ScopeSim
A flexible astronomical instrument data simulation environment in Python

K. Leschinski[a], H. Buddelmeijer[b], O. Czoske[a], M. Verdugo[a], G. Verdoes-Kleijn[b], and W. Zeilinger[a]

[a]Department of Astrophysics, University of Vienna
[b]Kapteyn Astronomical Institute, University of Groningen

ABSTRACT

ScopeSim is a flexible multipurpose instrument data simulation ecosystem built in python. It enables both raw and reduced observation data to be simulated for a wide range of telescopes and instruments quickly and efficiently on regular laptops. The software is currently being used to generate simulated raw input data for developing the data reduction pipelines for the MICADO and METIS instruments at the ELT. The ScopeSim environment consists of three main packages which are responsible for providing on-sky target templates (ScopeSim_templates), the data to build the optical models of various telescopes and instruments (instrument reference database), and the simulation engine (ScopeSim). This strict division of responsibilities allows ScopeSim to be used to simulate observation data for many different instrument and telescope configurations for both imaging and spectroscopic instruments. ScopeSim has been built to avoid redundant calculations where ever possible. As such it is able to deliver simulated observations on time scales of seconds to minutes. All the code and data is open source and hosted on Github. The community is also most welcome, and indeed encouraged to contribute to code ideas, target templates, and instrument packages.

Keywords: Instrument Data Simulation, Instrumentation, Data, Simulator, ELT, Python

1. INTRODUCTION

ScopeSim is a modular and flexible suite of python packages that enable many common astronomical optical (observatory/telescope/instrument) systems to be simulated. The suite of packages can be used by a wide audience for a variety of purposes; from the astronomer interested in simulating reduced observational data, to a pipeline developer needing raw calibration data for testing the pipeline.

ScopeSim achieves this level of flexibility by adhering to strict interfaces between the package, e.g: the ScopeSim engine package is completely instrument and object agnostic. All information and data relating to any specific optical configuration is kept exclusively in the instrument packages hosted in the instrument reference database (IRDB). The description of the on-sky source is kept exclusively within the target templates package (see ScopeSim_templates). Finally, the engine has no clue about what it is observing until run-time.

But why does the community need yet another instrument simulator? Until now, most instrument consortia have developed their own simulators. The general consensus is that every new instrument is sufficiently different from anything that has been previously developed, that it would make little sense to adapt already existing code. This statement is true to some extent. Every new instrument must differ in some way from all existing instruments in order for it to be useful to the astronomical community. However when looked at from a global perspective, every optical system is comprised primarily of elements common to all other systems. Atmospheric emission, mirror reflectivities, filter transmission curves, point spread functions, read-out noise, detector linearity, hot pixels, are just a few of the effects and artifacts that every astronomical optical system contains. Furthermore, while the amplitude and shape of each effect differs between optical systems, there are still commonalities in the way each effect can be described programmatically.

ScopeSim's main goal is to provide a framework for modelling (almost) any astronomical optical system by taking advantage of all these commonalities. What Astropy has done for the general python landscape in astronomy, ScopeSim aims to do for the instrument simulator landscape.

Further author information: (Send correspondence to Kieran Leschinski)
Kieran Leschinski: E-mail: kieran.leschinski@univie.ac.at

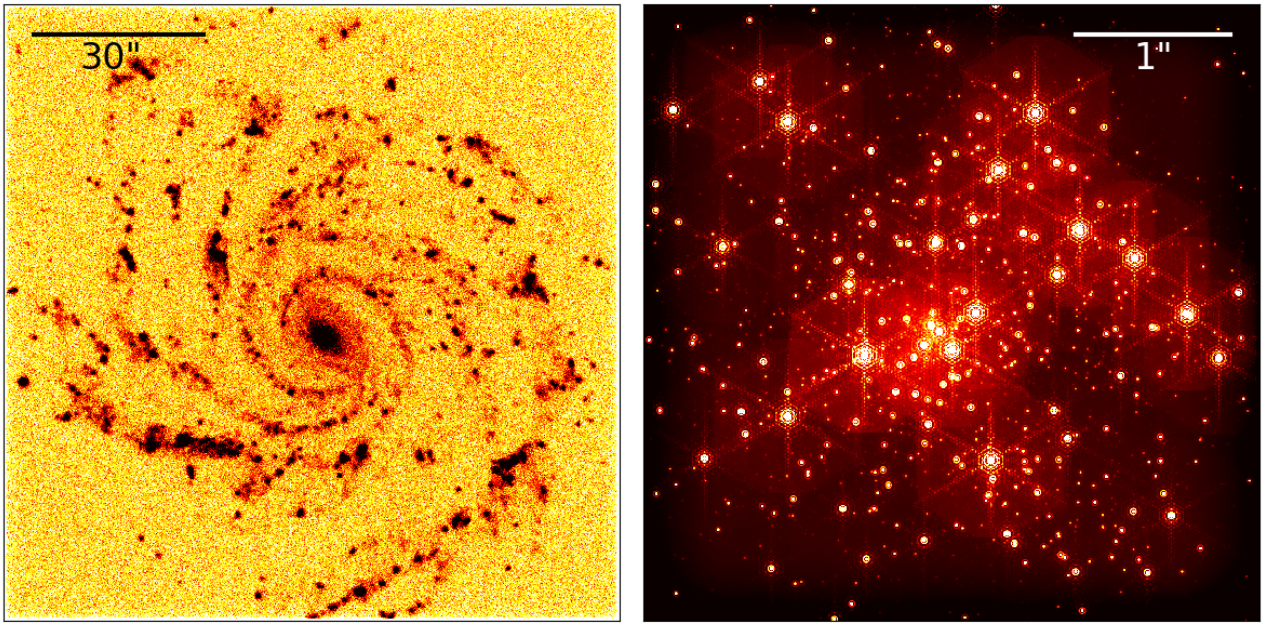

Figure 1. Left: A simulated one second observation in the Ks filter of a spiral galaxy similar to NGC 1232 using HAWKI at the VLT Right: A simulated one hour observation in the Ks filter of a dense 3000$M_\odot$ star cluster in the Large Magellanic Cloud using MICADO at the ELT.

This paper is not intended to be a comprehensive description of the `ScopeSim` environment. Rather it aims to introduce the reader to the elements that make up `ScopeSim` and directs the reader towards the online documentation for each of the packages, should the reader wish to dive deeper into the material.

2. EXAMPLES

The purpose of the ScopeSim ecosystem is to simulate data from astronomical instruments. ~~As the saying goes: Pictures are worth a thousand words.~~ This section will illustrate how this is done with very few lines of python code for three common science cases. It is the authors' hope that the code snippets provided are readable by the audience of this paper. If this is in fact not the case, the authors refer the reader to the scopesim online documentation.

