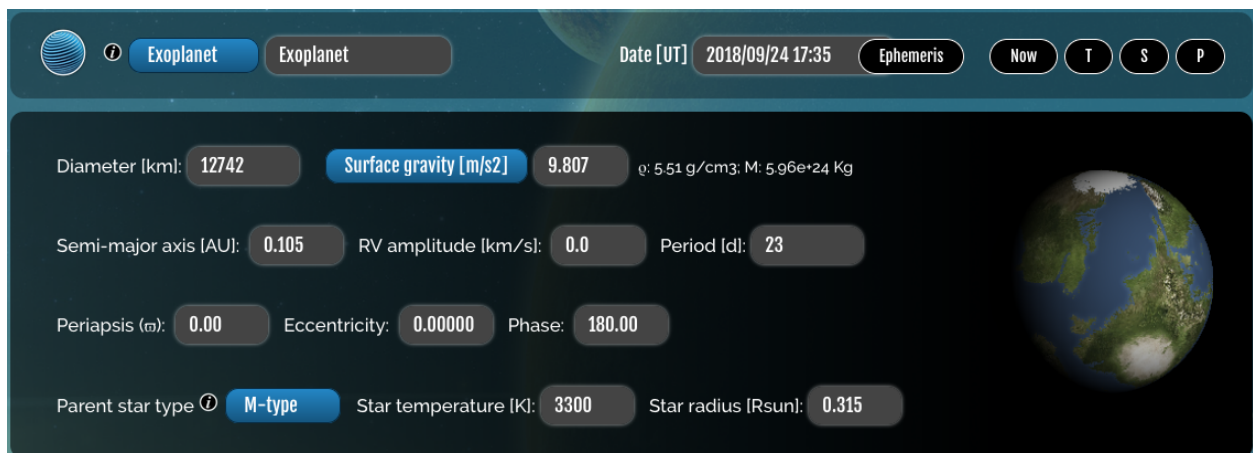
Transit duration [s]:

Time / exposure [s]:  Number of exposures:  Number of pixels:

Read [e-]:  Dark [e-/s]:   $\eta_{\text{Total}} \cdot \epsilon_{\text{opt}} \cdot T_{\text{opt}} [K]$ :    Table

These values reflect the capabilities of the Nautilus Space Observatory, a space telescope array whose total light-collecting area would equal that of a single 50-meter aperture. As a basis for the noise calculation, we have set the exposure time to 100 hr (approximately 50 transit observations) and the total throughput to 60%. Finally, we have configured PSG to produce a spectrum of the transit depth in ppm.

Return to the main PSG interface and click ‘Change Object’ to view the properties of the planet and its star. For the default transit survey, the reference planet is an Earth analog orbiting in the habitable zone of a mid-M dwarf at a distance of 50 parsecs, and is observed at its transit midpoint:



**Exoplanet** Exoplanet Date [UT] 2018/09/24 17:35 Ephemeris Now T S P

Diameter [km]:  Surface gravity [m/s<sup>2</sup>]   $\rho: 5.51 \text{ g/cm}^3; M: 5.96 \times 10^{24} \text{ Kg}$

Semi-major axis [AU]:  RV amplitude [km/s]:  Period [d]:

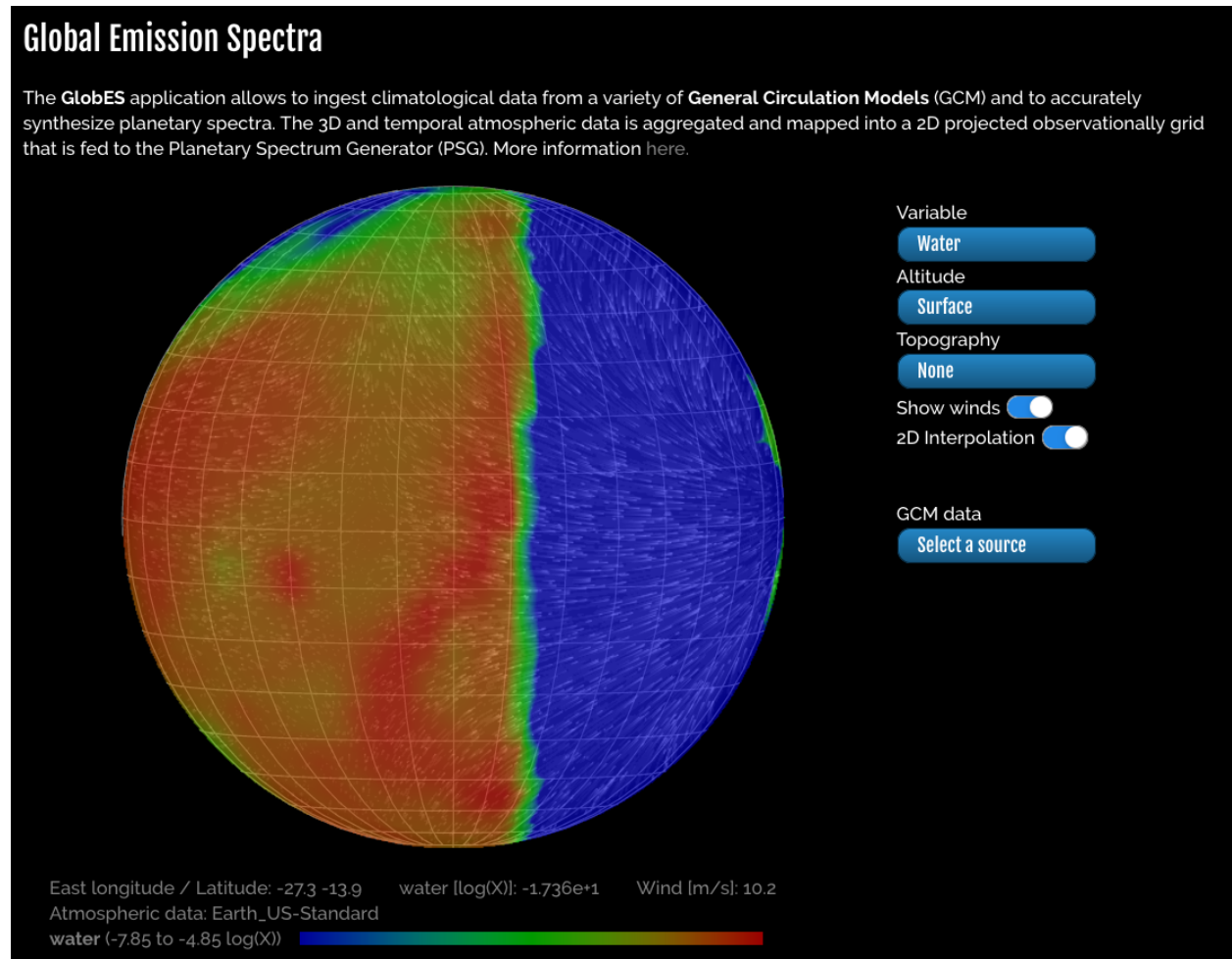
Periastris (m):  Eccentricity:  Phase:

Parent star type M-type Star temperature [K]:  Star radius [Rsun]:

### 3.10.2 3D atmosphere model

By default, the `transit_100hr.cfg` template simulates a 1D model for an Earth-like atmosphere with no clouds or hazes. In reality, clouds are expected to dilute the features of transit spectra of Earth-like planets. To accurately simulate this, we have borrowed one of the 3D GCM models from Komacek & Abbot (2019), which contains realistic 3D abundance profiles of water and water ice clouds (as well as N<sub>2</sub>, H<sub>2</sub>O, and CO<sub>2</sub>) for a tidally-locked world whose bulk properties and host star are similar to those of the above planet. Additionally, we have manually injected Earth-like O<sub>2</sub> and O<sub>3</sub> abundance profiles into the model.

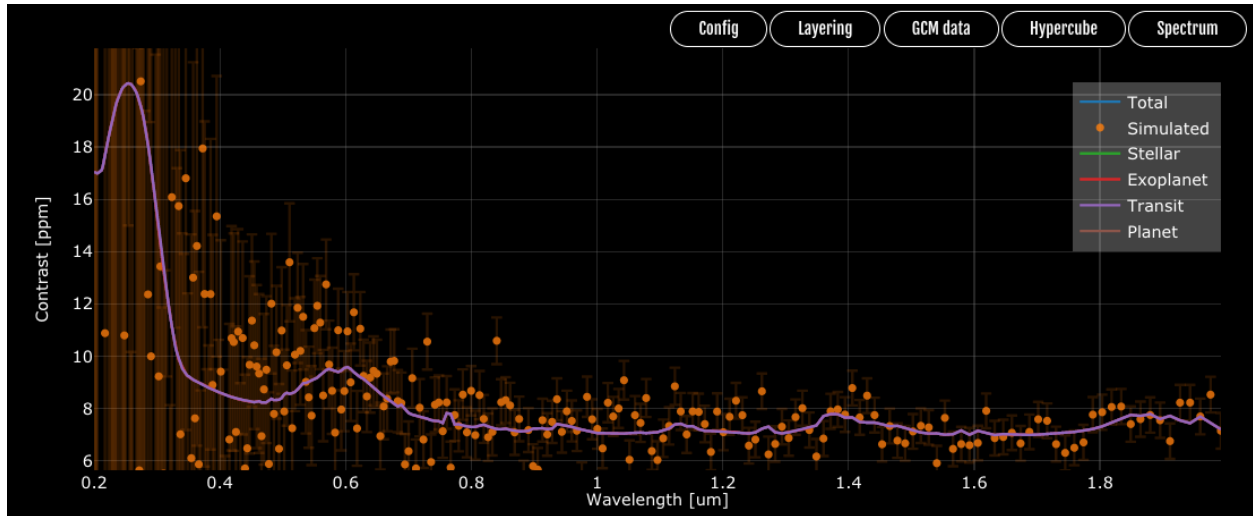
We can produce a simulated spectrum for this 3D model using the **GLOBES module** of PSG. Navigate to the GLOBES module and load the following file containing the GCM data: `Templates/transit.gcm`. Select ‘Water’ under the first drop-down menu, then rotate the globe to view the spatial distribution of water clouds along the terminator:



The strong dayside cloud cover is a result of tidal locking, which promotes more efficient convection. High-altitude clouds near the terminator block sightlines into the lower atmosphere, reducing the amplitude of transit spectroscopy signals from species below the cloud deck.

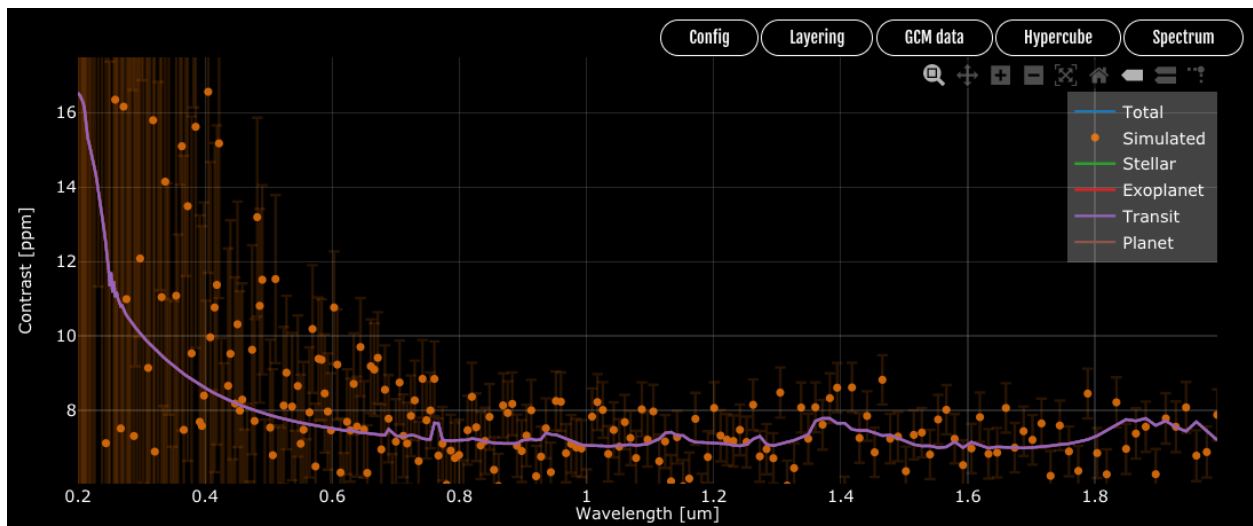
### 3.10.3 Simulated transit spectra

Click 'Generate 3D Spectra' to simulate the transit spectrum for this model using the above telescope configuration:



The near-infrared absorption features of water vapor - typically 2-3 ppm strong - are significantly muted due to clouds. However, the ozone feature around 0.6 microns remains visible, as most of the ozone resides above the cloud deck. To estimate the detectability of this ozone feature, we must compare simulated spectra both with and without O<sub>3</sub>. Begin by clicking 'Spectrum' to download the simulated spectrum and uncertainties (save it as `spectrum_03.dat`). Then, click the button to change the parameters of the atmosphere, and change the ozone abundance to zero:

Save this change at the bottom of the page, return to the GLOBES module, and simulate the spectrum again:



As expected, the ozone feature has vanished. Save this spectrum under a different filename (e.g., `spectrum_no03.dat`).

### 3.10.4 Estimating t\_ref

The final step is to calculate the detection SNR for the simulated 100 hr exposure time and scale that value to determine the requirements for a 5-sigma detection. We can calculate this using `compute_t_ref()`. Assuming the spectra are saved as `spectrum_03.dat` and `spectrum_no03.dat`:

```
from bioverse.util import compute_t_ref

t_ref = compute_t_ref(filenamees=('spectrum_03.dat', 'spectrum_no03.dat'), t_exp=100, wl_
    min=0.4, wl_max=0.8)
print("Required exposure time: {:.1f} hr".format(t_ref))
```

Output: Required exposure time: 73.9 hr

To use this value in a Survey, edit the `t_ref` parameter of the `has_02` Measurement (also specify the effective wavelength of the absorption feature as `wl_eff`. These values should be converted into days and microns, respectively:

```
from bioverse.survey import TransitSurvey

survey = TransitSurvey('default')
survey.measurements['has_02'].t_ref = t_ref / 24.
survey.measurements['has_02'].wl_eff = 0.6
survey.save()
```

Bioverse will now scale this value to determine the exposure time required to detect (or reject) ozone for each individual planet, and prioritize planets appropriately.

## 3.11 Example 1: Finding the habitable zone

In Section 6 of [Bixel & Apai \(2021\)](#), we propose that the concept of a “habitable zone” could be validated by searching for a region of space where planets with atmospheric water vapor are statistically more frequent.

In this example, you will use Bioverse to determine whether a LUVOIR-like imaging survey could test this hypothesis.

