Soapy Documentation

Release v0.13.1-214-g093ce9c-dirty

Andrew Reeves

Jul 20, 2017

Contents

Contents:

CHAPTER 1

Introduction

Soapy is a Montecarlo Adaptive Optics (AO) simulation written exclusively in the Python programming language. It is aimed at rapidly developing new AO concepts and being a learning tool for those new to the field of AO.

The code can be used as an end to end simulation, where the entire system parameters are controlled by a configuration file. This can be used from the Python command line, python scripts or a GUI which is included, operation is described in the *Basic Usage* section.

The codes real strength lies in its modular nature. Each AO component is modelled as a Python object, with intuitive methods and attributes. Components can be imported and used separately to create novel AO configurations. Starting with the main *Simulation* module, these are described in detail in this documentation.

Quick-Start

Try out some of the code examples in the *conf* directory, either run the *soapy* script in *bin*, or load a python or IPython terminal:

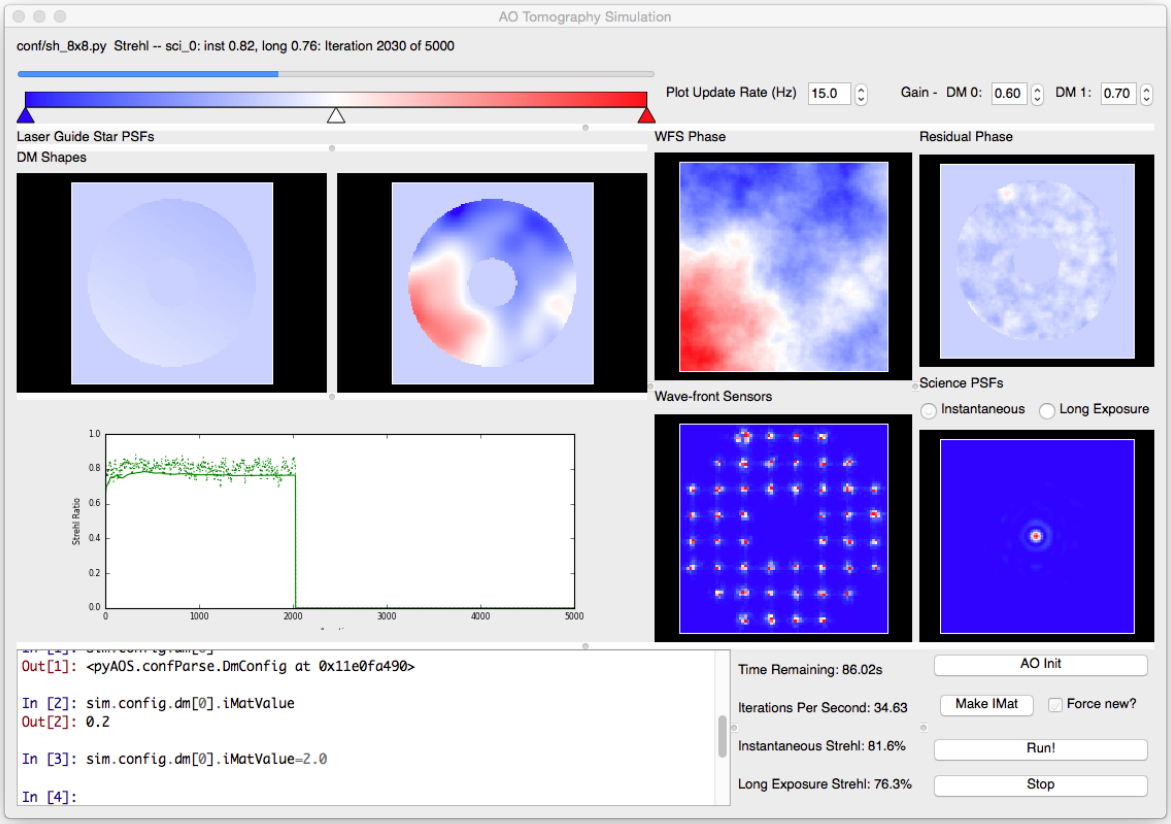
```
import soapy
sim = soapy.Sim("configFilename")
sim.aoinit()
sim.makeIMat()
sim.aoloop()
```

Data will now be saved in the directory specified as *filePrefix* in the configuration file.