2.1 Example 1: Extended source imaging

The first ~~part~~ example simulates a short (1 second) exposure with HAWKI[1] at the VLT in No-AO mode using the Ks filter. The ~~ob~~jet is a two component spiral galaxy using a template based on NGC,1232L from the `scopesim_templates` package. The galaxy was resized to a diameter of 3 arcminutes and the associated flux spectra[2] were rescaled to 12 mag arcsec^{-2}. The detector window of 1024×1024 pixels covers $\sim 1.6 \times 1.6$ arcminutes on sky. The final simulated image shows primarily the star forming regions in the inner regions of the spiral arms. The simulated detector output is shown in the left panel of Figure 2.1.

This simulation setup was chosen to illustrate the noise characteristics introduced by ScopeSim. The observation simulation requires only the following eight lines of code:

```
import scopesim
from scopesim_templates.basic.galaxy import spiral_two_component

scopesim.download_packages(["locations/Paranal",
                            "telescopes/VLT",
                            "instruments/HAWKI"])

spiral = spiral_two_component(extent=180,       # arcsec
```

```
                                fluxes=(15, 12))   # mag

hawki = scopesim.OpticalTrain("HAWKI")
hawki.cmds["!OBS.dit"] = 1                         # seconds
hawki.observe(spiral)
fits_hdulists = hawki.readout()
```

The simulation work flow will be discussed in more detail later in this paper. Briefly it involves downloading the instrument packages relevant to the observation, generating an on-sky source target, creating a model of the combined optical system, and observing and reading out the detectors in the instrument model. The output is a FITS HDUList, containing the instrument data in the format generated by the instrument. By default the HAWKI package produces images of size 1024×1024 pixels from a fictional detector window located at the centre of the focal plane. The package however also includes the configuration data needed to produce the standard 2×2 grid of 2048×2048 detector images. Switching between the two configurations is a simple matter of turning off the fictional detector window and turning on the realistic representation of the real detector array.

2.2 Example 2: Point source imaging

The major structures seen in the galaxy image produced in Example 1 are star forming regions. Given the pixel-scale of HAWKI ($0.106\,\mathrm{arcsec\,pixel^{-1}}$) it would be impossible to resolve the individual stars in these regions. MICADO on the ELT, with its $4\,\mathrm{mas\,pix^{-1}}$ pixel scale[3] and adaptive optics (AO) capabilities may well be able to detect individual stars in these regions.

The following code shows how to use the ELT and MICADO (Science Team) packages to simulate observations of highly dense star cluster outside the Milky Way. The result of this code is show in the right panel of Figure 2.1:

```
import scopesim
from scopesim_templates.basic.stars import cluster

scopesim.download_packages(["locations/Armazones",
                            "telescopes/ELT",
                            "instruments/MICADO_Sci"])

cluster = cluster(mass=3e3,                        # solar masses
                  distance=50e3,                   # parsec
                  core_radius=0.3)                 # parsec
micado = scopesim.OpticalTrain("MICADO_Sci")
micado.cmds["!OBS.dit"] = 3600                     # seconds
micado.observe(cluster)
fits_hdulists = micado.readout()
```

This code uses the star cluster template from the `scopesim_templates` package to create a model of a dense star cluster located in the Large Magellanic cloud ($D \sim 50\,\mathrm{kpc}$), with a core radius of 0.3 pc and a mass of $3000\,\mathrm{M}_\odot$. An exposure time of 1 hour with the Ks filter was used for the simulated observation. This setup was chosen to show the effect of the ELT PSF on observations of densely populated fields with several bright sources.

It should be noted that the instrument package used above (`MICADO_Sci`) is the slimmed down version of the full `MICADO` instrument package. Simulations using the `MICADO_Sci` package are less computationally intensive than when using the full `MICADO` package (which is aimed at pipeline development). The `MICADO_Sci` package was compiled specifically for the MICADO science team to test the feasibility of various science cases with MICADO and the ELT.

2.3 Example 3: Spectroscopy

The third example illustrates that ScopeSim can also be used to simulate spectroscopic observations. While MICADO is primarily a near infrared imaging camera, it will also contain a long-slit spectrograph. The spectroscopic mode of the `MICADO_Sci` package allows the user to simulate reduced spectral trace data over a restricted wavelength range data – similar to what can be expected as output from the MICADO data reduction pipeline.

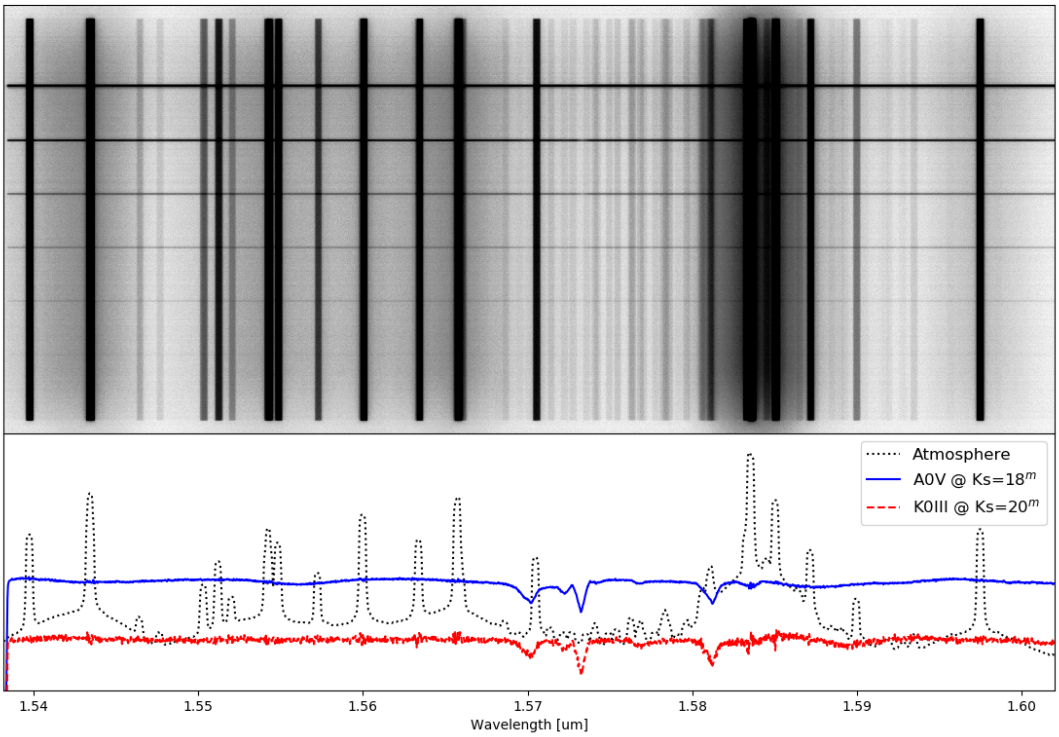

Figure 2. Top: A rectified spectral image from the MICADO detectors for a 1 hour spectrographic observation of 6 progressively fainter stars ($18 \leq K_s \leq 23$). The dark vertical bars are the atmospheric emission lines. The thin horizontal bars are the observed stellar spectra. The simulated wavelength range was restricted to $1.54 < \lambda < 1.6\,\mu$m. Bottom: Extracted spectra for the brightest ($K_s = 18$), third brightest ($K_s = 20$) stars, and the atmospheric background. The atmospheric background spectrum has been subtracted from the stellar spectra. The noise in the fainter stellar spectrum is a result of the simulated noise characteristics introduced by ScopeSim.