### 3.11.1 Setup

First, we’ll import the Bioverse code:

```
[1]: # Import numpy
import numpy as np

# Import the relevant modules
from bioverse.survey import ImagingSurvey
from bioverse.generator import Generator
from bioverse.hypothesis import Hypothesis
from bioverse import analysis

# Import pyplot (for making plots later) and adjust some of its settings
from matplotlib import pyplot as plt
%matplotlib inline
plt.rcParams['font.size'] = 20.
```

For this example, we will use the LUVOIR-like imaging survey and host star catalog.

```
[2]: generator = Generator('imaging')
survey = ImagingSurvey('default')
```

### 3.11.2 Injecting the statistical effect

The first step is to inject the statistical effect we are searching for into the simulated planet population. Specifically, we will simulate the likelihood that a planet has atmospheric water vapor as follows:

```
[3]: def habitable_zone_water(d, f_water_habitable=0.75, f_water_nonhabitable=0.01):
    d['has_H2O'] = np.zeros(len(d), dtype=bool)

    # Non-habitable planets with atmospheres
    m1 = d['R'] > 0.8*d['S']**0.25
    d['has_H2O'][m1] = np.random.uniform(0,1,size=m1.sum()) < f_water_nonhabitable

    # exo-Earth candidates
    m2 = d['EEC']
    d['has_H2O'][m2] = np.random.uniform(0,1,size=m2.sum()) < f_water_habitable

    return d

generator.insert_step(habitable_zone_water)
```

Next, let's make a simulated dataset using this modified Generator object and the imaging Survey. Let's start with a relatively optimistic assumption that 75% of EECs are habitable and only 1% of non-EECs have “false positive” water vapor.

```
[4]: sample, detected, data = survey.quickrun(generator, f_water_habitable=0.75, f_water_
↪nonhabitable=0.01)
```

Let's take a look at the simulated data set by plotting which planets have H<sub>2</sub>O versus their insolation. You might notice that planets within the habitable zone (approx  $0.3 < S < 1.1$ ) are more likely to have water-rich atmospheres.

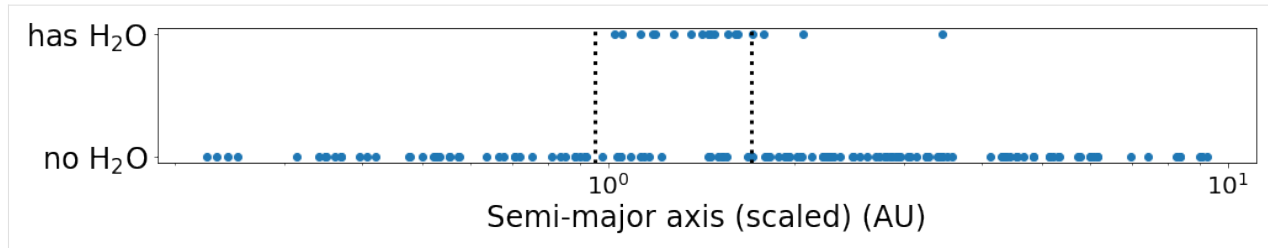
```
[5]: x, y = data['a_eff'], data['has_H2O']

# Now plot the water-rich/water-poor planets versus a_eff (in log-space)
fig, ax = plt.subplots(figsize=(16,2))
ax.scatter(x,y)
ax.set_xscale('log')
ax.set_yticks([0,1])
ax.set_yticklabels(['no H$_{2}$O', 'has H$_{2}$O'], fontsize=24)
ax.set_xlabel('Semi-major axis (scaled) (AU)', fontsize=24)

# Highlight the boundaries of the habitable zone (water-rich planets should be more_
↪common here)
ax.axvline(0.95, linestyle='dotted', c='black', lw=3)
ax.axvline(1.7, linestyle='dotted', c='black', lw=3)

plt.show()
```





The effect injected by `habitable_zone_water` is apparent in the dataset. But is this effect of high enough statistical significance to confirm the habitable zone hypothesis?

### 3.11.3 Defining the hypothesis

Next, we'll create a new Hypothesis object representing the hypothesis that planets within one region of space are more likely to have atmospheric water vapor than those outside of it. Since we do not know the planets' sizes, the only independent variable is the distance from the star modulated by the stellar luminosity, called `a_eff` (this is already calculated). Likewise, the dependent variable is the presence or absence of water vapor, called `has_H2O`, which is either 0 or 1. The relationship between these values is parameterized by `a_inner` (the inner edge of the habitable zone in AU), `delta_a` (the width in AU), `f_HZ` (the fraction of HZ planets with H2O), and `df_notHZ` (the fraction of non-HZ planets with H2O *divided by* `f_HZ`). By defining the four parameters in this way, we can easily avoid parameter combinations inconsistent with our hypothesis (such as the fraction of non-HZ planets with H2O being higher than `f_HZ` planets).

```
[6]: # Define the hypothesis in functional form
def f(theta, X):
    a_inner, delta_a, f_HZ, df_notHZ = theta
    in_HZ = (X > a_inner) & (X < (a_inner + delta_a))
    return in_HZ * f_HZ + (~in_HZ) * f_HZ*df_notHZ

# Specify the names of the parameters (theta), features (X), and labels (Y)
params = ('a_inner', 'delta_a', 'f_HZ', 'df_notHZ')
features = ('a_eff',)
labels = ('has_H2O',)
```

In addition, we must consider the prior probability distribution of these parameters. Conservatively, we suppose the inner edge might extend far inward, or be slightly farther from the Sun than Earth, and that the HZ could be very narrow or very wide. Similarly, the fraction of planets with water vapor in the HZ and outside of it could span many orders of magnitude. We therefore choose to impose log-uniform prior distributions on these parameters across the following ranges:

$0.1 < a_{\text{inner}} < 2 \text{ AU}$

$0.01 < \delta a < 10 \text{ AU}$

$0 < f_{\text{HZ}} < 1$

$0 < f_{\text{notHZ}} < 1$

After deciding on the bounds, we can initialize the Hypothesis object.

```
[7]: bounds = np.array([[0.1, 2], [0.01, 10], [0.001, 1.0], [0.001, 1.0]])
h_HZ = Hypothesis(f, bounds, params=params, features=features, labels=labels, log=(True,
↪ True, True, True))
```



We also need to define the null hypothesis against which `h_HZ` is to be compared. The null hypothesis says that the fraction of planets with water vapor is independent of their orbits - this is a one parameter hypothesis where `f_H2O`, the fraction of planets with water vapor, is the only parameter. Again, the prior distribution on `f_H2O` should be broad (0 to 1) and log-uniform in shape. We can define this null hypothesis and attach it to `h_HZ` as follows:

```
[8]: def f_null(theta, X):
      shape = (np.shape(X)[0], 1)
      return np.full(shape, theta)
      bounds_null = np.array([[0.001, 1.0]])
      h_HZ.h_null = Hypothesis(f_null, bounds_null, params=('f_H2O',), features=features,
      ↪ labels=labels, log=(True,))
```

Now that the Hypothesis has been formed, we can formally test it using our simulated dataset. The `fit()` method will automatically extract the appropriate variables from the data and estimate the Bayesian evidence for both the hypothesis and null hypothesis using nested sampling. It will return the difference between the two i.e. the evidence in favor of the habitable zone hypothesis.

```
[9]: results = h_HZ.fit(data)
      print("The evidence in favor of the hypothesis is: dlnZ = {:.1f} (corresponds to p = {:.
      ↪ 1E})".format(results['dlnZ'], np.exp(-results['dlnZ'])))

The evidence in favor of the hypothesis is: dlnZ = 13.1 (corresponds to p = 2.1E-06)
```

Generally speaking, a result where `dlnZ > 3` is considered significant, so this was a successful test of the hypothesis.

**Note for iPython/Jupyter users:** We will need to reload the `generator` and `h_HZ` objects here to replace the ones you have defined above. These will produce the same results, but the next code block uses the `multiprocessing` module which is incompatible with functions defined in iPython.

```
[10]: # Reload `generator` and `h_HZ`
      generator = Generator('imaging')
      from bioverse.hypothesis import h_HZ
```

### 3.11.4 Computing statistical power

We have not yet tackled the most uncertain part of this analysis: namely, just how common are habitable worlds? The fraction of Earth-sized planets in the habitable zone with atmospheric water vapor could far smaller than the assumed 75%, in which case it might be impossible to test the habitable zone hypothesis. To quantify the importance of this assumption, we will need to repeat the previous analysis several times with different values of `f_water_habitable`.

The analysis module enables this through its `test_hypothesis_grid()` function, which loops the planet simulation, survey simulation, and hypothesis test routines over a grid of input values. Let's use it to iterate over values of `f_water_habitable` ranging from 1% to 100%. For each value, we will repeat the analysis 20 times to average over Poisson noise.

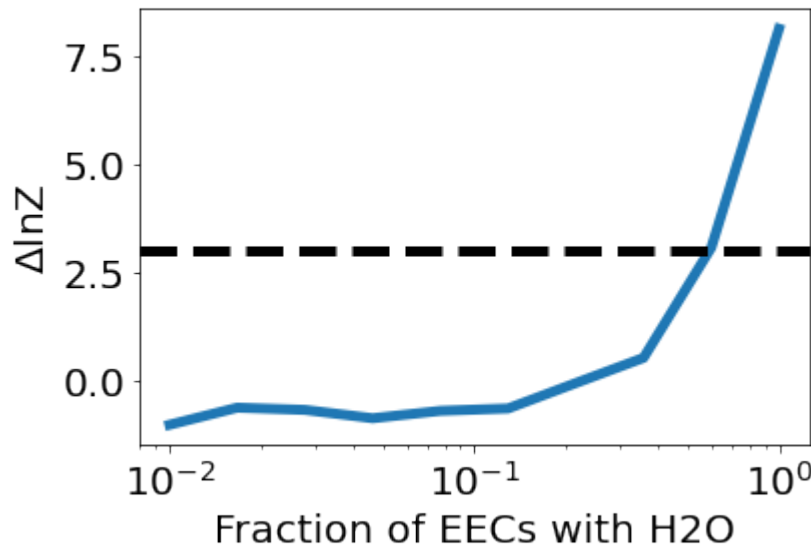
This may take a few minutes. To speed things up, we will run 8 processes in parallel (you may need to change this number for an older CPU).

```
[11]: f_water_habitable = np.logspace(-2, 0, 10)
      results = analysis.test_hypothesis_grid(h_HZ, generator, survey, f_water_habitable=f_
      ↪ water_habitable, t_total=10*365.25, processes=8, N=20)

100%| 200/200 [01:53<00:00, 1.76it/s]
```

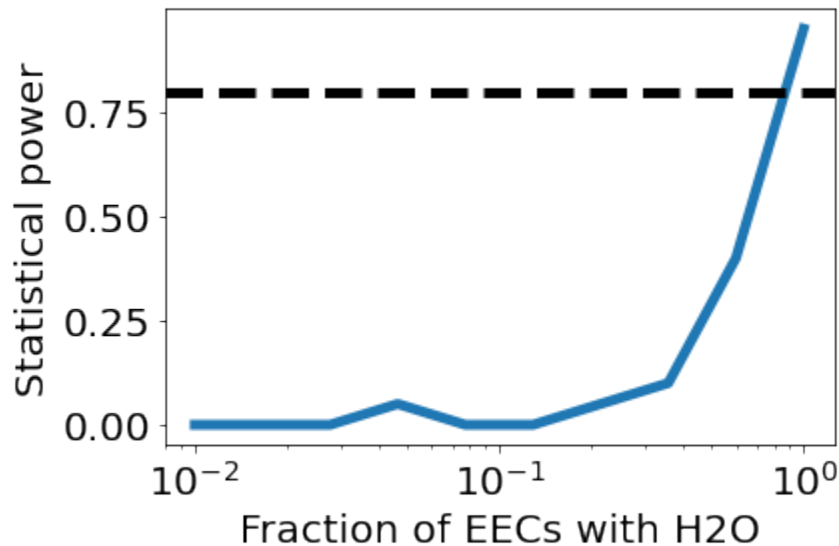
Now, let's plot the average Bayesian evidence for the hypothesis as a function of `f_water_habitable`.

```
[12]: plt.plot(f_water_habitable, results['dlnZ'].mean(axis=-1), lw=5)
plt.xlabel('Fraction of EECs with H2O', fontsize=20)
plt.ylabel('$\Delta\ln Z$', fontsize=20)
plt.axhline(3, lw=5, c='black', linestyle='dashed')
plt.xscale('log')
```



A more useful metric than the average Bayesian evidence is the survey’s “statistical power”, which is the likelihood that the survey could successfully test the hypothesis under a certain set of assumptions. Run this cell to plot the statistical power versus the fraction of EECs with water vapor.

```
[13]: power = analysis.compute_statistical_power(results, method='dlnZ', threshold=3)
plt.plot(f_water_habitable, power, lw=5)
plt.xlabel('Fraction of EECs with H2O', fontsize=20)
plt.ylabel('Statistical power', fontsize=20)
plt.axhline(0.8, lw=5, c='black', linestyle='dashed')
plt.xscale('log')
```



We can see that a LUVOIR-like survey will be able to determine the existence of the habitable zone - but only if habitable planets are relatively common.

## 3.12 Example 2: Detecting the age-oxygen correlation

In Section 7 of [Bixel & Apai \(2021\)](#) (and in [Bixel & Apai 2020](#)), we propose that Earth-like planets might have similar atmospheric evolution to Earth's, i.e. toward greater biogenic oxygen content over Gyr timescales. If so, this would imply a positive “age-oxygen correlation” between the fraction of Earth-like planets with atmospheric oxygen and their ages.

In this example, you will use Bioverse to determine whether a [Nautilus](#)-like transit spectroscopy survey could test this hypothesis.

### 3.12.1 Setup

We will begin by importing modules from Bioverse

```
[1]: # Import numpy
import numpy as np

# Import the relevant modules
from bioverse.survey import TransitSurvey
from bioverse.generator import Generator
from bioverse.hypothesis import Hypothesis
from bioverse import analysis, plots

# Import pyplot (for making plots later) and adjust some of its settings
from matplotlib import pyplot as plt
%matplotlib inline
plt.rcParams['font.size'] = 20.
```

In this example, we will use the simulator for a transit spectroscopy survey with an effecting diameter of 50 meters.

```
[2]: generator = Generator('transit')
survey = TransitSurvey('default')
```

### 3.12.2 Injecting the statistical effect

We propose that the fraction of Earth-like planets with oxygen-rich atmospheres should increase over Gyr timescales. We can simulate this in Bioverse using a function with two parameters:

- `f_life`: the fraction of exo-Earth candidates with life
- `t_half`: the “half-life” timescale over which inhabited planets become oxygenated

Let's define a function and append it to the generator.

```
[3]: def oxygen_evolution(d, f_life=0.8, t_half=3.):
    # First, assign no O2 to all planets
    d['has_O2'] = np.zeros(len(d))
```

(continues on next page)

(continued from previous page)

```
# Calculate the probability that each EEC has O2 based on its age
EEC = d['EEC']
P = f_life * (1 - 0.5**(d['age'][EEC]/t_half))

# Randomly assign O2 to some EECs
d['has_O2'][EEC] = np.random.uniform(0, 1, EEC.sum()) < P

return d

generator.insert_step(oxygen_evolution)
```

Next, we can simulate a dataset and investigate the relationship between the ages and oxygen content of EECs. For now, we assume `f_life = 80%` and `t_half = 3 Gyr`.

