Introduction_to_Scopesim_for_METIS

March 8, 2022

This notebook presents a basic description of the simulation procedure with scopesim and tries to give some useful hints how the behaviour of each step can be inspected and controlled. Once scopesim and the necessary instrument packages are installed, a simulation is basically a four-step process using the commands `UserCommands`, `OpticalTrain`, `observe` and `readout`. The cells in this notebook are not executable; the practical application to a variety of different source objects and instrument modes is presented in other notebooks.

1 Installation and prerequisites

The easiest way to install Scopesim is to use `pip` from the command line:

```
pip install scopesim
```

If you want to upgrade an existing installation, do

```
pip install -U scopesim
```

In your favourite python environment (e.g. `ipython`, `jupyter notebook` or a script), scopesim is loaded by

```
import scopesim
```

or

```
import scopesim as sim
```

Scopesim has a utility function `sim.bug_report()` which checks a number of other python packages required by scopesim and reports their version numbers. The output of this command should be included in any bug report or question submitted to the scopesim team.

2 Instrument packages

Scopesim is a general simulation framework. In order to simulate observations with any particular instrument, you will have to load an instrument package, as well as packages describing the telescope and observatory location (the latter is important for the atmospheric conditions). In the case of METIS, the packages can be downloaded with

```
sim.download_package(['instruments/METIS', 'telescopes/ELT', 'locations/Armazones'])
```

By default, scopesim looks for the packages in the subdirectory `inst_pkgs` of the current working directory. The download command will install them there, so there should be no problem.

It is possible to install the packages elsewhere; if you do that you will have to declare to scopesim where they are. This can be done by setting

```
sim.rc.__config__["!SIM.file.local_packages_path"] = "/path/to/inst_pgks"
```

If you encounter the error `File cannot be found: default.yaml` when calling `sim.UserCommands` (see below), either call `sim.download_package` from the working directory or set the `local_packages_path` as just explained.

We suggest following the default scheme, because keeping the instrument packages together with the input and output data of scopesim makes it easier later to reconstruct the conditions under which a simulation was run.

The command `sim.list_packages()` lists all packages that are available for download, as well as those that are already installed.

3 Setting up the instrument

The instrument is configured using the `sim.UserCommands()` class. A basic setup for METIS could be

```
cmd = sim.UserCommands(use_instrument='METIS', set_modes=['img_lm'])
```

This creates a (nested dictionary) containing all parameters that are relevant for the description of the instrument and observational conditions, and the way that the simulation is to be run.

The modes are predefined in the instrument package (in the file `METIS/default.yaml`). The available modes for METIS are

- `img_lm` for imaging using the LM imager
- `img_n` for imaging using the N imager
- `lss_l` for long-slit spectroscopy in the L band
- `lss_m` for long-slit spectroscopy in the M band
- `lss_n` for long-slit spectroscopy in the N band

The LMS modes `lms` and `lms_extended` are not yet functional.

The `UserCommands` dictionary is structured into a number of sections that can be accessed using a "bang string", starting with an exclamation mark. For instance, the list of "observation" parameters is obtained with `cmd['!OBS']`. Individual parameters are addressed as `cmd['!OBS.filter_name']` etc. The `!OBS` class contains the parameters that users are most likely to want to change from one simulation to the next. The other sections are more static, and users should only change them with more caution. The following sections are available:

- `!OBS`: parameters that might be changed from one simulation to the other
- `!SIM`: parameters that set up file paths and simulation parameters for scopesim
- `!ATMO`: parameters that relate to the atmosphere and atmospheric conditions.
- `!TEL`: parameters related to the telescope. Of interest for METIS is the parameter `!TEL.ter_curve`, which defaults to `TER_ELT_6_mirror_field_track.dat` but could be changed to `TER_ELT_6_mirror_pupil_track.dat`. The files differ in that field tracking includes thermal emission from the telescope support spiders, which are assumed to be masked in pupil tracking.
- `!INST`: parameters related to the instrument

- `!DET`: parameters related to the detector

The parameters can be changed at this point, for instance to change the pupil transmission (this refers to undersizing of the aperture by inserting a cold stop):

```
cmd['!OBS.pupil_transmission'] = 0.9
```

Individual parameters can be set immediately in the `UserCommands` by giving a `properties` dictionary. This gives the compact form

```
cmd = sim.UserCommands(use_instrument='METIS', set_modes=['lss_m'],
                       properties={'!OBS.slit': 'D-57_1',
                                   '!OBS.detector_readout_mode': 'slow'})
```

Some parameters can also be set later as will be demonstrated.