The following code simulates the spectral traces of 6 stars spaced equidistantly along the long-slit aperture with magnitudes in the range $K_s = [18, 23]$. In order to reduce computation time, the simulated wavelength range is restricted to 1024 spectral bins either side of a desired wavelength (1.578 µm).

```
import numpy as np
import astropy.units as u
from scopesim import UserCommands, OpticalTrain
from scopesim_templates.basic.stars import stars

stars = stars(filter_name="Ks",
              amplitudes=np.linspace(18, 23, 6)*u.mag,
              spec_types=["A0V", "G2V", "K0III"]*2,
              x=np.linspace(-1, 1, 6),
              y=[0]*6)
cmds = UserCommands(use_instrument="MICADO_Sci",
                    set_modes=["SCAO", "SPEC"],
                    properties={"!OBS.dit": 3600,
                                "!SIM.spectral.wave_mid": 1.578,
                                "!SIM.spectral.spectral_resolution": 0.00001,
                                "!DET.height": 2048,
                                "!DET.width": 800})
micado_spec = OpticalTrain(cmds)
```

```
micado_spec.observe(stars)
micado_spec.readout(filename="basic_spectral_trace.fits")
```

As can be seen in Fig. 2.3 the atmospheric emission lines are prominent in the simulated raw detector output. The 6 stellar spectra can be seen as thin horizontal lines. The spectra displayed in the lower panel of Fig. 2.3 were extracted for the detector readout in the upper panel. The noise in the (red) K0III spectrum is a product of the noise characteristics of the simulated observation. These include, but are not limited to photon shot noise and electronic noise sources.

2.4 Effects included in instrument packages

The instrument packages used for these examples can be found online in the Instrument Reference Database (IRDB) Github repository (see Section 5). Each package contains a description of the optical effects that are inherent to the instrument or telescope, as well as the data needed to replicate these effects. ScopeSim allows the user to view which effects are included in the current optical model. This example uses the `MICADO_Sci` optical system from the previous examples:

```
micado = scopesim.OpticalTrain("MICADO_Sci")
print(micado.effects)
```

During run-time ScopeSim creates an Effect object for each effect listed in the instrument configuration files. It then applies each of these Effect objects to the on-sky Source description in turn. Effects can be included or excluded from a simulation by using the `.include` flag on the relevant Effect object:

```
micado["readout_noise"].include = False
micado["shot_noise"].include = True
```

More information about the Effect objects is given in Section 4.2 as well as in the online documentation.

3. BUILDING BLOCKS

The ScopeSim ecosystem has been designed to maintain strict boundaries between the simulation code, the optical model data, and the user input. As with real systems, the astronomical objects and the light they emit exist regardless of the telescope or astronomer. The telescope and instruments also exist independently of the astronomer. Only when an astronomer uses a telescope to observe a celestial object are the three connected. A similar philosophy has been applied to the development of the ScopeSim ecosystem.

The ScopeSim ecosystem consists of three main packages:

- `ScopeSim`: the core simulation engine,
- `ScopeSim_templates`: a library of functions for generating descriptions of on-sky objects,
- `IRDB`: The instrument reference database containing the data and configuration files needed to generate the digital models of a range of telescope and instrument optical system.

Figure 3 illustrates the relationship between these three packages. Although there is a strict delineation between the scopes of each package, the interfaces between the packages allow them to interact almost seamlessly with each other. This can be seen in the code examples from Section 2. The main code pattern for simulating observations with a specific instrument is the same for all use cases:

1. download the required instrument packages from the instrument reference database (IRDB) using `scopesim`,
2. create a description of the astronomical object using `scopesim_templates`,
3. generate a model of the desired optical system using `scopesim` and referencing the `irdb` package,
4. simulate and output the observed data using `scopesim`.

The following subsections briefly describe each of these three packages.

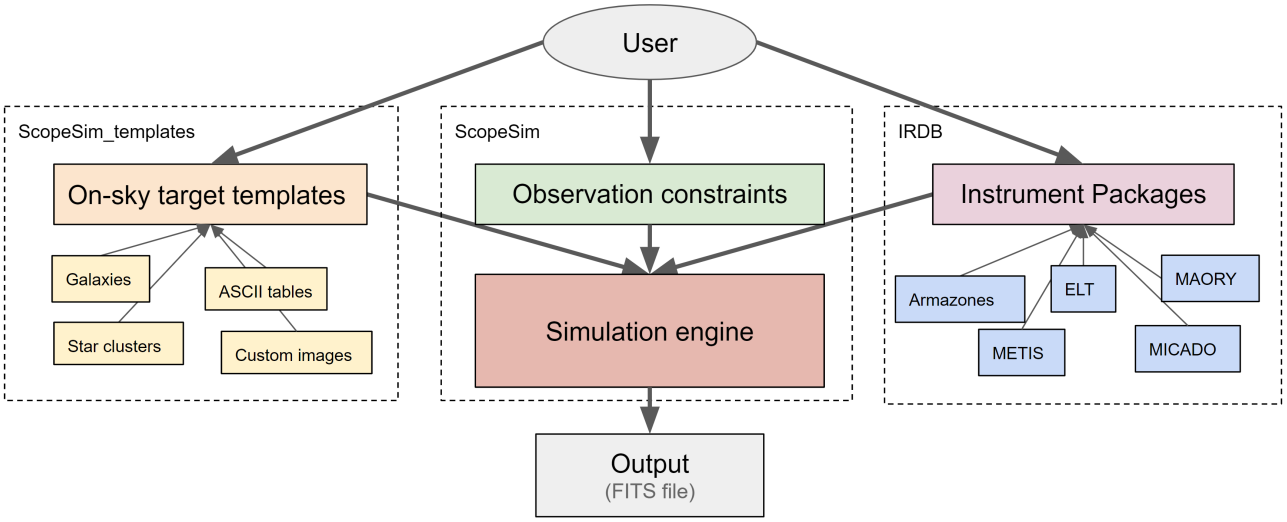

Figure 3. An illustration of the scopes of the three main packages in the ScopeSim environment. `ScopeSim_templates` (left) only provides the functionality to generate descriptions of on-sky objects in the format used by ScopeSim. The instrument reference database (right) contains the data and configuration files required to generate a model of an optical system in discrete instrument packages. The ScopeSim engine (centre) requires input from both of these packages, along with the observation constraints (e.g. exposure time, observing mode, etc.) in order to simulate mock observation data. The final output of a simulation is one or more FITS files containing the images or spectra of the user's target including all expected optical aberrations associated with the optical system.