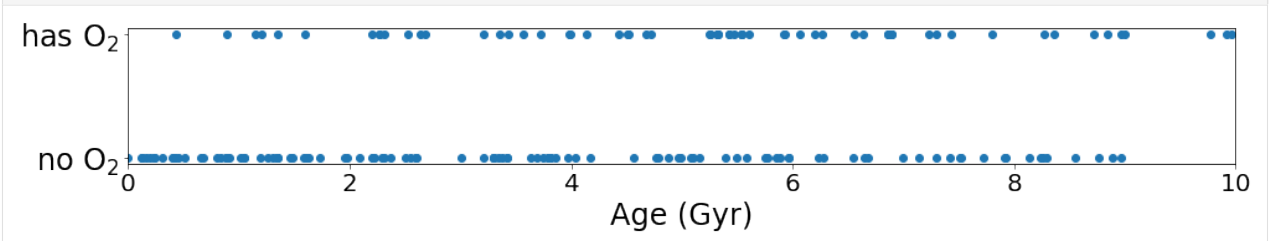
Note that the transit survey is more readily capable of detecting ozone than O<sub>2</sub>, as it has broader and deeper absorption bands and mostly resides above the cloud deck. We assume O<sub>3</sub> is a reliable proxy for O<sub>2</sub>.

```
[4]: # Create the planet sample
sample, detected, data = survey.quickrun(generator, f_life=0.8, t_half=3., t_
    total=10*365.25)

# Extract measured ages and the presence of oxygen
x, y = data['age'], data['has_O2']

# Now plot the oxygen-rich/oxygen-poor planets versus age
fig, ax = plt.subplots(figsize=(16, 2))
ax.scatter(x, y)
ax.set_yticks([0, 1])
ax.set_xlim([0,10])
ax.set_yticklabels(['no O2', 'has O2'], fontsize=24)
ax.set_xlabel('Age (Gyr)', fontsize=24)

plt.show()
```



As in Example 1, the expected trend appears to be present i.e. the presence of oxygen (via its proxy ozone) appears to correlate with age. But is this trend statistically significant?

### 3.12.3 Defining the hypothesis

To answer this, we will define a new Hypothesis. The functional form of this hypothesis will be the same as above i.e. a two-parameter decay rate function. As in the previous example, we must specify the names of the hypothesis parameters, independent variable (age), and dependent variable (has\_O2) in the function signature. We again select log-uniform prior distributions for the two parameters:

0.01 < f\_life < 1  
0.3 < t\_half < 30 Gyr

```
[5]: def f(theta, X):
      f_life, t_half = theta
      return f_life * (1-0.5**(X/t_half))

      params = ('f_life', 't_half')
      features = ('age',)
      labels = ('has_O2',)

      bounds = np.array([[0.01, 1], [0.3, 30]])
      h_age_oxygen = Hypothesis(f, bounds, params=params, features=features, labels=labels,
      ↪ log=(True, True))
```

We also need to define the null hypothesis, which states that the fraction of planets with O2 is (log-uniform) random between 0.001 to 1 and independent of age:

```
[6]: def f_null(theta, X):
      shape = (np.shape(X)[0], 1)
      return np.full(shape, theta)

      bounds_null = np.array([[0.001, 1.]])
      h_age_oxygen.h_null = Hypothesis(f_null, bounds_null, params=('f_O2',),
      ↪ features=features, labels=labels, log=(True,))
```

We can calculate the Bayesian evidence supporting h\_age\_oxygen in favor of h\_null from our simulated dataset.

```
[7]: results = h_age_oxygen.fit(data)
      print("The evidence in favor of the hypothesis is: dlnZ = {:.1f} (corresponds to p = {:.
      ↪ 1E})".format(results['dlnZ'], np.exp(-results['dlnZ'])))

      The evidence in favor of the hypothesis is: dlnZ = 9.9 (corresponds to p = 4.9E-05)
```

The default nested sampling test reveals some compelling evidence in favor of the hypothesis, but it may not be the most efficient test for this problem.

We can also use the Mann-Whitney U test to determine whether the typical age of planets with oxygen-rich atmospheres is higher than that of oxygen-poor planets. This test should be more sensitive, as it makes no specific assumptions about the functional form of the correlation (i.e. func defined above is not used).

```
[8]: results = h_age_oxygen.fit(data, method='mannwhitney')
      print('Correlation detected with p = {:.1E} significance'.format(results['p']))

      Correlation detected with p = 1.2E-05 significance
```

Typically, the Mann-Whitney test detects the correlation with even greater significance (i.e. lower p-values)

**Note for iPython/Jupyter users:** We will need to reload the generator and h\_age\_oxygen objects here to replace the ones you have defined above. These will produce the same results, but the next code block uses the multiprocessing

module which is incompatible with functions defined in iPython.

```
[9]: generator = Generator('transit')
    from bioverse.hypothesis import h_age_oxygen
```

### 3.12.4 Computing statistical power

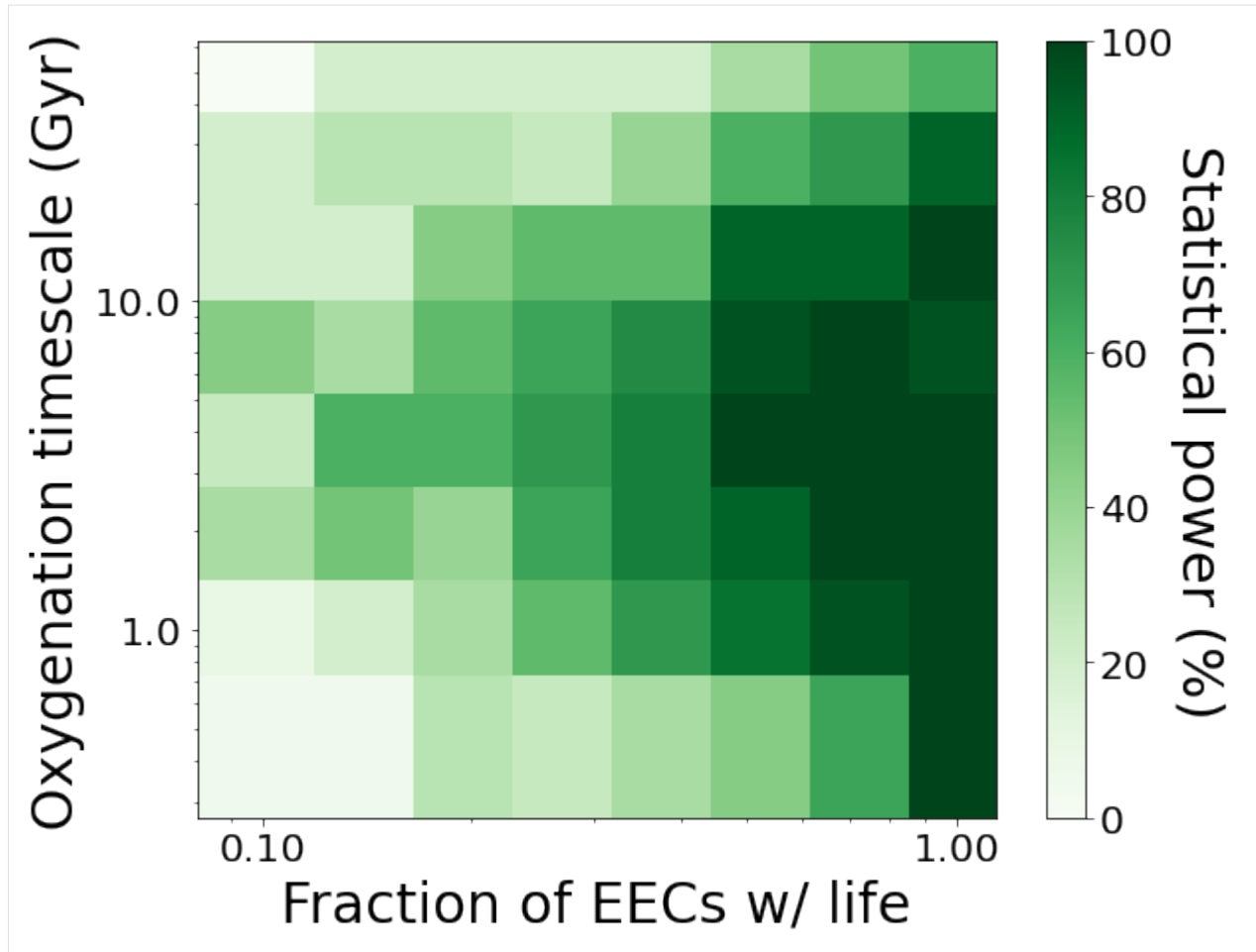
Thus far, we have assumed the fraction of planets with life to be  $f_{\text{life}} = 80\%$  and  $t_{\text{half}} = 3$  Gyr. If fewer planets are inhabited, or if the evolutionary timescale is much longer than the maximum ages probed ( $\sim 10$  Gyr), then the correlation will be more difficult to detect. Let's test the sensitivity of our results to both of these parameters simultaneously using the `test_hypothesis_grid` function.

Once again, we will use the more sensitive Mann-Whitney test to determine whether the correlation exists. This may take up to ten minutes - reduce `N_grid` for quicker results.

```
[10]: N_grid = 8
    f_life = np.logspace(-1, 0, N_grid)
    t_half = np.logspace(np.log10(0.5), np.log10(50), N_grid)
    results = analysis.test_hypothesis_grid(h_age_oxygen, generator, survey, method=
    → 'mannwhitney', f_life=f_life, t_half=t_half, N=20, processes=8, t_total=10*365.25)
    100%| 1280/1280 [10:18<00:00, 2.07it/s]
```

Now, we will use the `plot_power_grid()` function of the `plots` module to plot the statistical power of the survey (i.e. the fraction of tests achieving  $p < 0.05$  significance) versus both parameters.

```
[11]: plots.plot_power_grid(results, method='p', axes=('f_life', 't_half'), labels=('Fraction_
    → of EECs w/ life', 'Oxygenation timescale (Gyr)'), log=(True, True), levels=None)
```



The “sweet spot” where high statistical power is achieved is where life is common ( $f_{\text{life}} > 50\%$ ) and the oxygenation timescale falls within  $\sim 2\text{--}10$  Gyr.

We can also investigate the survey’s sensitivity as a function of total survey time. Let’s try values from 0.1 to 10 years, assuming  $f_{\text{life}} = 50\%$  and  $t_{\text{half}} = 3$  Gyr. This will take a few minutes.

```
[12]: t_total = np.logspace(-1, 1, 10) * 365.25
results = analysis.test_hypothesis_grid(h_age_oxygen, generator, survey, method=
    ↪ 'mannwhitney', f_life=0.5, N=20, processes=8, t_total=t_total)
```

```
100%| 200/200 [01:52<00:00, 1.77it/s]
```

```
[13]: fig, ax = plt.subplots(1, 2, figsize=(12, 4))

ax[0].plot(t_total, results['p'].mean(axis=-1), lw=5)
ax[0].set_xlabel('Total time (d)', fontsize=20)
ax[0].set_ylabel('p', fontsize=20)
ax[0].axhline(0.05, lw=5, c='black', linestyle='dashed')
ax[0].set_xscale('log')

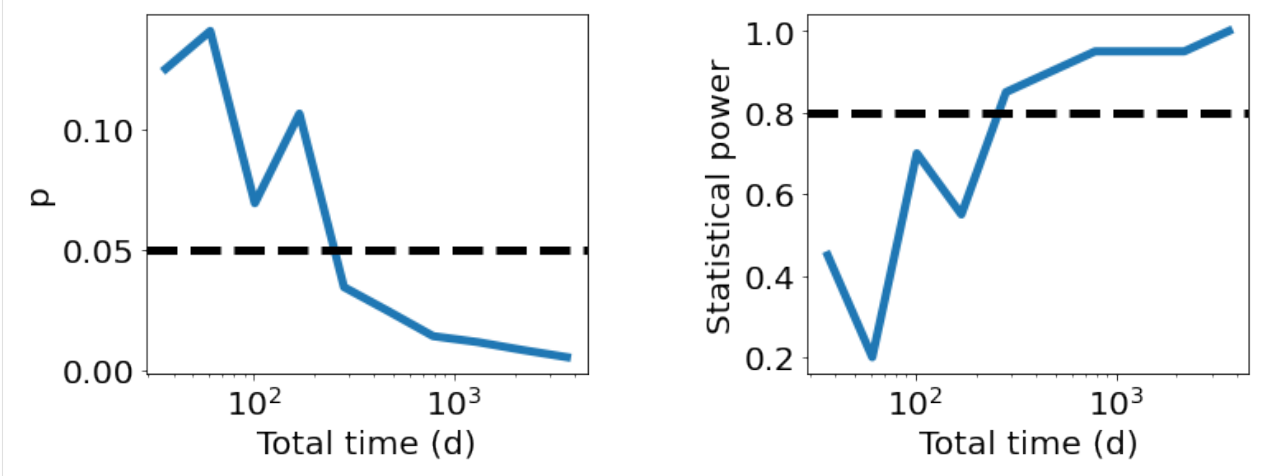
power = analysis.compute_statistical_power(results, method='p', threshold=0.05)
ax[1].plot(t_total, power, lw=5)
ax[1].set_xlabel('Total time (d)', fontsize=20)
```

(continues on next page)

(continued from previous page)

```
ax[1].set_ylabel('Statistical power', fontsize=20)
ax[1].axhline(0.8, lw=5, c='black', linestyle='dashed')
ax[1].set_xscale('log')

plt.subplots_adjust(wspace=0.5)
```



Approximately 1-2 years will be required to detect the correlation under these assumptions.

### 3.13 bioverse.analysis module

Provides functions for iterating over several simulations to compute statistical power and more.

`bioverse.analysis.test_hypothesis_grid(h, generator, survey, N=10, processes=1, do_bar=True, bins=15, return_chains=False, mw_alternative='greater', method='dynesty', nlive=100, **kwargs)`

Runs simulated surveys over a grid of survey and astrophysical parameters. Each time, uses the simulated data set to fit the hypothesis parameters and computes the model evidence versus the null hypothesis.

`bioverse.analysis.test_hypothesis_grid_iter(h, generator, survey, bins, return_chains, mw_alternative, method, seed, nlive, kwargs)`

Runs a single iteration for `test_hypothesis_grid` (separated for multiprocessing).

`bioverse.analysis.compute_statistical_power(results, threshold=None, method='dlnZ')`

Computes the statistical power of a hypothesis test, i.e. the fraction of simulated tests which pass a model comparison significance threshold.