4 Building and modifying the instrument

The combination of atmosphere, telescope and instrument is represented in scopesim as an `OpticalTrain` object, which is instantiated as

```
metis = sim.OpticalTrain(cmd)
```

The optical train consists of a number of `Effect` objects, which can be listed as

```
metis.effects
```

The table has four columns of which `name` and `included` are important. An effect is addressed by its name; for example, `metis['detector_linearity']` is the effect that describes the (non-)linearity of the detector. Each effect has a `meta` dictionary that contains parameters used to set it up, as well as meta data from configuration files. To resolve a parameter that contains a bang string, the function `from_currsys` has to be used. For instance, `metis['dark_current']` returns `!DET.dark_current`, which is resolved by

```
sim.utils.from_currsys(metis['dark_current'])
```

into a number of electrons per second.

Some parameters support changing parameters, these are notably `filter_wheel` and `slit_wheel`. These have a number of predefined options that can be seen with

```
metis['filter_wheel'].filters
```

The current setting is found by `metis['filter_wheel'].current_filter` and can be changed by

```
metis['filter_wheel'].change_filter('PAH_3.3')
```

5 Defining and observing the source

The `observe()` method of an `OpticalTrain` transmits a source object through the atmosphere, telescope and instrument into the detector plane but does not yet include the detector itself (with the exception of the quantum efficiency, which is treated as a transmission effect). The result is an ideal noise-free image in units of electrons per second (electrons because of the inclusion of QE).

The definition of `Source` objects is described in other notebooks. The observation is done by

```
src = sim.source.source_templates.star()
metis.observe(src)
```

The same optical train can be used to observe multiple sources in succession. In this case it is advisable (and should never harm) to include the parameter `update=True`:

```
metis.observe(src, update=True)
```

Sometimes one might have several optical trains to observe the same source, e.g `metis_l` and `metis_n` for observation in the L and M bands, respectively. To switch from one to the other it is necessary to call the method `set_focus()`, as in

```
metis_l.observe(src)
metis_n.set_focus()
metis_n.observe(src)
```

The noise-free image is an `ImagePlane` object, which can be accessed as

```
metis.image_planes[0]
```

In general, `image_planes` is a list, although METIS always produces a single `ImagePlane`. An `ImagePlane` is essentially a FITS HDU whose parts are accessed as

```
metis.image_planes[0].header
metis.image_planes[0].data
```

6 Creating detector images

The `readout()` method applies photon noise and detector noise to the image and creates detector images with a given exposure time. The result of `readout()` is a list of FITS HDULists (essentially FITS "files" in memory). For METIS only a single HDULIST is created, so a command like

```
result = metis.readout()[0]
```

is convenient. The detector image is in the first extension of `result` and one might want to look at

```
result[0].header
result[1].header
result[1].data
```

By default, the exposure time is set by the `UserCommands` parameter `!OBS.exptime`. In many cases it is more convenient to set it as a parameter to the `readout()` method:

```
result = metis.readout(exptime=3600)[0]     # exposure time in seconds
```

The exposure time is automatically split into `NDIT` subexposures of length `DIT`, ensuring that the detector is not saturated during a subexposure (a warning is issued if a bright source saturates the detector even in the minimum possible `DIT`).

The METIS detectors each have two settings ("fast" and "slow" for the HAWAII detectors, "high_capacity" and "low_capacity" for the Geosnap). There is a default setting for each instrument mode, but an optimal readout mode can also be automatically determined by

```
result = metis.readout(detector_readout_mode='auto')
```

The result can be written to disk with

```
result.writeto("simulation_result.fits", overwrite=True)    # careful with overwrite
```

for analysis with external tools.