3.1 `ScopeSim`: the observation simulator engine

The ScopeSim core package, also referred to as the ScopeSim engine, contains the code necessary for running observations simulations. The code has been written in such a way as to be completely agnostic to the instrument setup as well as agnostic towards the on-sky target. At its heart the code transports flux from a description of the on-sky target to a detector focal plane. During the process it applies any optical aberrations contained in the optical model to the flux description. Section 4 describes this process in more detail. ScopeSim attempts to remove as much redundancy and inefficiency from observation simulations as possible by recognising the fact that there is little need to redo the majority of the calculations executed in high-fidelity simulations. In other words, the ScopeSim engine does not work with ray-tracing methods or use Fourier optics, except in isolated cases. Instead it uses the fact that the observed image is a linear combination of independent optical aberrations applied to an incoming spectro-spatial flux distribution.

This focus on removing as many redundant calculations as possible results in very quick execution times. The images from the code examples were generated on a standard laptop in around 10 seconds. Such speed makes ScopeSim suitable for use cases with short iteration times, such a quick look feasibility studies, e.g. "playing" with a science case, or advanced exposure time calculators. At the other end of the scale, ScopeSim is also useful for generating simulated raw data needed during the development of instrument data reduction pipelines. As the ScopeSim engine takes its cues from the instrument packages, the fidelity of simulations is limited only the number and accuracy of the Effects listed in these packages.

3.2 `ScopeSim_templates`: Descriptions of on-sky targets

The `scopesim_templates` package provides a series of functions to help the user create a description of the on-sky targets they wish to observe in the format required by the `ScopeSim` engine. These helper function populate one or more instances of the `ScopeSim` `Source` class with the data needed to best describe the target.

3.2.1 Format of a `Source` object

In order to optimise memory usage the `Source` objects split the spatial and spectral characteristics of a target.[4] These are held separately in two lists: fields and spectra. A spatial field can be either a table of coordinated and flux scaling factors (e.g. the positions of stars in cluster) or a 2D weight map (e.g. an image of a galaxy). Each entry in a field must reference one of the entries in the list of spectra, although there is no requirement for a one-to-one relationship between field entries and spectral list entries. For example in the case of star cluster, all A0V stars can reference a single A0V

spectrum. ScopeSim memory requirements are further reduced by taking advantage of this redundancy.

Galaxies and similar extended objects can also be adequately represented in this manner. A galaxy generally contains populations of stars (new, old, high- or low-metallicity, etc.) and in most cases observations do not resolve individual stars. Hence it can be assumed that if each component is represented by a unique flux weight map (i.e. an image) then each stellar population can be represented by a single spectrum. As an example, the scopesim_templates function galaxy.spiral_two_component uses the B filter image of the galaxy NGC 1232 to represent the young population and the I filter image for the old population. Each image references a spectrum for a young or old population.

The extreme limit for this type of representation would be the case where every single pixel in an image is associated with a unique spectrum. An example might be the turbulent motions of gas in a star forming region, although it is still arguable that even here there will still be spectrally redundant regions. In this case the spectral and spatial components can still be split, although the result will be that each spatial field entry will consist of an image with only a single pixel and referencing the associated spectrum from the list of spectra. Such a use case would be particularly computationally expensive. It is therefore highly recommended in such cases to search for possible spectral redundancy before creating a Source object from a spectral cube.

3.2.2 Structure of ScopeSim_templates package

Currently ScopeSim_templates splits the helper functions into categories based on the complexity of the Source object that is produced. The basic subpackage contains helper functions that are useful for quick look investigations, but which should not be used for in-depth feasibility studies. For example the stars.cluster function does not allow age or metallicity to be set. In contrast the advanced subpackage contains functions that can be useful for very specific science cases, but are not adapted for general use.

In addition to the general functions, it is possible to add helper functions for objects used by specific instruments. The micado subpackage for example contains functions that produce objects specific to the MICADO instrument at the ELT.

Community participation is most welcome to help expand the number of object template function in the ScopeSim_templates package.

3.3 Instrument Reference Database: The optical model data

ScopeSim aims to be a general-purpose instrument data simulator that can be used to simulate the output of many different optical systems. To make this goal a reality it was mandatory that the scopesim engine be completely instrument agnostic. There is however a large amount of instrument specific data that is needed to accurately model the optical aberrations inherent in any optical system. For ScopeSim this data is stored in instrument packages in a separate instrument reference database (IRDB). Instrument packages can be created for any self-contained section of an optical train. For example, the telescope, the atmosphere, the relay optics, and the instrument are generally assumed to be self-contained optical sections. For small observatories like the University of Vienna's 1.5 m telescope there is no benefit to splitting the optical elements into separate packages. However, for large observatories like the VLT where multiple instruments can be attached to a single telescope, it makes sense to split the telescope optical system from the instrument description. Not only does this avoid multiple versions of a single optical element (e.g. telescope) becoming unsynchronised when one instrument package is updated and another is not, it also reduces the scope of responsibility for maintaining packages. For example, this means that instrument consortia need only concentrate on maintaining their own instrument package, while the telescope operator is responsible for maintaining the telescope package. It also means a telescope or relay optics package can be updated without needing (theoretically) to inform the maintainers of all instrument packages that use those subsystem.

3.3.1 Instrument package format

Each instrument package contains two main types of data:

1. A series of configuration files describing which optical aberrations should be modelled by scopesim, and
2. The empirical data files needed for scopesim to apply the aberrations to the incoming photon flux.

The configuration files are written in YAML. They contain lists of Effect object descriptions as well as global properties that are common to all Effect objects in the subsystem. The Effect object descriptions must call an existing Effect class from the ScopeSim core package. Effect objects are discussed in more detail in section 4.2. For the Effects

that rely on external empirical data, these files must also be contained in the instrument package. The empirical data files must be either ASCII tables or FITS images/tables. Examples of empirical data files include the filter response curves or pre-computed sets of PSF kernels.

The raw instrument data currently resides in the instrument reference database on Github. Periodically, or when explicitly needed, the data on this repository are compiled into packages and uploaded onto the `ScopeSim` server. It is from here that `ScopeSim` downloads a package when asked to do so by the user (as seen in the code examples). Packages are downloaded using `Astropy`, and hence are saved locally in the `Astropy` cache. This allows the packages to be used offline. Updated packages can be downloaded by either clearing the `Astropy` cache, or by forcing `scopesim` to redownload a package via the RC parameters. An example of this is available in the online documentation.