Alternatively, the GUI can be started with:

```
soapy -g <configFilename>
```

The use the buttons to initialise the simulation, make interaction matrices and run the AO loop. The interactive python console can be used to view data or change parameters


AO Tomography Simulation
conf/sh_8x8.py Strehl -- sci_0: inst 0.82, long 0.76: Iteration 2030 of 5000
Plot Update Rate (Hz) 15.0
Gain - DM 0: 0.60
DM 1: 0.70
Laser Guide Star PSFs
DM Shapes
WFS Phase
Residual Phase
Wave-front Sensors
Science PSFs
Instantaneous
Long Exposure
Strehl Ratio
Out[1]: <pyAOS.confParse.DmConfig at 0x11e0fa490>
In [2]: sim.config.dm[0].iMatValue
Out[2]: 0.2
In [3]: sim.config.dm[0].iMatValue=2.0
In [4]:
Time Remaining: 86.02s
AO Init
Iterations Per Second: 34.63
Make IMat
Force new?
Instantaneous Strehl: 81.6%
Run!
Long Exposure Strehl: 76.3%
Stop

CHAPTER 2

Installation

Firstly, you'll need Python. This comes with pretty much every linux distribution and is also installed by default on Mac OS X. For Windows, I'd recommend using a Python distribution, such as anaconda or enthought canopy. The code should be compatible with Windows as Python and all the libraries used are platform independent. Nonetheless, I only use it on Mac OSX and Linux so I can't guarantee that there won't be bugs not present on other OSs.

Installation

Once all the requirements outlined below are met, you are ready to install Soapy. Download the source from github, either as a zip file, or clone the git repository with:

```
git clone https://github.com/soapy/soapy.git
```

If downloading the code as a zip, you can choose which version to use with the drop down box on the left of the page, entitled `branch:master`. Whilst I try not to, the master branch will occasionally be broken so you might want to get the latest stable version by clicking "tags" in the dropdown list, and selecting the most recent version number.

Once the code is downloaded (and unzipped) or cloned, navigate to the resulting directory using the command line. You can import it into python straight away from this directory. To use the `soapy` script, run:

```
python soapy <options> <configfile>
```

If you wish to have it available elsewhere on your system, either set the relavant `PATH` and `PYTHONPATH` variables to `<soapy dir>/bin` and `<soapy dir>/` respectively, or run the install script with:

```
python setup.py install
```

This latter method may require superuser permissions for your system and should setup the paths for you. You should now be able to run `soapy` and import soapy into python from any directory on your system.

Required Libraries

Soapy doesn't have too many requirements in terms of external libraries, though it does rely on some. Performance of the simulation is made reasonable (for ELT scale operation) by using pyfftw and the numba library. Pyfftw simply wraps the FFTW library for fast fourier transforms. Numba, is a clever library that leverages the LLVM compiler infrastructure to compile python code directly to machine code. A library of functions has been written for the most computationally challenging algorithms, which are in pure python so can be easily read and improved, but operate quickly with the option of using multiple threads. There are also some optional libraries which are recommended for plotting.

Required

```
numpy
scipy
astropy
pyfftw
numba
yaml
```

For GUI

```
PyQt5 (PyQt4 supported)
matplotlib
ipython
```

Linux

If your starting with python from scratch, there a couple of options. For Ubuntu (14.04+) linux users, all these packages can be installed via apt-get:

```
sudo apt-get install python-numpy python-scipy python-fftw python-astropy python-qt4␣
↪python-matplotlib ipython ipython-qtconsole python-yaml python-numba
```

for Red-hat based systems these packages should also be available from repositories, though I'm not sure of they're names.

Mac OSX

for mac os, all of these packages can be install via macports, with:

```
sudo port install python36 py36-numpy py36-scipy py36-astropy py36-pyqt5 py36-ipython␣
↪py36-jupyter py36-numba py36-yaml py36-qtconsole
```

pyfftw is not available for python3.6 on macports, so must be installed with another method, such as pip (see below)

If you're using Python 2.7:

```
sudo port install python27 py27-numpy py27-scipy py27-astropy py27-pyfftw py27-pyqt5␣
↪py27-ipython py27-jupyter py27-numba py27-qtconsole py27-yaml
```

Any OS

Anaconda Python

For any OS, including Windows, python distributions exist which include lots of python packages useful for science. A couple of good examples are Enthought Canopy (https://www.enthought.com), which is free for academics, and Anaconda (https://store.continuum.io/cshop/anaconda/) which is also free. Anaconda includes most of the required libraries by default apart from pyfftw and pyyaml. These can be installed with:

```
conda install pyyaml
pip install pyfftw
```

pip

A lot of python packages are also listed on pypi. Usually when python is installed, a script called `easy_install` is installed also, which can be used to get any package on pypi with `easy_install <package>`. Confusingly, `pip` is now the recommended Python package manager instead of `easy_install`. If you've only got `easy_install` you can install `pip` using `easy_install pip`, or it can be installed using the script linked here.