#### Parameters

- **results** (*dict*) – Output of `test_hypothesis_grid`.
- **threshold** (*float, optional*) – Significance threshold to enforce.
- **method** (*('dAIC', 'dBIC', 'dlnZ', 'p', 'logp')*) – Specifies which method to use.

#### Returns

**power** – Array of statistical power for each test in the results grid. Shape is `shape(results[method])[:-1]`.

#### Return type

float array



```

bioverse.analysis.random_simulation(results, generator, survey, bins=15, mw_test=False,
                                   mw_alternative='greater', method='dynesty', nlive=100,
                                   return_chains=True, **grid_kwargs)

bioverse.analysis.compare_methods(h, data, methods=['dynesty', 'emcee'], **kwargs)

bioverse.analysis.number_vs_time(h, generator, survey, t_total, N=30, average=True, **kwargs)
    Determines how many planets are characterized by the simulated survey versus time budget.

bioverse.analysis.number_vs_eta(h, generator, survey, eta_Earth, N=30, average=True, **kwargs)
    Determines how many planets are characterized by the simulated survey versus eta Earth.

bioverse.analysis.number_vs_distance(h, generator, survey, d_max, N=30, average=True, **kwargs)
    Determines how many planets are characterized by the simulated survey versus d_max.

```

## 3.14 bioverse.classes module

Contains class definitions.

**class** bioverse.classes.Object(*label=None*)

Bases: object

This class allows the Generator and Survey classes to be saved as .pkl files under the Objects/ directory.

### Parameters

**label** (*str*, *optional*) – Name of the Generator or Survey. Default is to create a new object.

**save**(*label=None*)

Saves the Object as a template in a .pkl file under ./<object type>s/.

**get\_filename\_from\_label**(*label*)

**class** bioverse.classes.Table(\**args*, \*\**kwargs*)

Bases: dict

Class for storing numpy arrays in a table-like format. Inherits dict. Each key is treated as a separate table column.

**split\_key**(*key*)

Splits a key such as 'Planets:Atmosphere:O2' into ('Planets','Atmosphere:O2') and ensures that the first key refers to a dict-like object.

**keys**()

Returns an array of keys instead of a dict\_keys object, because I prefer it this way.

**sort\_by**(*key*, *inplace=False*, *ascending=True*)

Sorts the table by the values in one column.

### Parameters

- **key** (*str*) – Name of the column by which to sort the table.
- **inplace** (*bool*, *optional*) – If True, sort the table in-place. Otherwise return a new sorted Table.
- **ascending** (*bool*, *optional*) – If True, sort from least to greatest.

### Returns

**sortd** – A sorted copy of this table. Only returned if *inplace* is True.

### Return type

*Table*

#### **get\_stars()**

Returns just the first entry for each star in the Table.

**legend**(*keys=None*,  
*filename='/home/docs/checkouts/readthedocs.org/user\_builds/bioverse/envs/stable/lib/python3.7/site-packages/bioverse-1.1.0-py3.7.egg/bioverse/Data/legend.dat'*)

Prints the description of parameter(s) in the Table.

### Parameters

- **keys** (*str* or *str* array, *optional*) – Key or list of keys to describe. If not specified, every key is described.
- **filename** (*str*, *optional*) – CSV file containing the list of parameter descriptions. Default is *./legend.dat*.

#### **copy()**

Returns a deep copy of the Table instead of a shallow copy (as in *dict.copy()*). This way, if a column is filled by objects (such as Atmosphere objects), a copy of those is returned instead of a reference.

#### **append**(*table*, *inplace=True*)

Appends another table onto this table in-place. The second table must have the same columns, unless this table is empty, in which case the columns are copied over.

### Parameters

- **table** (*Table*) – Table to be appended onto this one.
- **inplace** (*bool*, *optional*) – If True, append inplace and return None. If False, return a new Table.

#### **compute**(*key*, *force=False*)

Computes the value of *key* using other values in the dictionary and a pre-defined formula. Useful for translating measured values (e.g. 'a', 'L\_st') into secondary data products (e.g., 'S'). Will also propagate uncertainties contained in *self.error*.

#### **shuffle**(*inplace=True*)

Re-orders rows in the Table. If *inplace* is False, return a new re-ordered Table instead.

#### **pdshow()**

If pandas is installed, show the Table represented as a DataFrame. Otherwise return an error.

#### **to\_pandas()**

export Table into a pandas DataFrame

#### **observed**(*key*)

Returns the subset of rows for which *self[key]* is not nan.

### **class bioverse.classes.Stopwatch**

Bases: *object*

This class uses the time module to profile chunks of code. Use the *start()* and *stop()* methods to start and stop the Stopwatch, and the *mark()* methods to record a time. *stop()* and *read()* will report the total time elapsed as well as the time between each step.

#### **clear()**

`mark(flag=None)`

`stop(flag=None)`

`read()`

## 3.15 bioverse.constants module

Defines constant values used elsewhere in the code.

## 3.16 bioverse.custom module

Define new functions for planet simulation here. Function arguments should be provided a default value.

## 3.17 bioverse.functions module

Contains all functions currently used to simulate planetary systems. To define new functions, add them to custom.py.

`bioverse.functions.luminosity_evolution(d)`

Computes age-dependent luminosities based on the stellar evolution tracks in Baraffe et al. (1998).

### Parameters

**d** ([Table](#)) – Table with stars. Has to have columns for mass and age.

### Returns

**d**

### Return type

Table containing age-dependent luminosities.

`bioverse.functions.read_stars_Gaia(d, filename='gcns_catalog.dat', d_max=120.0, M_st_min=0.075, M_st_max=2.0, R_st_min=0.095, R_st_max=2.15, T_min=0.0, T_max=10.0, inc_binary=0, seed=42, M_G_max=None, lum_evo=True)`

Reads a list of stellar properties from the Gaia nearby stars catalog.

### Parameters

- **d** ([Table](#)) – An empty Table object.
- **filename** (*str*, *optional*) – Filename containing the Gaia target catalog.
- **d\_max** (*float*, *optional*) – Maximum distance to which to simulate stars, in parsecs.
- **M\_st\_min** (*float*, *optional*) – Minimum stellar mass, in solar units.
- **M\_st\_max** (*float*, *optional*) – Maximum stellar mass, in solar units.
- **R\_st\_min** (*float*, *optional*) – Minimum stellar radius, in solar units.
- **R\_st\_max** (*float*, *optional*) – Maximum stellar radius, in solar units.
- **T\_min** (*float*, *optional*) – Minimum stellar age, in Gyr.
- **T\_max** (*float*, *optional*) – Maximum stellar age, in Gyr.
- **inc\_binary** (*bool*, *optional*) – Include binary stars? Default = False.

- **seed** (*int*, *optional*) – seed for the random number generators.
- **mult** (*float*, *optional*) – Multiple on the total number of stars simulated. If > 1, duplicates some entries from the LUV0IR catalog.
- **M\_G\_max** (*float*, *optional*) – Maximum Gaia magnitude of stars. Example: M\_G\_max=9. keeps all stars brighter than M\_G = 9.0.
- **lum\_evo** (*bool*, *optional*) – Assign age-dependent stellar luminosities (based on randomly assigned ages and stellar luminosity tracks in Baraffe et al. 1998).

#### Returns

**d** – Table containing the sample of real stars.

#### Return type

*Table*

```
bioverse.functions.create_stars_Gaia(d, d_max=150, M_st_min=0.075, M_st_max=2.0, T_min=0.0,
                                     T_max=10.0, T_eff_split=4500.0, seed=42)
```

Reads temperatures and coordinates for high-mass stars from Gaia DR2. Simulates low-mass stars from the Chabrier+2003 PDMF. Ages are drawn from a uniform distribution, by default from 0 - 10 Gyr. All other stellar properties are calculated using the scaling relations of Pecaut+2013.

#### Parameters

- **d** (*Table*) – An empty Table object.
- **d\_max** (*float*, *optional*) – Maximum distance to which to simulate stars, in parsecs.
- **M\_st\_min** (*float*, *optional*) – Minimum stellar mass, in solar units.
- **M\_st\_max** (*float*, *optional*) – Maximum stellar mass, in solar units.
- **T\_min** (*float*, *optional*) – Minimum stellar age, in Gyr.
- **T\_max** (*float*, *optional*) – Maximum stellar age, in Gyr.
- **T\_eff\_split** (*float*, *optional*) – Effective temperature (in Kelvin) below which to simulate stars from a PDMF instead of using Gaia data.
- **seed** (*int* or *1-d array\_like*, *optional*) – Seed for numpy's RandomState. Must be convertible to 32 bit unsigned integers.

#### Returns

**d** – Table containing the sample of simulated stars.

#### Return type

*Table*

```
bioverse.functions.read_stellar_catalog(d,
                                       filename='/home/docs/checkouts/readthedocs.org/user_builds/bioverse/envs/stable/
packages/bioverse-1.1.0-
py3.7.egg/bioverse/Data/LUVOIR_targets.dat', d_max=30.0,
                                       T_min=0.0, T_max=10.0, mult=1, seed=42)
```

Reads a list of stellar properties from the LUV0IR target catalog and fills in missing values.

#### Parameters

- **d** (*Table*) – An empty Table object.
- **filename** (*str*, *optional*) – Filename containing the LUV0IR target catalog.
- **d\_max** (*float*, *optional*) – Maximum distance to which to simulate stars, in parsecs.
- **T\_min** (*float*, *optional*) – Minimum stellar age, in Gyr.

- **T\_max** (*float, optional*) – Maximum stellar age, in Gyr.
- **mult** (*float, optional*) – Multiple on the total number of stars simulated. If > 1, duplicates some entries from the LUVIR catalog.
- **seed** (*int or 1-d array\_like, optional*) – Seed for numpy's RandomState. Must be convertible to 32 bit unsigned integers.

#### Returns

**d** – Table containing the sample of simulated stars.

#### Return type

*Table*

`bioverse.functions.create_planets_bergsten(d, R_min=1.0, R_max=3.5, P_min=2, P_max=100.0, transit_mode=False, f_eta=1.0, seed=42)`

Generates planets with periods and radii according to Bergsten+2022 occurrence rate estimates.

#### Parameters

- **d** (*Table*) – Table containing simulated host stars.
- **R\_min** (*float, optional*) – Minimum planet radius, in Earth units.
- **R\_max** (*float, optional*) – Maximum planet radius, in Earth units.
- **P\_min** (*float, optional*) – Minimum orbital period, in days.
- **P\_max** (*float, optional*) – Maximum orbital period, in days.
- **transit\_mode** (*bool, optional*) – If True, only transiting planets are simulated.
- **f\_eta** (*float, optional*) – Occurrence rate scaling factor. The default `f_eta = 1` represents the occurrence rates in Bergsten+2022. A different factor will scale the overall occurrence rates accordingly.
- **seed** (*int or 1-d array\_like, optional*) – Seed for numpy's RandomState. Must be convertible to 32 bit unsigned integers.

#### Returns

**d** – Table containing the sample of simulated planets. Replaces the input Table.

#### Return type

*Table*

`bioverse.functions.create_planets_SAG13(d, eta_Earth=0.075, R_min=0.5, R_max=14.3, P_min=0.01, P_max=10.0, normalize_SpT=True, transit_mode=False, optimistic=False, optimistic_factor=5, seed=42)`

Generates planets with periods and radii according to SAG13 occurrence rate estimates, but incorporating the dependence of occurrence rates on spectral type from Mulders+2015.

#### Parameters

- **d** (*Table*) – Table containing simulated host stars.
- **eta\_Earth** (*float, optional*) – The number of Earth-sized planets in the habitable zones of Sun-like stars. All occurrence rates are uniformly scaled to produce this estimate.
- **R\_min** (*float, optional*) – Minimum planet radius, in Earth units.
- **R\_max** (*float, optional*) – Maximum planet radius, in Earth units.
- **P\_min** (*float, optional*) – Minimum orbital period, in years.
- **P\_max** (*float, optional*) – Maximum orbital period, in years.

- **normalize\_SpT** (*bool, optional*) – If True, modulate occurrence rates by stellar mass according to Mulders+2015. Otherwise, assume no dependency on stellar mass.
- **transit\_mode** (*bool, optional*) – If True, only transiting planets are simulated. Occurrence rates are modified to reflect the  $R_*/a$  transit probability.
- **optimistic** (*bool, optional*) – If True, extrapolate the results of Mulders+2015 by assuming rocky planets are much more common around late-type M dwarfs. If False, assume that occurrence rates plateau with stellar mass for stars cooler than ~M3.
- **optimistic\_factor** (*float, optional*) – If optimistic = True, defines how many times more common rocky planets are around late-type M dwarfs compared to Sun-like stars.
- **seed** (*int or 1-d array\_like, optional*) – Seed for numpy's RandomState. Must be convertible to 32 bit unsigned integers.

**Returns**

**d** – Table containing the sample of simulated planets. Replaces the input Table.

**Return type**

*Table*

`bioverse.functions.create_planet_per_star(d, R_min=0.5, R_max=14.3)`

Generates a single planet for each star with a radius drawn from a uniform distribution between  $R_{\min}$  and  $R_{\max}$

**Parameters**

- **d** (*Table*) – Table containing simulated host stars.
- **R\_min** (*float, optional*) – Minimum planet radius, in Earth units.
- **R\_max** (*float, optional*) – Maximum planet radius, in Earth units.

**Returns**

**d** – Table containing the sample of simulated planets. Replaces the input Table.

**Return type**

*Table*

`bioverse.functions.name_planets(d)`

Assign a name to each star and each planet based on its order in the system.

**Parameters**

**d** (*Table*) – Table containing the sample of simulated planets.

**Returns**

**d** – Table containing the sample of simulated planets.

**Return type**

*Table*

`bioverse.functions.assign_orbital_elements(d, transit_mode=False, seed=42)`

Draws values for any remaining Keplerian orbital elements. Eccentricities are drawn from a beta distribution following Kipping et al. (2013).

**Parameters**

- **d** (*Table*) – Table containing the sample of simulated planets.
- **transit\_mode** (*bool, optional*) – If True, only transiting planets are simulated, so  $\cos(i) < R_*/a$  for all planets.

- **seed**(*int or 1-d array\_like, optional*) – Seed for numpy’s RandomState. Must be convertible to 32 bit unsigned integers.

**Returns**

**d** – Table containing the sample of simulated planets.

**Return type**

*Table*

`bioverse.functions.impact_parameter(d, transit_mode=False)`

Calculates the impact parameter/transit duration.

**Parameters**

- **d** (*Table*) – Table containing the sample of simulated planets.
- **transit\_mode** (*bool, optional*) – If True, only transiting planets are simulated, so planets with  $b > 1$  are discarded.

**Returns**

**d** – Table containing the sample of simulated planets.

**Return type**

*Table*

`bioverse.functions.assign_mass(d, mr_relation='Wolfgang2016')`

Determines planet masses using a probabilistic mass-radius relationship, following Wolfgang et al. (2016). Also calculates density and surface gravity.