For readers interested in creating their own instrument packages for a local telescope or instrument, the authors recommend looking inside the `LFOA` (Leopold Figl Observatory for Astrophysics) package on the IRDB Github page. This contains everything needed to simulate observations with the Viennese 1.5 m telescope.

4. SCOPESIM ARCHITECTURE

In order to work as a multi-purpose optical* instrument simulator, `ScopeSim` needs to be able to handle (at least) the two main types of instruments: imagers and spectrographs.

While every instrument is unique, all instruments, by virtue of their astronomical nature, have several key aspects in common. All instruments:

- transport incoming photons through an optical system towards a detector (array),
- use a limited number of optical components, e.g: mirrors, lenses, and gratings,
- are only a single element in a combined optical train, which includes the atmosphere, telescope, and relay optics,
- introduce a series of optical aberrations depending on the configuration of the optical system,
- are generally built to behave in a predictable and repeatable manner.

These five points are important to recognise, as they have the following consequences:

- each optical element is responsible for one or more optical aberrations, which are not dependent on the aberrations inherent to the other optical elements,
- the effect of each aberration on the spatial and spectral distribution of photons remains constant for a given optical configuration,
- this constancy means the characteristics of these effects need only be calculated once and can be described by an analytical function, or an empirical data set,
- common elements (e.g. telescopes, atmospheres, etc.) of complex optical trains can be re-used with different instruments to create new combined optical systems.

This list of consequences implies that the final observed image from a telescope/instrument optical system is simply the sum of a discrete number of independent optical effects repositioning the incoming photons on the focal plane.

While this conclusion may seem obvious and trivial, by using it as the basis for `ScopeSim`, it has allowed us to design and build a flexible, lightweight, general purpose instrument simulator that is capable of simulating the majority of current and future optical astronomical instruments. `ScopeSim` is able to mimic the optical aberrations seen in imagers, long-slit and multi-object spectrographs, as well as integral-field spectrographs. The architecture could also theoretically be used to simulate high-contrast and high-time-resolution imagers, however these systems have not yet been tested.

*Optical refers to the wavelength ranges where telescopes act as "photon buckets" and detectors are in essence "photon counters", i.e. from the near ultraviolet (0.1 μm) to the mid infrared (30 μm).

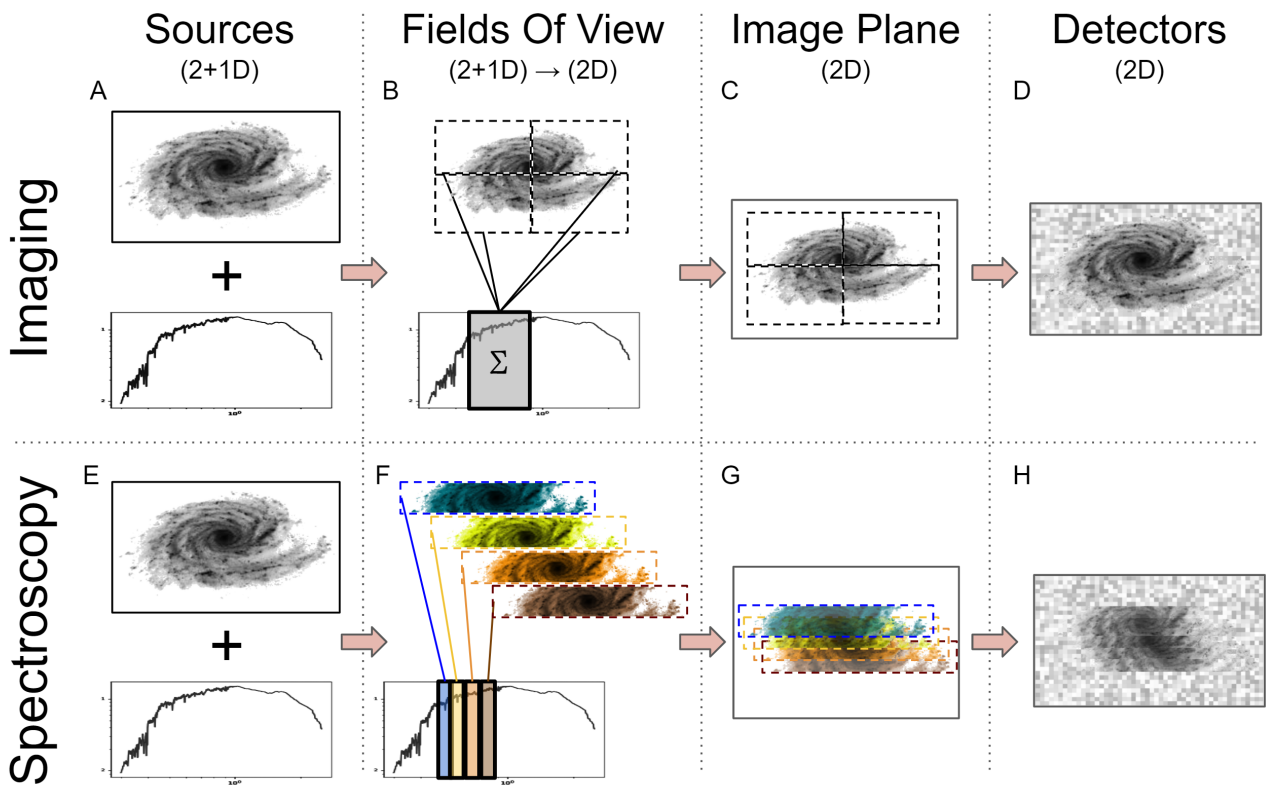

Figure 4. An illustration of the connections between the main internal classes in `ScopeSim`: `Source`, `FieldOfView`, `ImagePlane`, `DetectorArray`. The upper panels show the work flow for imaging simulations. The lower panels show the work flow for spectroscopy simulations. The work flow is in principle the same for both types of simulation. (A, E) Both modes require a 2+1D description of the on-sky target(s) containing linked spatial (2D) and spectral (1D) information. The main difference lies in how and where the spatial and spectral borders for each `FieldOfView` object are set. `FieldOfView` objects (B, E) extract (2D) integrated photon maps from the `Source` object(s) and project these onto an `ImagePlane` object, which creates a normalised expectation image, similar to what happens at the detector focal plane in a real instrument. The `DetectorArray` (D, H) extracts the regions of the `ImagePlane` that each detector would see. Simulation output in both imaging and spectroscopic cases is the same: A FITS file with detector read images in the same format as generated by the real instrument.

4.1 Simulation workflow

The main `ScopeSim` engine architecture is based around five major python classes:

- **`Source`**: holds a spectro-spatial description of the on-sky target.
- **`FieldOfView`**: extracts quasi-monochromatic flux maps from a `Source` object and projects these into focal plane coordinates.
- **`ImagePlane`**: mimics the focal plane and acts like a 2D canvas for collecting the flux maps held in the `FieldOfView` objects.
- **`DetectorArray`**: mimics the functionality of the instrument detector array in converting the final expectation flux image from the `ImagePlane` into FITS format pixel maps similar to those delivered by the systems read-out electronics.
- **`Effect`**: the interface base class for introducing spectral and spatial aberrations into the final flux map.