Once you have `pip`, the required libraries can be installed by using the `requirements.txt` file. From the soapy directory, just run (may need to be as `sudo`):

```
pip install numpy scipy astropy pyfftw pyyaml numba
```

and all the requirements should be installed for the simulation, though not the GUI. For the GUI PyQt4 or PyQt5 is required, I dont think these are available from pip.

Sometimes pyfftw has a hard time finding your installation of fftw to link against. On a Mac, these lines usually help before running the pip command:

```
export DYLIB_LIBRARY_PATH=$DYLIB_LIBRARY_PATH:<path/to/fftw>/lib
export LDFLAGS=-L<path/to/fftw>/lib
export CFLAGS=-I<path/to/fftw>/include/
```

export CFLAGS=-I<path/to/fftw>/include/

Testing

Once you think everything is installed, tests can be run by navigating to the `test` directory and running:

```
python testSimulation.py
```

Currently, this only runs system wide tests, but further, more atomic tests will be added in future. To run the tests, soapy must be either "installed", or manually put into the PYTHONPATH.

CHAPTER 3

Basic Usage

This section describes how to the simulation for basic cases, that is, using the full end to end code to create and save data which can then be analysed afterwards. Such a scenario is a common one when exploring parameters on conventional AO systems.

Configuration

In Soapy, all AO parameters are controlled from the configuration file. This is a python script which contains all the information required to run many AO configurations. A few examples are provided in the `conf` directory when you download the code. All parameters are held in one large dictionary, titled `simConfiguration`, and are then grouped into relavent sections.

`Sim` parameters control simulation wide parameters, such as the filename to save data, the number of simulated phase points, the number of WFSs, DMs and Science cameras as well as the name of the reconstructor used to tie them together. The `simName` parameter specifies a directory, which will be created if it does not already exist, where all AO run data will be recorderd. Each run will create a new time-stamped directory within the parent `simName` one to save run specific data. Data applying to all runs, such as the interaction and control matrices are stored in the `simName` directory.

`Atmosphere` parameters are responsible for the structure of the simulated atmosphere. This includes the number of simulated turbulence layers and the integrated seeing strength, r_0. Some values in the Atmosphere group must be formatted as a list or array, as they describe parameters which apply to different turbulence layers.

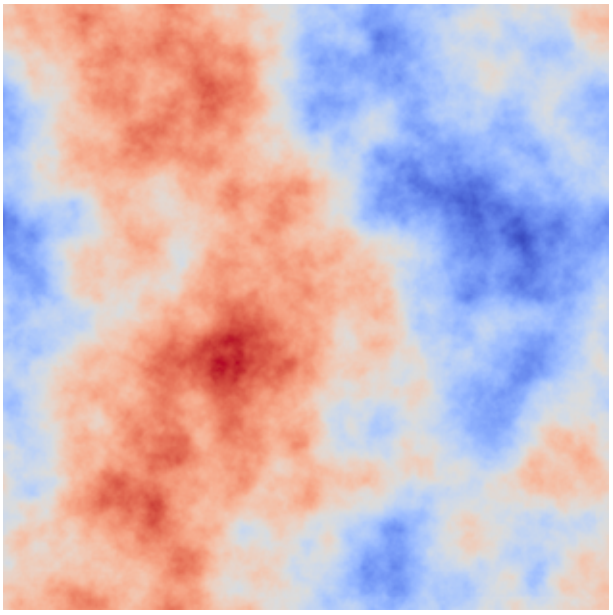
Parameters describing the physical telescope are given in the `Telescope` group. These include the telescope and central obscuration diameters, and a pupil mask.

WFSs, LGSs, DMs and Science camera are configured by the `WFS`, `LGS`, `DM` and `Science` parameter groups. As multiple instances of each of these components may be present, every parameters in these groups is represented by either a list or numpy array, where each element specifies that component number. For WFSs and DMs, a `type` parameter is also given. This is a the name of the python object which will be used to represent that component, and a class of the same name must be present in the `WFS.py` or `DM.py` module, respectively. Other WFS or DM parameters may then have different behaviours depending on the type which is to be used.

Each parameter that can be set is described in the *Configuration* section.

Creating Phase Screens

For most applications of Soapy, some randomly generated phase screens are required. These can either be created just before the simulation begins, during the initialisation phase, or some existing screens can be specified for the simulation to use. To generate new phase screens with the parameters specified in `Atmosphere` each time the simulation is run, set the `Atmosphere` parameter, `newScreens` to `True`.