**Parameters**

- **d** (*Table*) – Table containing the sample of simulated planets.
- **mr\_relation** (*str, optional*) – Mass-radius relationship to consider. Must be either ‘Wolfgang2016’ (Wolfgang et al., 2016) or ‘Zeng2016’ (Zeng et al., 2016).

**Returns**

**d** – Table containing the sample of simulated planets.

**Return type**

*Table*

`bioverse.functions.classify_planets(d)`

Classifies planets by size and instellation following Kopparapu et al. (2018).

**Parameters**

**d** (*Table*) – Table containing the sample of simulated planets.

**Returns**

**d** – Table containing the sample of simulated planets.

**Return type**

*Table*

`bioverse.functions.compute_habitable_zone_boundaries(d)`

Computes the habitable zone boundaries from Kopparapu et al. (2014), including dependence on planet mass.

**Parameters**

**d** (*Table*) – Table containing the sample of simulated planets.

**Returns**

**d** – Table containing the sample of simulated planets.

**Return type**

*Table*

`bioverse.functions.scale_height(d)`

Computes the equilibrium temperature and isothermal scale height by assigning a mean molecular weight based on size.

**Parameters**

**d** (*Table*) – Table containing the sample of simulated planets.

**Returns**

**d** – Table containing the sample of simulated planets.

**Return type**

*Table*

`bioverse.functions.geometric_albedo(d, A_g_min=0.1, A_g_max=0.7, seed=42)`

Assigns each planet a random geometric albedo from 0.1 – 0.7, and computes the contrast ratio when viewed at quadrature.

**Parameters**

- **d** (*Table*) – Table containing the sample of simulated planets.
- **A\_g\_min** (*float, optional*) – Minimum geometric albedo.
- **A\_g\_max** (*float, optional*) – Maximum geometric albedo.
- **seed** (*int or 1-d array\_like, optional*) – Seed for numpy’s RandomState. Must be convertible to 32 bit unsigned integers.

**Returns**

**d** – Table containing the sample of simulated planets.

**Return type**

*Table*

`bioverse.functions.effective_values(d)`

Computes the “effective” radius and semi-major axis (i.e. assuming an Earth-like planet).

**Parameters**

**d** (*Table*) – Table containing the sample of simulated planets.

**Returns**

**d** – Table containing the sample of simulated planets.

**Return type**

*Table*

`bioverse.functions.compute_transit_params(d)`

Computes the transit depth of each planet.

**Parameters**

**d** (*Table*) – Table containing the sample of simulated planets.

**Returns**

**d** – Table containing the sample of simulated planets.

**Return type**

*Table*



`bioverse.functions.apply_bias(d, M_min=0.0, M_max=inf, S_min=0.0, S_max=inf, depth_min=0.0)`

Apply detection biases and custom selections to the sample to generate.

#### Parameters

- **d** ([Table](#)) – Table containing the sample of simulated planets.
- **M\_min** (*float*) – Minimum planet mass in Mearth
- **M\_max** (*float*) – Maximum planet mass in Mearth
- **S\_min** (*float*) – Minimum absolute instellation in W/m2
- **S\_max** (*float*) – Maximum absolute instellation in W/m2
- **depth\_min** (*float*) – Minimum transit depth

#### Returns

**d** – Table containing the new sample after applying the cuts.

#### Return type

[Table](#)

`bioverse.functions.Example1_water(d, f_water_habitable=0.75, f_water_nonhabitable=0.1, minimum_size=True, seed=42)`

Determines which planets have water, according to the following model:

$f(S, R) = f_{\text{water\_habitable}}$  if  $S_{\text{inner}} < S < S_{\text{outer}}$  and  $0.8 S^{0.25} < R < 1.4$   
 $= f_{\text{water\_nonhabitable}}$  if  $R > 0.8 S^{0.25}$

#### Parameters

- **d** ([Table](#)) – Table containing the sample of simulated planets.
- **f\_water\_habitable** (*float, optional*) – Fraction of potentially habitable planets (“exo-Earth candidates”) with atmospheric water vapor.
- **f\_water\_nonhabitable** (*float, optional*) – Fraction of non-habitable planets with atmospheric water vapor.
- **minimum\_size** (*bool, optional*) – Whether or not to enforce a minimum size for non-habitable planets to have H2O atmospheres.
- **seed** (*int or 1-d array\_like, optional*) – Seed for numpy’s RandomState. Must be convertible to 32 bit unsigned integers.

#### Returns

**d** – Table containing the sample of simulated planets.

#### Return type

[Table](#)

`bioverse.functions.Example2_oxygen(d, f_life=0.7, t_half=2.3, seed=42)`

Applies the age-oxygen correlation from Example 2.

#### Parameters

- **d** ([Table](#)) – Table containing the sample of simulated planets.
- **f\_life** (*float, optional*) – Fraction of EECs (Earth-sized planets in the habitable zone) with life.
- **tau** (*float, optional*) – Timescale of atmospheric oxygenation (in Gyr), i.e. the age by which 63% of inhabited planets have oxygen.

- **seed**(*int or 1-d array\_like, optional*) – Seed for numpy’s RandomState. Must be convertible to 32 bit unsigned integers.

**Returns**

**d** – Table containing the sample of simulated planets.

**Return type**

*Table*

`bioverse.functions.magma_ocean(d, wrt=0.005, S_thresh=280.0, simplified=False, diff_frac=0.54, f_rgh=1.0, gh_increase=True, water_incorp=True)`

Assign a fraction of planets global magma oceans that change the planet’s radius.

### 3.17.1 Parameters:

**d**

[Table] The population of planets.

**wrt**

[float, optional] water-to-rock ratio for Turbet+2020 model. Defines the amount of radius increase due to a steam atmosphere. Possible values: [0, 0.0001, 0.001 , 0.005 , 0.01 , 0.02 , 0.03 , 0.04 , 0.05 ] (default: 0.01 = 1% water) If wrt=0, the pure rock MR relation of Zeng+2016 is applied.

**S\_thresh**

[float, optional] threshold instellation for runaway greenhouse phase (in W/m2)

**simplified**

[bool, optional] increase the radii of all runaway greenhouse planets by the same fraction

**diff\_frac**

[float, optional] fractional radius change in the simplified case. E.g., diff\_frac = -0.10 is a 10% decrease.

**f\_rgh**

[float, optional] fraction of planets within the runaway gh regime that have a runaway gh climate

**gh\_increase**

[bool, optional] wether or not to consider radius increase due to runaway greenhouse effect (Turbet+2020)

**water\_incorp**

[bool, optional] wether or not to consider water incorporation in the melt of global magma oceans (Dorn & Lichtenberg 2021)

**returns**

**d** – Table containing the sample of simulated planets with new columns ‘has\_magmoocean’.

**rtype**

Table

## 3.18 bioverse.generator module

This module defines the Generator class and demonstrates how to create new Generators.

**class** `bioverse.generator.Generator`(*label=None*)

Bases: *Object*

This class executes a series of functions to generate a set of nearby systems.

### Parameters

**label** (*str*, *optional*) – Name of the Generator. Leave as None to create a new Generator.

### steps

List of Steps to be performed sequentially by the Generator.

### Type

list of *Step*

### copy()

Returns a deep copy of the Generator.

### initialize(*filename*)

Creates a new program from a CSV file containing the function names and descriptions.

### Parameters

**filename** (*str*) – Name of the CSV file from which to initialize the Generator.

### update\_steps(*reload=False*)

Loads or re-loads the keyword arguments and description of each step in the Generator.

### Parameters

**reload** (*bool*) – Whether or not to reload the default values for the arguments.

### insert\_step(*function*, *idx=None*, *filename=None*)

Inserts a step into the program sequence at the specified index.

### Parameters

- **function** (*str* or *function*) – Name of the function to be run by this step *or* the function itself.
- **idx** (*int*, *optional*) – Position in the program at which to insert the step. Default is -1 i.e. at the end.
- **filename** (*str*, *optional*) – Filename containing the function.

### replace\_step(*new\_function*, *idx*, *new\_filename=None*)

Replaces a step into the program sequence at the specified index.

### Parameters

- **new\_function** (*str* or *function*) – Name of the function to be run by this step *or* the function itself.
- **idx** (*int*) – Position in the program at which to replace the step.
- **new\_filename** (*str*, *optional*) – Filename containing the function.

### get\_arg(*key*)

Gets the default value of a keyword argument, and warns if there are multiple values.

**set\_arg**(*key, value*)

Sets the default value of a keyword argument for every step it applies to.

**generate**(*d=None, timed=False, idx\_start=0, idx\_stop=None, \*\*kwargs*)

Runs the generator with the current program and returns a simulated set of stars and planets.

**Parameters**

- **d** ([Table](#), *optional*) – Pre-existing table of simulated planets to be passed as input. If not specified, an empty table is created.
- **timed** (*bool*, *optional*) – If True, times each step in the program and prints the results.
- **idx\_start** (*int*, *optional*) – Specifies at which step in the program the Generator should start.
- **idx\_stop** (*int*, *optional*) – Specifies at which step in the program the Generator should stop.
- **\*\*kwargs** – Keyword argument(s) to be passed to the individual steps, e.g. `d_max=20`. Can have unintended consequences if the keyword argument appears in more than one step.

**Returns**

**d** – Table of simulated planets (plus host star parameters).

**Return type**

[Table](#)

**class** `bioverse.generator.Step`(*function, filename=None*)

Bases: `object`

This class runs one function for a Generator and saves its keyword argument values.

**Parameters**

- **function** (*str or function*) – Name of the function to be run *or* the function itself.
- **filename** (*str*, *optional*) – Name of the file containing the function for this step. If None, looks in `custom.py` and `functions.py`.

**description**

Docstring for this step's function.

**Type**

`str`

**run**(*d, \*\*kwargs*)

Runs the function described by this step.

**Parameters**

- **d** ([Table](#)) – Table of simulated planets to be passed as the function's first argument.
- **\*\*kwargs** – Keyword argument(s) to override. Ignores arguments which don't apply to this step.

**Returns**

**d** – Updated version of the simulated planet table.

**Return type**

[Table](#)

**get\_arg**(*key*)

Returns the value of a keyword argument.

**set\_arg**(*key*, *value*)

Sets the value of a keyword argument.

**Parameters**

- **key** (*str*) – Name of the argument whose value will be set.
- **val** – New value of the argument.

**find\_filename**()

If the filename is not specified, look in custom.py followed by functions.py.

**load\_function**(*reload=False*)

Loads or re-loads the function's description and keyword arguments.

**Parameters**

**reload** (*bool*, *optional*) – Whether or not to reload the default values for the arguments.

`bioverse.generator.reset_imaging_generator()`

Re-creates the default Generator for imaging surveys.

`bioverse.generator.reset_transit_generator()`

Re-creates the default Generator for transit surveys.

## 3.19 bioverse.hypothesis module

Defines the Hypothesis class as well as two hypotheses used in Bixel & Apai (2021).

**class** `bioverse.hypothesis.Hypothesis`(*f*, *bounds*, *params=()*, *features=()*, *labels=()*,  
*lnprior\_function=None*, *guess\_function=None*,  
*tfprior\_function=None*, *log=None*, *h\_null=None*, *\*\*kwargs*)

Bases: `object`

Describes a Bayesian hypothesis.

**Parameters**

- **f** (*function*) – Function describing the hypothesis. Must be defined as  $f(\theta, X)$  where  $\theta$  is a tuple of parameter values and  $X$  is a set of independent variables. Returns the calculated values of  $Y$ , the set of dependent variables for each entry in  $X$ .
- **bounds** (*array*) –  $N \times 2$  array describing the [min, max] limits of each parameter. These are enforced even if a different prior distribution is defined.
- **params** (*tuple of str*, *optional*) – Names of the parameter(s) of the hypothesis.
- **features** (*tuple of str*, *optional*) – Names of the feature(s) or independent variables.
- **labels** (*tuple of str*, *optional*) – Names of the label(s) or dependent variables.
- **lnprior\_function** (*function*, *optional*) – Used by emcee. Function which returns  $\ln(P_{\text{prior}})$ , must be defined as  $\text{prior}(\theta)$ . If None, assume a (log-)uniform distribution.
- **guess\_function** (*function*, *optional*) – Used by emcee. Function which guesses valid sets of parameters. Must be defined as  $\text{guess\_function}(n)$ , and should return an  $n \times m$  set of parameter guesses. If None, draw parameters randomly within *bounds*.

- **tfprior\_function** (*function, optional*) – Used by dynesty. Function which transforms (0, 1) into (min, max) with the appropriate prior probability. If None, assume a (log-)uniform distribution.
- **log** (*bool array, optional*) – Array of length N specifying which parameters should be sampled by a log-uniform distribution.
- **kwargs** (*key, value pairs*) – Additional keyword arguments (e.g., boolean switches) for the hypothesis function

**guess\_uniform**(*n, bounds*)

Default guess function. Guesses uniformly within self.bounds.

**guess**(*n*)

Guesses a set of values for theta, preferably where  $P(\theta) > -\infty$ .

**lnprior\_uniform**(*theta*)

Default (log-)uniform prior distribution, checks that all values are within bounds.

**lnprior**(*theta*)

Returns  $P(\theta)$  (for emcee).

**tfprior**(*u*)

**tfprior\_uniform**(*u*)

Transforms the unit cube *u* into parameters drawn from (log-)uniform prior distributions.

**lnlike\_binary**(*theta, x, y, \_*)

Likelihood function  $L(y | x, \theta)$  if *y* is binary. The last argument is a placeholder.

**lnlike\_multivariate**(*theta, x, y, sigma*)

Likelihood function  $L(y | x, \theta)$  if *y* is continuous and has *sigma* uncertainty.

**lnprob**(*theta, x, y, sigma*)

Posterior probability function  $P(\theta | x, y)$ .

**sample\_posterior\_dynesty**(*X, Y, sigma, nlive=100, nburn=None, verbose=False, sampler\_results=False*)

Uses dynesty to sample the parameter posterior distributions and compute the log-evidence.

**sample\_posterior\_emcee**(*x, y, sigma, nsteps=500, nwalkers=32, nburn=100, autocorr=False*)

Uses emcee to sample the parameter posterior distributions.

**compute\_AIC**(*theta\_opt, x, y, sigma*)

Computes the Akaike information criterion for optimal parameter set *theta\_opt*.

**compute\_BIC**(*theta\_opt, x, y, sigma*)

Computes the Bayesian information criterion for optimal parameter set *theta\_opt*.

**get\_observed**(*data*)

Identifies which planets in the data set have measurements of the relevant features/labels.

**get\_XY**(*data*)

Returns the X (features) and Y (labels) matrices for valid planets. Computes values as needed.