Figure 4.1 illustrates how the first four of these classes interact with each other.

The `Source` objects (A, E) are supplied by the user. These contain a 2+1D description of the on-sky target(s). The spatial

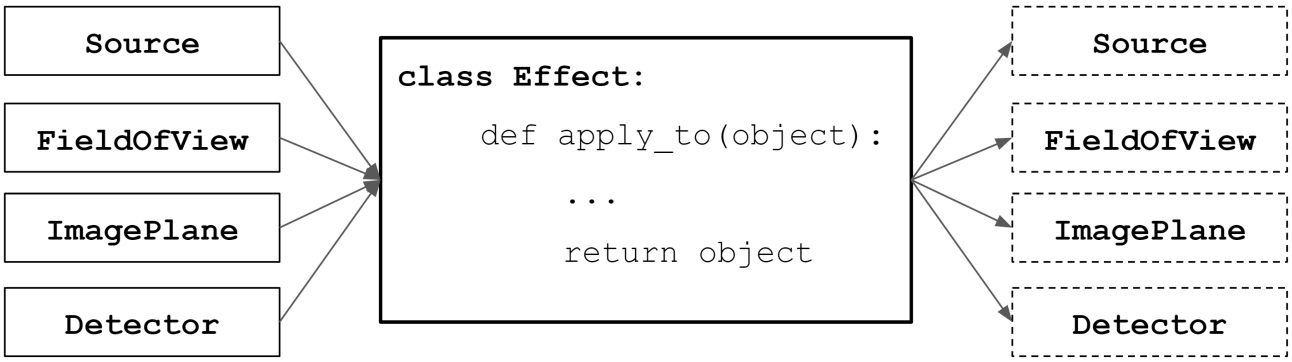

Figure 5. Effect objects are similar to matrix operator in mathematics. What goes in must come out. Each `Effect` object has a single point of entry: the `apply_to` method, which can accept any one of the four major `ScopeSim` classes. This method is responsible for applying optical aberrations to the flux distribution contained within those four major flux container classes.

(2D) information is stored either as tables (collection of point sources) or as `ImageHDU` objects (for extended objects). Each of the spatial "fields" must be accompanied by one or more unique spectrum. There need not be a one-to-one relationship between the spatial and spectral inputs. Multiple spatial fields can reference a single spectrum. In doing so, `ScopeSim` can vastly reduce the amount of data that needs to be processed. For example, a star cluster will contain many thousands of point sources. However, only several tens of spectra are needed to adequately describe all the stars in the cluster. There will be many hundreds of M-type stars that can reference a single common M-type stellar spectrum.

`ScopeSim` builds a model of the optical train by importing instrument packages. Based on the list of `Effects` contained in the configuration files, `ScopeSim` splits the full spectral and spatial parameter space of the instrument into 3D "puzzle" pieces, known as `FieldOfView` objects. Each `FieldOfView` object (B, F) then extracts only the flux from the `Source` object that fits within its spectro-spatial limits. This process essentially creates a series of quasi-monochromatic puzzle pieces from the 2+1D source flux. The spatial size and spectral depth of each puzzle piece is determined by which optical effects are included in the optical model. For imager instruments where chromatic effects rarely play a large role, the spectral depth of each `FieldOfView` object will be relatively large. It is not uncommon for the spectral depth to be equal to the spectral width of the filter bandpass. The on-sky area for imager is generally very large and so the viewing area is split into multiple pieces. This is illustrated in the upper half of panel B in Figure 4.1. For spectrographs, the spatial component is generally small (e.g. long slits, MOS fibre heads). The spectral space however must be very finely sampled to accurately reproduce the spectral traces on the focal plane. Spectrographic optical systems therefore contain many `FieldOfView` objects which cover the same on-sky spatial region, yet cover very shallow and unique spectral ranges. The `FieldOfView` objects also contain two sets of spatial coordinates which connect the object's position on sky (in units of arcseconds from the optical axis) to the projected position on the detector focal plane (in units of millimeters from the optical axis).

The `ImagePlane` (C, G) inside each optical model acts as a 2D canvas for the integrated flux contained inside the `FieldOfView` objects. When each `FieldOfView` object deposits its flux map onto the `ImagePlane`, it simply adds the photon counts to what is already on the canvas at the `FieldOfView`'s projected focal plane position. The resulting `ImagePlane` image is therefore the final integrated projected expectation flux map as would exist at the detector focal plane of a real image, in units of $\mathrm{ph\,s^{-1}\,pixel^{-1}}$. All information on telescope aperture, viewing angle, and spectral bandpass has been integrated into the normalised photon count map – thus the name "expectation" flux map. At this stage of the simulation all sources of background flux (atmospheric, thermal) have also been projected onto the `ImagePlane`, but no noise characteristics are included.

The `DetectorArray` class contains a list of `Detector` objects (D, H). `Detector` objects extract a region of the `ImagePlane`'s expection flux map corresponding to its own footprint on the detector focal plane and scales this to match the user's desired exposure time (DIT in seconds). The resulting image is the flux that a real detector would register in an ideal world. At this point all noise characteristics are introduced, e.g. shot noise, read noise, dark current, etc. The final detector output is returned in the form of a FITS `HDUList`.

4.2 `Effect` Objects

A further special and arguably the most important `ScopeSim` class is the `Effect` object. `Effect` objects are responsible for applying any and all optical aberrations to the flux descriptions contained in the other four major flux container classes. `Effect` objects can contain code to alter the flux descriptions in a multitude of manners, from simple 0D alterations like adding a dark current to each pixel, to the 3D chromatic shear caused by atmospheric refraction. In short, `Effect` objects can be classified according to the dimensionality of their alterations to the flux descriptions:

- 3D: Effects are spatially and spectrally dependent aberrations, e.g: the broadband point spread function, atmospheric diffraction, etc.,
- 2D: Effects are only spatially dependent, e.g: telescope vibration and wind shake, pupil tracking rotations, etc.,
- 1D: Effects are only spectrally dependent, e.g: reflection and transmission curves, quantum efficiency, etc.,
- 0D: Effects are spectrally and spatially independent. This are primarily effects that are related to photons counts and electronic noise sources, e.g: p Poisson shot noise, read-out noise, exposure stacking, detector linearity, etc.

Higher dimensional Effects are also possible albeit very rare, e.g. field varying PSFs.

Functionally, the `Effect` class is similar to a quantum mechanical operator. ~~What goes in must come out.~~ In other words, if a `Source` object is the input to an `Effect` object's `apply_to` function then a `Source` object will also be returned. The `Effect` object may alter the distribution of flux inside an object, but it must return the same object. This is illustrated in Figure 4.2.