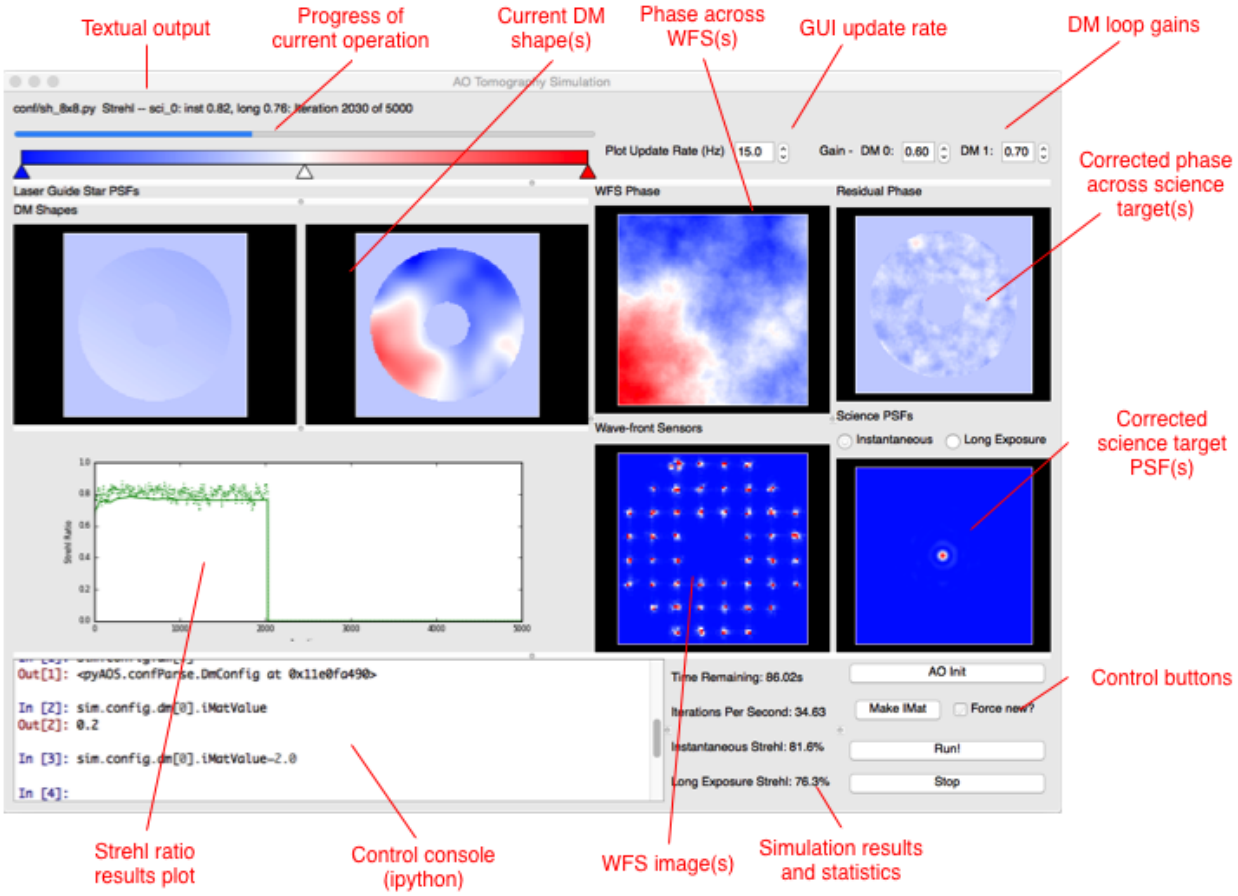
If instead you wish to used existing phase screens, provide the path to, and filename of each screen in the `screenNames` parameter as a list. Screens specified to be loaded must be saved as FITS files, where each file contains a single, 2 dimensional phase screen. The simulation will largely trust that the screen parameters are valid, so other parameters in the `Atmosphere` group, such as the `wholeScreenSize`, `r0` and `L0` may be discounted. If you would like the simulation to be able to scale your phase screens such that they adhere to the `r0` and `screenStrength` values set in the configuration file, then the FITS file header must contain a parameter `R0` which is expressed in units of phase pixels.

Running the Simulation

Once all the configuration parameters have been set, and you have decided how whether to load or generate phase screens, the simulation is ready to be run. This can be either from the GUI, the command line or from a script.