**fit**(*data, nsteps=500, nwalkers=16, nburn=100, nlive=100, return\_chains=False, verbose=False, method='dynesty', mw\_alternative='greater', return\_data=False, sampler\_results=False*)

Sample the posterior distribution of  $h(\theta | x, y)$  using a simulated data set, and compare to the null hypothesis via a model comparison metric.

### Parameters

- **data** ([Table](#)) – Simulated data set containing the features and labels.
- **nsteps** (*int*, *optional*) – Number of steps per MCMC walker.
- **nburn** (*int*, *optional*) – Number of burn-in steps for the Monte Carlo walk.
- **nlive** (*int*, *optional*) – Number of live points for the nested sampler.
- **return\_chains** (*bool*, *optional*) – Whether or not to return the Monte Carlo chains.
- **verbose** – Whether or not to generate extra output during the run.
- **method** (*str*, *optional*) – Which sampling method to use. Options: dynesty (default), emcee, mannwhitney,
- **mw\_alternative** (*str*, {'two-sided', 'less', 'greater'}, *optional*) – Defines the alternative hypothesis. Default is 'two-sided'. Let  $F(u)$  and  $G(u)$  be the cumulative distribution functions of the distributions underlying  $x$  and  $y$ , respectively. Then the following alternative hypotheses are available:
  - 'two-sided': the distributions are not equal, i.e.  $F(u) \neq G(u)$  for at least one  $u$ .
  - 'less': the distribution underlying  $x$  is stochastically less than the distribution underlying  $y$ , i.e.  $F(u) > G(u)$  for all  $u$ .
  - 'greater': the distribution underlying  $x$  is stochastically greater than the distribution underlying  $y$ , i.e.  $F(u) < G(u)$  for all  $u$ .
- **return\_data** (*bool*) – Whether or not to return the data
- **sampler\_results** (*bool*) – Whether or not to return the whole results object from dynesty runs

## Returns

**results** –

### Dictionary containing the results of the model fit:

'means' : mean value of each parameter's posterior distribution 'stds' : std dev of each parameter's posterior distribution 'medians' : median value of each parameter's posterior distribution 'UCIs' : 2-sigma confidence interval above the median 'LCIs' : 2-sigma confidence interval below the median 'CIs' : width of the  $\pm 2$  sigma confidence interval about the median 'AIC' : Akaike information criterion compared to the null hypothesis (i.e.  $AIC_{null} - AIC_{alt}$ ) 'BIC' : Bayesian information criterion compared to the null hypothesis 'chains' : full chain of MCMC samples (if *return\_chains* is True)

## Return type

dict

`bioverse.hypothesis.f_null(theta, X)`

Function for a generic null hypothesis. Returns ( $\theta_1$ ,  $\theta_2$ , ...) for each element in X.

`bioverse.hypothesis.f_HZ(theta, X)`

Function for the habitable zone hypothesis.

`bioverse.hypothesis.f_age_oxygen(theta, X)`

Function for the age-oxygen correlation hypothesis.

`bioverse.hypothesis.magma_ocean_hypo_exp(theta, X)`

Define a hypothesis for a magma ocean-adapted radius-sma distribution that follows an exponential decay.

## Parameters

- **theta** (*array\_like*) – Array of parameters for the hypothesis. *f\_magma* : float

fraction of planets having a magma ocean

**a\_cut: float**

cutoff effective sma for magma oceans. Defines position of the exponential decay.

**lambda\_a: float**

Decay parameter for the semi-major axis dependence of having a global magma ocean.

- **X (array\_like)** – Independent variable. Includes semimajor axis a.

**Returns**

Functional form of hypothesis

**Return type**

array\_like

`bioverse.hypothesis.magma_ocean_hypo_step(theta, X)`

Define a hypothesis for a magma ocean-adapted radius-sma distribution following a step function. Tests the hypothesis that the average planet size is smaller within the cutoff effective radius.

**Parameters**

- **theta (array\_like)** – Array of parameters for the hypothesis. f\_magma : float  
fraction of planets having a magma ocean
- **a\_cut: float**  
cutoff effective sma for magma oceans. Defines where the step occurs.
- **radius\_reduction: float**  
The fraction by which a planet's radius is reduced due to a global magma ocean.
- **R\_avg**  
[float] Average radius of the planets \_without\_ magma oceans.
- **X (array\_like)** – Independent variable. Includes semimajor axis a.

**Returns**

Functional form of hypothesis

**Return type**

array\_like

`bioverse.hypothesis.compute_avg_deltaR_deltaRho(stars_args, planets_args, transiting_only=True, savefile=True)`

Compute average radius and bulk density changes of the magma ocean-bearing planets as a function of water-to-rock ratio. This will be used to inform the magma ocean hypothesis function and avoids lengthy computations on each call of the hypothesis.

**Parameters**

- **stars\_args (dict)** – dictionary containing parameters for star generation. Should contain all non-default arguments for star-related generator modules.
- **planets\_args (dict)** – As stars\_args, but for planet-related generator modules.
- **transiting\_only (bool)** – Consider only transiting planets?
- **savefile (bool)** – Save data to file in `DATA_DIR + 'avg_deltaR_deltaRho.csv'`?

**Returns**

**avg\_deltaR\_deltaRho** – DataFrame containing the average radius/density differences.



### Return type

pandas DataFrame

`bioverse.hypothesis.get_avg_deltaR_deltaRho(path=None)`

Read pre-calculated radius and density differences.

`bioverse.hypothesis.magma_ocean_f0(theta, X)`

Define the null hypothesis that the radius distribution is random and independent of sma.

`bioverse.hypothesis.magma_ocean_hypo(theta, X, gh_increase=True, water_incorp=True, simplified=False, diff_frac=-0.1, parameter_of_interest='R', f_dR=None)`

Define a hypothesis for a magma ocean-adapted radius-sma distribution following a step function.

### Parameters

- **theta** (*array\_like*) – Array of parameters for the hypothesis. S\_thresh : float  
threshold instellation for runaway greenhouse phase
- **wrr**  
[float] water-to-rock ratio. Will be discretized to the grid used in Turbet+2020, with possible values [0, 0.0001, 0.001, 0.005, 0.01, 0.02, 0.03, 0.04, 0.05].
- **f\_rgh**  
[float] fraction of planets within the runaway gh regime that have a runaway gh climate
- **avg**  
[float] average planet radius or bulk density *outside* the runaway greenhouse region
- **X** (*array\_like*) – Independent variable. Includes effective semimajor axis a\_eff.
- **gh\_increase** (*bool, optional*) – whether or not to consider radius increase due to runaway greenhouse effect (Turbet+2020)
- **water\_incorp** (*bool, optional*) – whether or not to consider water incorporation in the melt of global magma oceans (Dorn & Lichtenberg 2021)
- **simplified** (*bool, optional*) – change the radii of all runaway greenhouse planets by the same fraction
- **diff\_frac** (*float, optional*) – fractional radius or bulk density change in the simplified case. E.g., diff\_frac = -0.10 is a 10% decrease.
- **parameter\_of\_interest** (*str, optional*) – ‘label’, i.e. the observable in which to search for the pattern. Can be ‘R’ or ‘rho’.
- **f\_dR** (*scipy.interpolate.interpolate.interp1d, optional*) – function that interpolates in the table containing pre-computed average radius and bulk density differences. If not provided, the values will be computed for a grid of water-to-rock ratios (this might be slow).

### Returns

Functional form of hypothesis

### Return type

array\_like

## 3.20 bioverse.plots module

`bioverse.plots.plot(d, starID=None, order=None, fig=None, canvas=None)`

`bioverse.plots.plot_universe(d, N_max=100, ax=None, mark=None)`

`bioverse.plots.plot_system(d, starID, ax=None, mark=None)`

Scatter plot for a single system with one or more planets.

### Parameters

- **d** ([Table](#)) – Table of simulated planets.
- **starID** (*int*) – Unique identifier of this system in the table (`d['starID']`).
- **ax** (*Axes, optional*) – Matplotlib Axes to plot the figure on. If not given, a new figure is created.
- **mark** (*int, optional*) – Indicates which planet to circle (if any).

`bioverse.plots.plot_spectrum(x, y, dy=None, xunit=None, yunit=None, lw=2)`

Plots a spectrum with or without errorbars.

`bioverse.plots.occurrence_by_class(d, compare=True)`

Plots the number of planets per star as a function of size and instellation.

`bioverse.plots.plot_binned_average(d, key1, key2, log=True, bins=10, method='mean',  
match_bin_counts=False, ax=None, return_xy=False, xm=None,  
ym=None, **kwargs)`

Plots the average value of a parameter as a function of another parameter.

### Parameters

- **d** ([Table](#)) – Table of simulated planets or measurements.
- **key1** (*str*) – Name of the parameter with which to bin the data set.
- **key2** (*str*) – Name of the parameter for which to calculate the average value.
- **log** (*bool, optional*) – Whether to bin the data in log-space.
- **bins** (*int or float array, optional*) – Number of bins or list of bin edges.
- **method** (`{'mean', 'median'}`, *optional*) – Whether to take the mean or median in each bin.
- **match\_bin\_counts** (*bool, optional*) – If True, calculate bins with an equal number of planets in each.
- **xm** (*float array, optional*) – X values of a model to be plotted along with the binned data.
- **ym** (*float array, optional*) – Y values of a model to be plotted along with the binned data.
- **\*\*kwargs** – Keyword arguments, passed to `matplotlib.pyplot.errorbar`.

`bioverse.plots.Example1_priority(generator, survey, fig=None, ax=None, show=True)`

Plots the prioritization of targets according to `a_eff` and `R` (or `R_eff`).

`bioverse.plots.Example1_targets(data, fig=None, ax=None, show=True, cbar=True, bins=10, vmin=None, vmax=None, cax=None, smooth_sigma=None)`

Plots the distribution of targets in  $\log(a_{\text{eff}})$  and  $R$  (or  $R_{\text{eff}}$ ).

`bioverse.plots.Example1_dataset(data, a_inner=0.944911182523068, a_outer=1.643989873053573, show=True, plot_model=True)`

Plots data[‘has\_H2O’] versus data[‘S’] for a simulated data set. Also plots the habitable zone boundaries.

`bioverse.plots.Example2_priority(generator, survey, fig=None, ax=None, show=True)`

`bioverse.plots.Example2_targets(data, fig=None, ax=None, bins=10, show=True)`

`bioverse.plots.Example2_dataset(data, flife=0.8, thalf=5.0, show=True, plot_model=False)`

Plots data[‘has\_O2’] versus data[‘age’] for a simulated data set. Also plots  $f(\text{O}_2 | \text{life})(t)$ .

`bioverse.plots.Example2_model(show=True, t_half=2.3, f_life=0.75)`

Plots the distribution of O2/O3-rich planets versus age.

`bioverse.plots.compare_posteriors(results_dict, **kwargs)`

`bioverse.plots.plot_posterior(chains, params=None, bounds=None, log=None, nbins=30, plot_model=True, show=True, fig=None, axes=None, **hist_kwargs)`

Plots the posterior distributions of a set of parameters.

#### Parameters

- **chains** (*float array (NxM)*) – N samples from the posterior distribution of each of M parameters
- **params** (*string list, optional*) – List of M parameter names in the order that they appear in *chains*. Use None to designate which parameters to exclude. If not specified, label as  $\theta_0, \theta_1, \dots$
- **bounds** (*float array (Mx2), optional*) – Describes each parameter’s min/max values.
- **log** (*tuple, optional*) – Length M tuple of True/False values indicating which posterior distributions should be plotted in log-scale.
- **nbins** (*int, optional*) – Number of bins the posterior distribution plot.
- **plot\_model** (*bool, optional*) – If True, overplot a normal distribution with the same mean and variance as the sample.
- **show** (*bool, optional*) – If True, display the plot. Otherwise, return the figure and axes.
- **fig** (*Figure, optional*) – Figure in which to create the plot. *axes* must also be passed.
- **axes** (*Axis list, optional*) – List of Axis objects in which to create the plots. *fig* must also be passed.

#### Returns

**fig, axes** – Figure and flattened array of Axes objects containing the plots. Only returned if *show* is False.

#### Return type

Figure, Axes array

`bioverse.plots.plot_power_grid(results, axes=('f_water_habitable', 'f_water_nonhabitable'), log=(True, True), labels=None, cbar=True, method='dlnZ', threshold=None, smooth_sigma=None, fig=None, ax=None, show=True, levels=[15, 60, 80], cmap='Greens', zoom_factor=0, **grid_kwargs)`

```

bioverse.plots.plot_requirements_grid(results, axes=('f_water_habitable', 'f_water_nonhabitable'),
                                     variable='t_total', log=(True, True), labels=None,
                                     var_label=None, levels=None, method='dlnZ', threshold=None,
                                     smooth_sigma=None, show=True, min_power=0.8, fmt='%.0f d ',
                                     N_key='N_EEC', vmin=None, vmax=None, fig=None, ax=None,
                                     cmap='Greens_r', zoom_factor=0, **grid_kwargs)

bioverse.plots.plot_habitable_zone_accuracy(results, a_inner=0.931, a_outer=1.674,
                                           smooth_sigma=None)

bioverse.plots.plot_Example1_constraints(results, fig=None, ax=None, show=True, c='black', lw=1.0,
                                         truth=True, **grid_kwargs)

bioverse.plots.plot_Example2_constraints(results, fig=None, ax=None, show=True, c='black', lw=1.0,
                                         **grid_kwargs)

bioverse.plots.image_contour_plot(x, y, z, colorbar=True, labels=None, levels=None, fmt='%.0f', ticks=4,
                                  vmin=None, vmax=None, linecolor='black', log=None, fig=None,
                                  ax=None, return_ctr=False, zoom_factor=None, cmap='Greens',
                                  plus=False, smooth_sigma=0)

    Plots z(x, y) with a colorbar and contours.

bioverse.plots.plot_number_vs_time(results, smooth_sigma=None, exclude=['N_pl', 'N_EEC'],
                                   **grid_kwargs)

bioverse.plots.plot_precision_grid(results, param='a_inner', axes=('f_water_habitable',
                          'f_water_nonhabitable'), labels=None, cbar=True, method='dlnZ',
                          threshold=None, smooth_sigma=None, show=True, half=False,
                          levels=3, log=None, fmt='%.2f AU', **grid_kwargs)

bioverse.plots.plot_precision(results, params=('a_outer', 'a_inner'), axes=('f_water_habitable',
                          'f_water_nonhabitable'), labels=None, cbar=True, method='dlnZ',
                          threshold=None, smooth_sigma=None, show=True, half=False, levels=([1.0,
                          3.0], [0.2, 0.6]), log=None, fmt='%.2f AU ', **grid_kwargs)

bioverse.plots.plot_simulation_result(results, log=True, method='dlnZ', **grid_kwargs)

bioverse.plots.plot_clear_cloudy_spectra(x, y_clr, y_cld, f_clouds=0.75, lw=3, bands=[], c=[],
                                         alpha=0.3, legend=True, xlim=None, ymax=None, fig=None,
                                         ax=None)

```

## 3.21 bioverse.survey module

```

class bioverse.survey.Survey(label: Optional[str] = None, diameter: float = 15.0, t_max: float = 3652.5,
                             t_slew: float = 0.1, T_st_ref: float = 5788.0, R_st_ref: float = 1.0, D_ref: float
                             = 15.0, d_ref: float = 10.0)

```

Bases: dict, *Object*

Describes an exoplanet survey, including methods for creating simulated datasets. This class should not be called directly; instead use ImagingSurvey or TransitSurvey.