During the simulation workflow, the target object flux makes its way through the four main class objects described in section 4.1. While flux resides in each of these objects, the relevant `Effects` are sequentially applied to said object. For example, the telescope's (chromatic) PSF is applied to each of the `FieldOfView` objects, as this is a spectrally dependent spatial (3D) effect. In contrast, the wind-shake gaussian PSF has no spectral dependency and is therefore only applied to the `ImagePlane`.

The following pseudo-code snippet describes the major steps of the simulation workflow and illustrates how and when the Effect objects interact with the four major flux container classes:

```
source = deepcopy(orig_source)

for effect in source_effects:
    source = effect.apply_to(source)

fov.extract_from(source)

for effect in fov_effects:
    fov = effect.apply_to(fov)

image_plane.add(fov)

for effect image_plane_effects:
    image_plane = effect.apply_to(image_plane)

detector.extract_from(image_plane)

for effect detector_effects:
    detector = effect.apply_to(detector)

detector.write_to("file.fits")
```

As can be seen, there is a very similar pattern. Obviously there are a few more steps involved, in the actual `ScopeSim` code, however the `observe` method of an optical model consists of little more than a python implementation of this

Table 1. Links to the open source documentation and code bases

Package	Documentation	Code base
ScopeSim	https://scopesim.readthedocs.io/	https://github.io/astronomyk/scopesim
ScopeSim_templates	https://scopesim-templates.readthedocs.io/	https://github.com/astronomyk/ScopeSim_templates
IRDB	https://irdb.readthedocs.io/en/latest/	https://github.com/astronomyk/IRDB
AnisoCADO	https://anisocado.readthedocs.io/	https://github.com/astronomyk/AnisoCADO
SkyCalc_ipy	https://skycalc-ipy.readthedocs.io/en/latest/	https://github.com/astronomyk/SkyCalc_iPy
SpeXtra	https://spextra.readthedocs.io/en/latest/	https://github.com/miguelverdugo/speXtra
Pyckles	https://pyckles.readthedocs.io/en/latest/	https://github.com/astronomyk/Pyckles

pseudo-code.

The authors of ScopeSim have already included a large number of standard Effects in the ScopeSim core package (see online documentation). It is clear however that there are many more that could be added. Community participation is always welcome. The Effect object interface has been intentionally kept light weight to encourage users to implement their own custom effects for their own simulations. The online documentation contains a tutorial on how to write custom effects. Users are thus cordially invited to submit any custom Effects they deem useful to the wider community to the ScopeSim package as a pull request via the Github repository.

5. FULL SCOPESIM ECOSYSTEM

At its core the ScopeSim environment contains three packages:

- ScopeSim: the simulation engine
- ScopeSim_templates: descriptions of the on-sky sources
- IRDB: the instrument reference database, which contains the data files used to build the instrument models as well as the configuration files which tell the ScopeSim engine how to simulate observations.

In addition to the core package, there are several support packages:

- AnisoCADO: simulates SCAO PSFs for the ELT
- SkyCalc_ipy: queries the ESO skycalc service for atmospheric spectral curves
- SpeXtra: provides easy access to many well-known spectral libraries
- Pyckles: a light-weight wrapper for the Pickles (1998) and Brown (2010) catalogues.

These package are not direct dependencies of ScopeSim, but do help provide additional functionality to the simulation engine. Table 1 contains a list of the relevant links to both documentation and code-bases for these packages.

Community involvement is highly encouraged! The whole ScopeSim ecosystem is open source and the developers welcome any contributions, both code and comments, by members of the astronomical community. The astronomical object templates package is one area which will benefit greatly from community contributions. There is a wide variety of astronomical objects for which the authors have not yet created templates. Galaxy clusters, gravitational lenses, supernovae, exoplanets, solar system objects are all still missing from the scopesim_templates package. The instrument reference database also currently only contains the instruments directly relevant to the authors. There is no limit to the size of

telescopes or number of instruments that can be hosted on the server. Readers interested in submitting a package for their own telescope or instrument are very welcome to make a pull request on the IRDB github page.

6. CONCLUSION

`ScopeSim` is a flexible multipurpose instrument data simulation ecosystem built in python. It enables both raw and ideal observation data to be simulated for a wide range of telescopes and instruments quickly and efficiently on regular laptops. This is achieved by keeping the instrument model data, descriptions of the target objects, and simulation engine strictly separated. The three main packages are the `ScopeSim` engine, a library of target templates, and the instrument reference database. Several existing and future telescope and instrument systems have been already been implemented, with more to come in the future. For example, work is steadily progressing on the instrument packages for the MICADO and METIS instruments at the ELT. All the code and data is open source and hosted on Github. The community is also most welcome, and indeed encouraged to contribute to code ideas, target templates, and instrument packages by either opening an issue or submitting a pull request on Github.

ACKNOWLEDGMENTS

`ScopeSim` incorporates parts of Bernhard Rauscher's HxRG Noise Generator package for python.[5] This research made use of `Astropy`, a community-developed core Python package for astronomy.[6,7] This research made use of `Synphot`.[8] `ScopeSim` makes use of atmospheric transmission and emission curves generated by ESO's SkyCalc service, which was developed at the University of Innsbruck as part of an Austrian in-kind contribution to ESO. This research is partially funded by the project IS538004 of the Hochschulraumstrukturmittel (HRSM) provided by the Austrian Government and administered by the University of Vienna. The authors would also like to thank all the members of the consortium for their effort in the MICADO project, and their contributions to the development of this tool.

REFERENCES

[1] Kissler-Patig, M., Pirard, J. F., Casali, M., Moorwood, A., Ageorges, N., Alves de Oliveira, C., Baksai, P., Bedin, L. R., Bendek, E., Biereichel, P., Delabre, B., Dorn, R., Esteves, R., Finger, G., Gojak, D., Huster, G., Jung, Y., Kiekebush, M., Klein, B., Koch, F., Lizon, J. L., Mehrgan, L., Petr-Gotzens, M., Pritchard, J., Selman, F., and Stegmeier, J., "HAWK-I: the high-acuity wide-field K-band imager for the ESO Very Large Telescope," *Astron. & Astroph.* **491**, 941–950 (Dec. 2008).

[2] Brown, M. J. I., Moustakas, J., Smith, J. D. T., da Cunha, E., Jarrett, T. H., Imanishi, M., Armus, L., Brandl, B. R., and Peek, J. E. G., "An Atlas of Galaxy Spectral Energy Distributions from the Ultraviolet to the Mid-infrared," *Astroph. Journ. Suppl.* **212**, 18 (June 2014).