**label:** str = None

**diameter:** float = 15.0

**t\_max:** float = 3652.5

**t\_slew:** float = 0.1

**T\_st\_ref:** float = 5788.0

**R\_st\_ref:** float = 1.0

**D\_ref:** float = 15.0

**d\_ref:** float = 10.0

**add\_measurement**(*key*, *idx=None*, *\*\*kwargs*)

Adds a Measurement to the Survey.

#### Parameters

- **key** (*str*) – Name of the measured parameter.
- **idx** (*int*) – Position in the measurement sequence. By default, it is placed at the end.
- **\*\*kwargs** – Keyword arguments passed to `Measurement.__init__()`.

**move\_measurement**(*key*, *idx*)

Moves a Measurement to the designated position in the sequence.

#### Parameters

- **key** (*str*) – Name of the measured parameter.
- **idx** (*int*) – Position in the measurement sequence to which to move the Measurement.

**quickrun**(*generator*, *t\_total=None*, *N\_sim=1*, *\*\*kwargs*)

Convenience function that generates a sample, computes the detection yield, and returns a simulated data set.

#### Parameters

- **generator** ([Generator](#)) – Generator used to generate the planet population.
- **t\_total** (*float*, *optional*) – Total amount of observing time for any measurements with a limited observing time.
- **N\_sim** (*int*, *optional*) – If greater than 1, simulate the survey this many times and return the combined result.
- **\*\*kwargs** – Keyword arguments passed to `Generator.generate()`.

#### Returns

- **sample** (*Table*) – Table of all simulated planets.
- **detected** (*Table*) – Table of planets detected by the Survey.
- **data** (*Table*) – Simulated data set produced by the Survey.
- **error** (*Table*) – Uncertainties on the measurements in *data*.

**observe**(*y*, *t\_total=None*, *data=None*, *error=None*, *demographics=True*)

Returns a simulated data set for a Table of simulated planets.

#### Parameters

- **y** (*Table*) – Table containing the set of planets to be observed, usually the detection yield of the survey.
- **t\_total** (*float, optional*) – Sets the total time allocated to all Measurements.
- **data** (*Table, optional*) – Pre-existing Table in which to store the new measurements.
- **error** (*Table, optional*) – Pre-existing Table containing uncertainties on *data* values.
- **demographics** (*Bool, optional*) – compute some population-level statistics after taking all measurements

**Returns**

**data** – Table of measurements made by the Survey, with one row for each planet observed.

**Return type**

*Table*

```
class bioverse.survey.ImagingSurvey(label: str = None, diameter: float = 15.0, t_max: float = 3652.5,
                                     t_slew: float = 0.1, T_st_ref: float = 5788.0, R_st_ref: float = 1.0,
                                     D_ref: float = 15.0, d_ref: float = 10.0, inner_working_angle: float =
                                     3.5, outer_working_angle: float = 64, contrast_limit: float = -10.6,
                                     mode: str = 'imaging')
```

Bases: *Survey*

**inner\_working\_angle:** float = 3.5

**outer\_working\_angle:** float = 64

**contrast\_limit:** float = -10.6

**mode:** str = 'imaging'

**compute\_yield**(*d*, *wl\_eff*=0.5, *A\_g*=0.3)

Computes a simple estimate of the detection yield for an imaging survey. Compares the contrast ratio and projected separation of each planet when observed at quadrature to the contrast limit and inner/outer working angles of the survey. Planets that satisfy these criteria are considered to be detected.

**Parameters**

- **d** (*Table*) – Table of all simulated planets which the survey could attempt to observe.
- **wl\_eff** (*float, optional*) – Effective wavelength of observation in microns (used for calculating the IWA/OWA).
- **A\_g** (*float, optional*) – Geometric albedo of each planet, ignored if 'A\_g' is already assigned.

**Returns**

**yield** – Copy of the input Table containing only planets which were detected by the survey.

**Return type**

*Table*

**compute\_scaling\_factor**(*d*)

Computes the scaling factor for the reference exposure time in imaging mode for all planets in *d*.

```
class bioverse.survey.TransitSurvey(label: str = None, diameter: float = 15.0, t_max: float = 3652.5,
                                     t_slew: float = 0.1, T_st_ref: float = 5788.0, R_st_ref: float = 1.0,
                                     D_ref: float = 15.0, d_ref: float = 10.0, N_obs_max: int = 1000,
                                     mode: str = 'transit')
```

Bases: *Survey*

**N\_obs\_max: int = 1000**

**mode: str = 'transit'**

**compute\_yield(*d*)**

Computes a simple estimate of the detection yield for a transit survey. All transiting planets are considered to be detected.

**Parameters**

**d** ([Table](#)) – Table of all simulated planets which the survey could attempt to observe.

**Returns**

**yield** – Copy of the input table containing only planets which were detected by the survey.

**Return type**

[Table](#)

**compute\_scaling\_factor(*d*)**

Computes the scaling factor for the reference exposure time in transit mode for all planets in *d*.

**class bioverse.survey.Measurement**(*key, survey, precision=0.0, t\_total=None, t\_ref=None, priority={}, wl\_eff=0.5, debias=True*)

Bases: object

Class describing a simple measurement to be applied to a set of planets detected by a Survey.

**Parameters**

- **key** (*str*) – Name of the parameter that will be measured.
- **survey** ([Survey](#)) – Survey associated with this Measurement.
- **precision** (*str or float, optional*) – Precision of measurement, e.g. '10%' or 0.10 units. Default is zero.
- **t\_ref** (*float, optional*) – Amount of time required to perform the measurement for a typical target, in days.
- **t\_total** (*float, optional*) – Total amount of time allocated for this measurement, in days.
- **priority** (*dict, optional*) – Describes how target weights are assigned based on target properties. For example {'R':[[1, 2, 5]]} assigns weight = 5 to planets with  $1 < R < 2$ .
- **wl\_eff** (*float, optional*) – Effective wavelength of observation, used to estimate SNR.
- **debias** (*bool, optional*) – (Transit mode) If True, weight targets by  $a/R_*$  to cancel the transit detection bias.

**measure**(*detected, data=None, error=None, t\_total=None*)

Produces the measurement for planets in a Table and places them into a data Table.

**Parameters**

- **detected** ([Table](#)) – Table containing detected planets.
- **data** ([Table](#), *optional*) – Table in which to store the measured values for each planet. If not given, then a new table is created.
- **error** ([Table](#), *optional*) – Table in which to store the measurement uncertainties. Must be given if *data* is given.
- **t\_total** (*float, optional*) – Total amount of time allocated to this Measurement. If None, use self.t\_total.

### Returns

- **data** (*Table*) – Table containing the measured values for each planet.
- **error** (*Table*) – Table containing the measurement uncertainties for each planet.

**set\_weight**(*key, weight, min=None, max=None, value=None*)

Adds a new rule for determining target weight. *weight* can be set for targets whose parameter fall within (*min, max*) or exactly match *value*.

### Parameters

- **key** (*str*) – Name of the parameter being checked.
- **weight** (*float*) – Weight of targets that meet the conditions.
- **min** (*float, optional*) – Minimum value of range. Default is -inf.
- **max** (*float, optional*) – Maximum value of range. Default is +inf.
- **value** (*int or str or bool, optional*) – Exact value with which to compare.

**compute\_observable\_targets**(*data, t\_total=None*)

Determines which planets are observable based on the total allotted observing time.

### Parameters

- **data** (*Table*) – Table of data values already measured for these planets.
- **t\_total** (*float, optional*) – Total observing time for this measurement. If None, use self.t\_total.

### Returns

**observable** – Specifies which planets in the table are observable within the allotted time.

### Return type

bool array

**compute\_exposure\_time**(*d*)

Computes the exposure time and number of observations required to characterize each planet in *d*.

**compute\_overhead\_time**(*d, N\_obs=1*)

Computes the overheads associated with each observation.

**compute\_weights**(*d*)

Computes the priority weight of each planet in *d*.

**compute\_debias**(*d*)

Removes detection biases from the data set (transit mode only).

**perform\_measurement**(*x*)

Simulates measurements of the parameter from a set of true values. Measurements are clipped to  $\pm 5$  sigma of the true value to avoid non-physical results.

### Parameters

**x** (*array*) – Array of true values on which to perform the measurement.

### Returns

**xm** – Array of measured values with the same length and type as *x*.

### Return type

array



`bioverse.survey.reset_imaging_survey()`

Re-creates the default imaging survey.

`bioverse.survey.reset_transit_survey()`

Re-creates the default transit survey.

## 3.22 bioverse.util module

Miscellaneous functions used elsewhere in the code.

`bioverse.util.bar(arg, do_bar=True)`

Given an iterable, returns a progress bar if tqdm is installed. Otherwise, returns the iterable.

### Parameters

- **arg** (*iterable*) – Iterable for which to return a progress bar.
- **do\_bar** (*bool*) – If False, return *arg* and don't display a progress bar.

### Returns

**tqdm** – If tqdm is installed, return a progress bar formed from *arg*. Otherwise, just return *arg*.

### Return type

iterable

`bioverse.util.get_type(x)`

`bioverse.util.is_bool(a)`

`bioverse.util.as_tuple(x)`

Returns the parameter as a tuple.

`bioverse.util.import_function_from_file(function_name, filename)`

`bioverse.util.get_planet_colors(d)`

`bioverse.util.cycle_index(vals, val, delta)`

`bioverse.util.nan_fill(a, dtype=None)`

`bioverse.util.get_order(N)`

`bioverse.util.mask_from_model_subset(pl, subset)`

`bioverse.util.compute_bin_centers(bins)`

Given a set of N bin edges, returns N-1 bin centers and half-widths.

`bioverse.util.compute_eta_Earth(d, by_type=True)`

Computes the value of eta Earth for a simulated sample of planets. Note this could be inaccurate if there are stars without planets which are usually not listed in the simulated sample, although the algorithm does attempt to correct for this.

### Parameters

- **d** (*Table*) – Simulated sample of planets.
- **by\_type** (*bool, optional*) – If True, calculate eta Earth separately for each spectral type.

`bioverse.util.compute_occurrence_multiplier(optimistic=False, optimistic_factor=3, N_pts=30)`

Determines the multiplier for occurrence rates and planet periods as a function of stellar mass.

`bioverse.util.update_stellar_catalog(d_max=100, filename='/home/docs/checkouts/readthedocs.org/user_builds/bioverse/envs/stable/lib/packages/bioverse-1.1.0-py3.7.egg/bioverse/Data/Gaia.csv')`

Updates the catalog of nearby sources from Gaia DR2 and saves it to a file. Requires astroquery.

`bioverse.util.get_xyz(pl, t=0, M=None, n=3)`

`bioverse.util.normal(a, b, xmin=None, xmax=None, size=1)`

`bioverse.util.binned_average(x, y, bins=10, match_counts=True)`

Computes the average value of a variable in bins of another variable.

#### Parameters

- **x** (*float array*) – Array of independent values along which to perform the binning.
- **y** (*float array*) – Array of dependent values to be averaged.
- **bins** (*int or float array, optional*) – Number of bins or array of bin edges.
- **match\_counts** (*bool, optional*) – If True, adjust the bin sizes so that an equal number of data points fall in each bin. Passing an array of bin edges for *bins* will override this setting.

#### Returns

- **bins** (*float array*) – Array of bin edges.
- **values** (*float array*) – Average value of y in each bin.
- **errors** (*float array*) – Uncertainty on *values* in each bin, i.e. the standard error on the mean.

`bioverse.util.compute_t_ref(filenames, t_exp, wl_min, wl_max, threshold=5, usecols=(0, 1, 2))`

Computes *t\_ref* for the detection of a spectroscopic feature. User must first use PSG or other tools to simulate spectra of the reference target with and without the feature of interest.

#### Parameters

- **filenames** (*((str, str))*) – Points to two PSG output spectra files - one where the atmosphere contains the species of interest, and one where it does not (the order does not matter).
- **t\_exp** (*float*) – Exposure time for the PSG simulations - must be identical for both.
- **wl\_min** (*float*) – Minimum wavelength of the absorption feature, same units as the PSG output.
- **wl\_max** (*float*) – Maximum wavelength of the absorption feature.
- **threshold** (*float, optional*) – SNR threshold for a confident detection.
- **usecols** (*((int, int, int), optional)*) – Specifies the column numbers corresponding to (wavelength, radiance, uncertainty) in the input files.

#### Returns

**t\_ref** – Exposure time required to reach the targeted detection SNR, same units as *t\_exp*.

#### Return type

float

`bioverse.util.compute_logbins(binWidth_dex, Range)`

Compute the bin edges for a logarithmic grid.

#### Parameters

- **binWidth\_dex** (*float*) – width of bins in log space (dex)
- **Range** (*Tuple*) – range for parameter

#### Returns

**bins** – bins for one dimension

#### Return type

array

### Example

```
>>> binWidth_dex = 1.0
>>> Range = (10., 1000.)
>>> compute_logbins(binWidth_dex, Range)
array([ 10., 100., 1000.] )
```

`bioverse.util.interpolate_df(xvals, df, xcol, ycol)`

Interpolate values in a pandas DataFrame.

#### Parameters

- **xvals** (*iterable*) – input values for which to search in the x column
- **df** (*pandas DataFrame*) – dataframe in which to interpolate. Expected to be sorted by xcol.
- **xcol** (*str*) – column with values we’re comparing to xval
- **ycol** (*str*) – column with interpolated output values

#### Returns

**y\_interp** – interpolated values

#### Return type

iterable

`bioverse.util.S2a_eff(S)`

Convert instellation in W/m2 to solar-equivalent semi-major axis.

`bioverse.util.a_eff2S(a_eff)`

Convert solar-equivalent semi-major axis to instellation in W/m2.

`bioverse.util.compute_moving_average(d, window=25)`

Compute rolling mean of radius and density and their uncertainties, ordered by instellation.

#### Parameters

- **d** (*Table*) – Table containing the sample of simulated planets.
- **window** (*int, optional*) – window size of the rolling mean

#### Returns

**d** – Table containing new columns for rolling mean of radius, density.

#### Return type

*Table*

`bioverse.util.get_ideal_bins(data, method='freedman')`

return optimal bins for a given 1D data set

`bioverse.util.binned_stats(df, x_param, y_param, bins=None, statistic='mean', scale='log')`

Compute a binned statistic of parameter y's mean with respect to bins in parameter x.

`bioverse.util.compute_binned_average(d, x_param='S_abs', y_params=['R', 'rho'])`

Compute mean of radius and density and their uncertainties, binned in instellation.

#### Parameters

- **d** ([Table](#)) – Table containing the sample of simulated planets.
- **x\_param** (*str*) – Parameter axis along which we want to bin.
- **y\_params** (*str or iterable*) – Parameter(s) on which the binned average will be computed.

#### Returns

**d** – Table containing new columns for rolling mean of radius, density.

#### Return type

[Table](#)

`bioverse.util.generate_generator(g_args, stars_only=False, **kwargs)`

Helper function to create a planet generator.

`bioverse.util.find_distance4samplesize(N_target, g_args, tolerance=2, max_iterations=10, h=5)`

Iteratively find the distance d\_max needed to achieve a specified planet sample size. Uses the Secant method for root-finding. Sort of.

#### Parameters

- **N\_target** (*int*) – target sample size
- **g\_args** (*dict*) – arguments for generator object
- **tolerance** (*int*) – range around N\_target into which we need to land
- **max\_iterations** (*int*) – maximum number of iterations
- **h** (*int*) – distance delta (in pc) used for initial guess

#### Returns

- **N** (*int*) – closest sample size achieved
- **d0** (*float*) – distance at closest sample size

## PYTHON MODULE INDEX

### b

- `bioverse.analysis`, 52
- `bioverse.classes`, 53
- `bioverse.constants`, 55
- `bioverse.custom`, 55
- `bioverse.functions`, 55
- `bioverse.generator`, 63
- `bioverse.hypothesis`, 65
- `bioverse.plots`, 70
- `bioverse.survey`, 72
- `bioverse.util`, 77



## A

`a_eff2S()` (in module *bioverse.util*), 79  
`add_measurement()` (*bioverse.survey.Survey* method), 73  
`append()` (*bioverse.classes.Table* method), 54  
`apply_bias()` (in module *bioverse.functions*), 60  
`as_tuple()` (in module *bioverse.util*), 77  
`assign_mass()` (in module *bioverse.functions*), 59  
`assign_orbital_elements()` (in module *bioverse.functions*), 58

## B

`bar()` (in module *bioverse.util*), 77  
`binned_average()` (in module *bioverse.util*), 78  
`binned_stats()` (in module *bioverse.util*), 79  
`bioverse.analysis`  
    module, 52  
`bioverse.classes`  
    module, 53  
`bioverse.constants`  
    module, 55  
`bioverse.custom`  
    module, 55  
`bioverse.functions`  
    module, 55  
`bioverse.generator`  
    module, 63  
`bioverse.hypothesis`  
    module, 65  
`bioverse.plots`  
    module, 70  
`bioverse.survey`  
    module, 72  
`bioverse.util`  
    module, 77

## C

`classify_planets()` (in module *bioverse.functions*), 59  
`clear()` (*bioverse.classes.Stopwatch* method), 54  
`compare_methods()` (in module *bioverse.analysis*), 53  
`compare_posteriors()` (in module *bioverse.plots*), 71

`compute()` (*bioverse.classes.Table* method), 54  
`compute_AIC()` (*bioverse.hypothesis.Hypothesis* method), 66  
`compute_avg_deltaR_deltaRho()` (in module *bioverse.hypothesis*), 68  
`compute_BIC()` (*bioverse.hypothesis.Hypothesis* method), 66  
`compute_bin_centers()` (in module *bioverse.util*), 77  
`compute_binned_average()` (in module *bioverse.util*), 80  
`compute_debias()` (*bioverse.survey.Measurement* method), 76  
`compute_eta_Earth()` (in module *bioverse.util*), 77  
`compute_exposure_time()` (*bioverse.survey.Measurement* method), 76  
`compute_habitable_zone_boundaries()` (in module *bioverse.functions*), 59  
`compute_logbins()` (in module *bioverse.util*), 78  
`compute_moving_average()` (in module *bioverse.util*), 79  
`compute_observable_targets()` (*bioverse.survey.Measurement* method), 76  
`compute_occurrence_multiplier()` (in module *bioverse.util*), 77  
`compute_overhead_time()` (*bioverse.survey.Measurement* method), 76  
`compute_scaling_factor()` (*bioverse.survey.ImagingSurvey* method), 74  
`compute_scaling_factor()` (*bioverse.survey.TransitSurvey* method), 75  
`compute_statistical_power()` (in module *bioverse.analysis*), 52  
`compute_t_ref()` (in module *bioverse.util*), 78  
`compute_transit_params()` (in module *bioverse.functions*), 60  
`compute_weights()` (*bioverse.survey.Measurement* method), 76  
`compute_yield()` (*bioverse.survey.ImagingSurvey* method), 74  
`compute_yield()` (*bioverse.survey.TransitSurvey* method), 75  
`contrast_limit` (*bioverse.survey.ImagingSurvey*

attribute), 74  
 copy() (bioverse.classes.Table method), 54  
 copy() (bioverse.generator.Generator method), 63  
 create\_planet\_per\_star() (in module bioverse.functions), 58  
 create\_planets\_bergsten() (in module bioverse.functions), 57  
 create\_planets\_SAG13() (in module bioverse.functions), 57  
 create\_stars\_Gaia() (in module bioverse.functions), 56  
 cycle\_index() (in module bioverse.util), 77

## D

D\_ref (bioverse.survey.Survey attribute), 73  
 d\_ref (bioverse.survey.Survey attribute), 73  
 description (bioverse.generator.Step attribute), 64  
 diameter (bioverse.survey.Survey attribute), 72

## E

effective\_values() (in module bioverse.functions), 60  
 Example1\_dataset() (in module bioverse.plots), 71  
 Example1\_priority() (in module bioverse.plots), 70  
 Example1\_targets() (in module bioverse.plots), 70  
 Example1\_water() (in module bioverse.functions), 61  
 Example2\_dataset() (in module bioverse.plots), 71  
 Example2\_model() (in module bioverse.plots), 71  
 Example2\_oxygen() (in module bioverse.functions), 61  
 Example2\_priority() (in module bioverse.plots), 71  
 Example2\_targets() (in module bioverse.plots), 71

## F

f\_age\_oxygen() (in module bioverse.hypothesis), 67  
 f\_HZ() (in module bioverse.hypothesis), 67  
 f\_null() (in module bioverse.hypothesis), 67  
 find\_distance4samplesize() (in module bioverse.util), 80  
 find\_filename() (bioverse.generator.Step method), 65  
 fit() (bioverse.hypothesis.Hypothesis method), 66

## G

generate() (bioverse.generator.Generator method), 64  
 generate\_generator() (in module bioverse.util), 80  
 Generator (class in bioverse.generator), 63  
 geometric\_albedo() (in module bioverse.functions), 60  
 get\_arg() (bioverse.generator.Generator method), 63  
 get\_arg() (bioverse.generator.Step method), 64  
 get\_avg\_deltaR\_deltaRho() (in module bioverse.hypothesis), 69  
 get\_filename\_from\_label() (bioverse.classes.Object method), 53

get\_ideal\_bins() (in module bioverse.util), 79  
 get\_observed() (bioverse.hypothesis.Hypothesis method), 66  
 get\_order() (in module bioverse.util), 77  
 get\_planet\_colors() (in module bioverse.util), 77  
 get\_stars() (bioverse.classes.Table method), 54  
 get\_type() (in module bioverse.util), 77  
 get\_XY() (bioverse.hypothesis.Hypothesis method), 66  
 get\_xyz() (in module bioverse.util), 78  
 guess() (bioverse.hypothesis.Hypothesis method), 66  
 guess\_uniform() (bioverse.hypothesis.Hypothesis method), 66

## H

Hypothesis (class in bioverse.hypothesis), 65

## I

image\_contour\_plot() (in module bioverse.plots), 72  
 ImagingSurvey (class in bioverse.survey), 74  
 impact\_parameter() (in module bioverse.functions), 59  
 import\_function\_from\_file() (in module bioverse.util), 77  
 initialize() (bioverse.generator.Generator method), 63  
 inner\_working\_angle (bioverse.survey.ImagingSurvey attribute), 74  
 insert\_step() (bioverse.generator.Generator method), 63  
 interpolate\_df() (in module bioverse.util), 79  
 is\_bool() (in module bioverse.util), 77

## K

keys() (bioverse.classes.Table method), 53

## L

label (bioverse.survey.Survey attribute), 72  
 legend() (bioverse.classes.Table method), 54  
 lnlike\_binary() (bioverse.hypothesis.Hypothesis method), 66  
 lnlike\_multivariate() (bioverse.hypothesis.Hypothesis method), 66  
 lnprior() (bioverse.hypothesis.Hypothesis method), 66  
 lnprior\_uniform() (bioverse.hypothesis.Hypothesis method), 66  
 lnprob() (bioverse.hypothesis.Hypothesis method), 66  
 load\_function() (bioverse.generator.Step method), 65  
 luminosity\_evolution() (in module bioverse.functions), 55

## M

magma\_ocean() (in module bioverse.functions), 62  
 magma\_ocean\_f0() (in module bioverse.hypothesis), 69



- `magma_ocean_hypo()` (in module *bioverse.hypothesis*), 69
- `magma_ocean_hypo_exp()` (in module *bioverse.hypothesis*), 67
- `magma_ocean_hypo_step()` (in module *bioverse.hypothesis*), 68
- `mark()` (*bioverse.classes.Stopwatch* method), 54
- `mask_from_model_subset()` (in module *bioverse.util*), 77
- `measure()` (*bioverse.survey.Measurement* method), 75
- Measurement* (class in *bioverse.survey*), 75
- mode* (*bioverse.survey.ImagingSurvey* attribute), 74
- mode* (*bioverse.survey.TransitSurvey* attribute), 75
- module
- bioverse.analysis*, 52
  - bioverse.classes*, 53
  - bioverse.constants*, 55
  - bioverse.custom*, 55
  - bioverse.functions*, 55
  - bioverse.generator*, 63
  - bioverse.hypothesis*, 65
  - bioverse.plots*, 70
  - bioverse.survey*, 72
  - bioverse.util*, 77
- `move_measurement()` (*bioverse.survey.Survey* method), 73
- ## N
- N\_obs\_max* (*bioverse.survey.TransitSurvey* attribute), 74
- `name_planets()` (in module *bioverse.functions*), 58
- `nan_fill()` (in module *bioverse.util*), 77
- `normal()` (in module *bioverse.util*), 78
- `number_vs_distance()` (in module *bioverse.analysis*), 53
- `number_vs_eta()` (in module *bioverse.analysis*), 53
- `number_vs_time()` (in module *bioverse.analysis*), 53
- ## O
- Object* (class in *bioverse.classes*), 53
- `observe()` (*bioverse.survey.Survey* method), 73
- `observed()` (*bioverse.classes.Table* method), 54
- `occurrence_by_class()` (in module *bioverse.plots*), 70
- `outer_working_angle` (*bioverse.survey.ImagingSurvey* attribute), 74
- ## P
- `pdshow()` (*bioverse.classes.Table* method), 54
- `perform_measurement()` (*bioverse.survey.Measurement* method), 76
- `plot()` (in module *bioverse.plots*), 70
- `plot_binned_average()` (in module *bioverse.plots*), 70
- `plot_clear_cloudy_spectra()` (in module *bioverse.plots*), 72
- `plot_Example1_constraints()` (in module *bioverse.plots*), 72
- `plot_Example2_constraints()` (in module *bioverse.plots*), 72
- `plot_habitable_zone_accuracy()` (in module *bioverse.plots*), 72
- `plot_number_vs_time()` (in module *bioverse.plots*), 72
- `plot_posterior()` (in module *bioverse.plots*), 71
- `plot_power_grid()` (in module *bioverse.plots*), 71
- `plot_precision()` (in module *bioverse.plots*), 72
- `plot_precision_grid()` (in module *bioverse.plots*), 72
- `plot_requirements_grid()` (in module *bioverse.plots*), 72
- `plot_simulation_result()` (in module *bioverse.plots*), 72
- `plot_spectrum()` (in module *bioverse.plots*), 70
- `plot_system()` (in module *bioverse.plots*), 70
- `plot_universe()` (in module *bioverse.plots*), 70
- ## Q
- `quickrun()` (*bioverse.survey.Survey* method), 73
- ## R
- R\_st\_ref* (*bioverse.survey.Survey* attribute), 73
- `random_simulation()` (in module *bioverse.analysis*), 53
- `read()` (*bioverse.classes.Stopwatch* method), 55
- `read_stars_Gaia()` (in module *bioverse.functions*), 55
- `read_stellar_catalog()` (in module *bioverse.functions*), 56
- `replace_step()` (*bioverse.generator.Generator* method), 63
- `reset_imaging_generator()` (in module *bioverse.generator*), 65
- `reset_imaging_survey()` (in module *bioverse.survey*), 76
- `reset_transit_generator()` (in module *bioverse.generator*), 65
- `reset_transit_survey()` (in module *bioverse.survey*), 77
- `run()` (*bioverse.generator.Step* method), 64
- ## S
- `S2a_eff()` (in module *bioverse.util*), 79
- `sample_posterior_dynesty()` (*bioverse.hypothesis.Hypothesis* method), 66
- `sample_posterior_emcee()` (*bioverse.hypothesis.Hypothesis* method), 66
- `save()` (*bioverse.classes.Object* method), 53
- `scale_height()` (in module *bioverse.functions*), 60
- `set_arg()` (*bioverse.generator.Generator* method), 63
- `set_arg()` (*bioverse.generator.Step* method), 65
- `set_weight()` (*bioverse.survey.Measurement* method), 76

`shuffle()` (*bioverse.classes.Table* method), 54  
`sort_by()` (*bioverse.classes.Table* method), 53  
`split_key()` (*bioverse.classes.Table* method), 53  
`Step` (class in *bioverse.generator*), 64  
`steps` (*bioverse.generator.Generator* attribute), 63  
`stop()` (*bioverse.classes.Stopwatch* method), 55  
`Stopwatch` (class in *bioverse.classes*), 54  
`Survey` (class in *bioverse.survey*), 72

## T

`t_max` (*bioverse.survey.Survey* attribute), 73  
`t_slew` (*bioverse.survey.Survey* attribute), 73  
`T_st_ref` (*bioverse.survey.Survey* attribute), 73  
`Table` (class in *bioverse.classes*), 53  
`test_hypothesis_grid()` (in module *bioverse.analysis*), 52  
`test_hypothesis_grid_iter()` (in module *bioverse.analysis*), 52  
`tfprior()` (*bioverse.hypothesis.Hypothesis* method), 66  
`tfprior_uniform()` (*bioverse.hypothesis.Hypothesis* method), 66  
`to_pandas()` (*bioverse.classes.Table* method), 54  
`TransitSurvey` (class in *bioverse.survey*), 74

## U

`update_stellar_catalog()` (in module *bioverse.util*), 78  
`update_steps()` (*bioverse.generator.Generator* method), 63