[3] Davies, R., Alves, J., Clénet, Y., Lang-Bardl, F., Nicklas, H., Pott, J. U., Ragazzoni, R., Tolstoy, E., Amico, P., Anwand-Heerwart, H., Barboza, S., Barl, L., Baudoz, P., Bender, R., Bezawada, N., Bizenberger, P., Boland, W., Bonifacio, P., Borgo, B., Buey, T., Chapron, F., Chemla, F., Cohen, M., Czoske, O., Déo, V., Disseau, K., Dreizler, S., Dupuis, O., Fabricius, M., Falomo, R., Fedou, P., Förster Schreiber, N., Garrel, V., Geis, N., Gemperlein, H., Gendron, E., Genzel, R., Gillessen, S., Glück, M., Grupp, F., Hartl, M., Häuser, M., Hess, H. J., Hofferbert, R., Hopp, U., Hörmann, V., Hubert, Z., Huby, E., Huet, J. M., Hutterer, V., Ives, D., Janssen, A., Jellema, W., Kausch, W., Kerber, F., Kravcar, H., Le Ruyet, B., Leschinski, K., Mandla, C., Manhart, M., Massari, D., Mei, S., Merlin, F., Mohr, L., Monna, A., Muench, N., Müller, F., Musters, G., Navarro, R., Neumann, U., Neumayer, N., Niebsch, J., Plattner, M., Przybilla, N., Rabien, S., Ramlau, R., Ramos, J., Ramsay, S., Rhode, P., Richter, A., Richter, J., Rix, H. W., Rodeghiero, G., Rohloff, R. R., Rosensteiner, M., Rousset, G., Schlichter, J., Schubert, J., Sevin, A., Stuik, R., Sturm, E., Thomas, J., Tromp, N., Verdoes-Kleijn, G., Vidal, F., Wagner, R., Wegner, M., Zeilinger, W., Ziegleder, J., Ziegler, B., and Zins, G., "The MICADO first light imager for the ELT: overview, operation, simulation," in [*Ground-based and Airborne Instrumentation for Astronomy VII*], Evans, C. J., Simard, L., and Takami, H., eds., *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series* **10702**, 107021S (July 2018).

[4] Schmalzl, E., Meisner, J., Venema, L., Kendrew, S., Brandl, B., Blommaert, J., Glasse, A., Lenzen, R., Meyer, M., Molster, F., and Pantin, E., "An end-to-end instrument model for the proposed E-ELT instrument METIS," in [*Modeling, Systems Engineering, and Project Management for Astronomy V*], Angeli, G. Z. and Dierickx, P., eds., *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series* **8449**, 84491P (Sept. 2012).

[5] Rauscher, B. J., "Teledyne h1rg, h2rg, and h4rg noise generator," *PASP* **127**, 1144–1151 (nov 2015).

[6] Astropy Collaboration, Robitaille, T. P., Tollerud, E. J., Greenfield, P., Droettboom, M., Bray, E., Aldcroft, T., Davis, M., Ginsburg, A., Price-Whelan, A. M., Kerzendorf, W. E., Conley, A., Crighton, N., Barbary, K., Muna, D., Ferguson,

H., Grollier, F., Parikh, M. M., Nair, P. H., Unther, H. M., Deil, C., Woillez, J., Conseil, S., Kramer, R., Turner, J. E. H., Singer, L., Fox, R., Weaver, B. A., Zabalza, V., Edwards, Z. I., Azalee Bostroem, K., Burke, D. J., Casey, A. R., Crawford, S. M., Dencheva, N., Ely, J., Jenness, T., Labrie, K., Lim, P. L., Pierfederici, F., Pontzen, A., Ptak, A., Refsdal, B., Servillat, M., and Streicher, O., "Astropy: A community Python package for astronomy," *Astron. & Astroph.* **558**, A33 (Oct. 2013).

[7] Astropy Collaboration, Price-Whelan, A. M., SipHocz, B. M., G"unther, H. M., Lim, P. L., Crawford, S. M., Conseil, S., Shupe, D. L., Craig, M. W., Dencheva, N., Ginsburg, A., Vand erPlas, J. T., Bradley, L. D., P'erez-Su'arez, D., de Val-Borro, M., Aldcroft, T. L., Cruz, K. L., Robitaille, T. P., Tollerud, E. J., Ardelean, C., Babej, T., Bach, Y. P., Bachetti, M., Bakanov, A. V., Bamford, S. P., Barentsen, G., Barmby, P., Baumbach, A., Berry, K. L., Biscani, F., Boquien, M., Bostroem, K. A., Bouma, L. G., Brammer, G. B., Bray, E. M., Breytenbach, H., Buddelmeijer, H., Burke, D. J., Calderone, G., Cano Rodr'iguez, J. L., Cara, M., Cardoso, J. V. M., Cheedella, S., Copin, Y., Corrales, L., Crichton, D., D'Avella, D., Deil, C., Depagne, E., Dietrich, J. P., Donath, A., Droettboom, M., Earl, N., Erben, T., Fabbro, S., Ferreira, L. A., Finethy, T., Fox, R. T., Garrison, L. H., Gibbons, S. L. J., Goldstein, D. A., Gommers, R., Greco, J. P., Greenfield, P., Groener, A. M., Grollier, F., Hagen, A., Hirst, P., Homeier, D., Horton, A. J., Hosseinzadeh, G., Hu, L., Hunkeler, J. S., Ivezi'c, Z., Jain, A., Jenness, T., Kanarek, G., Kendrew, S., Kern, N. S., Kerzendorf, W. E., Khvalko, A., King, J., Kirkby, D., Kulkarni, A. M., Kumar, A., Lee, A., Lenz, D., Littlefair, S. P., Ma, Z., Macleod, D. M., Mastropietro, M., McCully, C., Montagnac, S., Morris, B. M., Mueller, M., Mumford, S. J., Muna, D., Murphy, N. A., Nelson, S., Nguyen, G. H., Ninan, J. P., N"othe, M., Ogaz, S., Oh, S., Parejko, J. K., Parley, N., Pascual, S., Patil, R., Patil, A. A., Plunkett, A. L., Prochaska, J. X., Rastogi, T., Reddy Janga, V., Sabater, J., Sakurikar, P., Seifert, M., Sherbert, L. E., Sherwood-Taylor, H., Shih, A. Y., Sick, J., Silbiger, M. T., Singanamalla, S., Singer, L. P., Sladen, P. H., Sooley, K. A., Sornarajah, S., Streicher, O., Teuben, P., Thomas, S. W., Tremblay, G. R., Turner, J. E. H., Terr'on, V., van Kerkwijk, M. H., de la Vega, A., Watkins, L. L., Weaver, B. A., Whitmore, J. B., Woillez, J., Zabalza, V., and Astropy Contributors, "The Astropy Project: Building an Open-science Project and Status of the v2.0 Core Package," *Astron. Journ.* **156**, 123 (Sept. 2018).

[8] STScI Development Team, "synphot: Synthetic photometry using Astropy," (Nov. 2018).