Graphical User Interface

When running Soapy configurations for the first time it can be a good idea to run them in the GUI to sure that components look to be operating as expected. The GUI is shown below running a simple SCAO case, with a tip-tilt mirror and a stack array DM.



If soapy has been installed, or the `bin` directory is in the bash PATH, the GUI is started from the command line with the command:

```
soapy -g path/to/configFile.yaml
```

The `soapy` script can do a few other things as well, use `soapy --help` to see all other available options.

Once the GUI has loaded it will begin the initialisation of the simulation. This stage initialises all the simulated components, loads or generates phase screens, allocates data buffers and calculates various required parameters from the parameters given in the configuration file. If any parameters or the configuration file is changed at any point, this initialisation step can be rerun by clicking the "AO Init" button.

The next step in most systems will be to record an interaction matrix, where the effect of each DM influence on the WFS(s) is recorded, and used to calculate a command matrix. From the GUI, this is achieved by clicking the "makeIMat" button. Interaction matrices, command matrices and DM influence functions can be saved in the `simName` directory and the simulation checks to see if there are valid ones in that directory it can load instead of making them again. If you would like to force a new interaction matrix to be made, perhaps because you've changed parameters which may effect the new interaction matrix, tick the "Force new?" box.

Once this is complete, you can now click "Run!" to run the simulation. You will now see the atmospheric phase moving across the WFS(s), and the resulting measurements on the WFS. This will be recorded, and transformed to DM commands measurements via the reconstructor, and finally, the science phase will be corrected and a better PSF achieved. The loop gain for each DM can be altered using the spin boxes in the top right of the GUI.

Using the GUI significantly slows down the simulation operation, but this can be aleviated by limiting the simulation update rate using the top spin box.

The console in the bottom left of the GUI can be used to either change parameters of the simulation or visualise other data sources. It is a complete python console, provided by the IPython library. To load a new config file into the GUI, go the file>Load Configuration File. You will then have to click "AO Init" to begin initialisation.

Command Line and Scripting

To run the simulation from the command line, either use

```
soapy -i /path/to/configFile.yaml
```

which will initialise the simulation before dropping you into an interaction ipython prompt, or simply start or python interpretter of choice and run

```
import soapy                                    #Imports python library
sim = soapy.Sim("/path/to/configFile.yaml")   #Loads the configuration file
sim.aoinit()                                   #Initialises all AO simulated objects
```

The above code would also be used in scripts to run the simulation.

To measure the interaction matrix run:

```
sim.makeIMat()
```

or:

```
sim.makeIMat(forceNew=True)
```

if you'd like to force the creation of interaction matrices, command matrices and DM influence functions.

Once complete, you're now ready to run the simulation with:

```
sim.aoloop()
```

You should now see a rolling counter of the frame number and current Strehl ratio of each science target.

Retrieving Simulation Data

After a simulation run has completed, the resulting data must be retrieved for analysis. The data stored by Soapy depends on the parameters set in the `sim` group in the configuration file. Once a `aoloop` has completed, the data will be saved into the `simName` directory, in a further, time-stamped directory for that particular run. Whithin the simulation, the data is stored in numpy array structures which can be accessed either after the run has completed or during the run (if it is run in the, or in a python thread on the command line).

The strehl ratio of each science target is always stored. Internally, it is kept in the arrays:

```
sim.instStrehl
```

and:

```
sim.longStrehl
```

Which are the instantaneous and long exposure strehl ratio for each science target. Each of these is of shape `sim.config.sim.nSci` by `sim.config.sim.nIters`. Note that this is even the case for only a single science target, when the science target Strehl ratios are always accessed with `sim.longStrehl[0]`. Strehl ratios may also saved in the `simName` directory as `instStrehl.fits` and `longStrehl.fits`.

There are many other data sources available to save or access from the simulation, these are listed in *Data Sources*.

CHAPTER 4

Simple Tutorial

This tutorial will go through some example AO systems using Soapy. We'll see how to make configuration files to run the AO system that you'd like to, then extract data which can be subsequently analysed. CANARY is an AO system on the 4.2m William Herschel Telescope on La Palma. It is designed to be very flexible to run various "modes" of AO, so makes a nice test bed for us to simulate. We'll simulate it in SCAO mode, in GLAO with multiple guide-stars and in SCAO with a LGS.

Running an Example SCAO Configuration

Before making new configuration files though, its a pretty good idea to make sure everything is working as expected by running one of the examples. First, lets create a directory where we do this tutorial, call it something like `soapy_tutorial`, make a further directory called `conf` inside and copy the example configuration file `sh_8x8.yaml` form the downloaded or cloned Soapy directory into it.

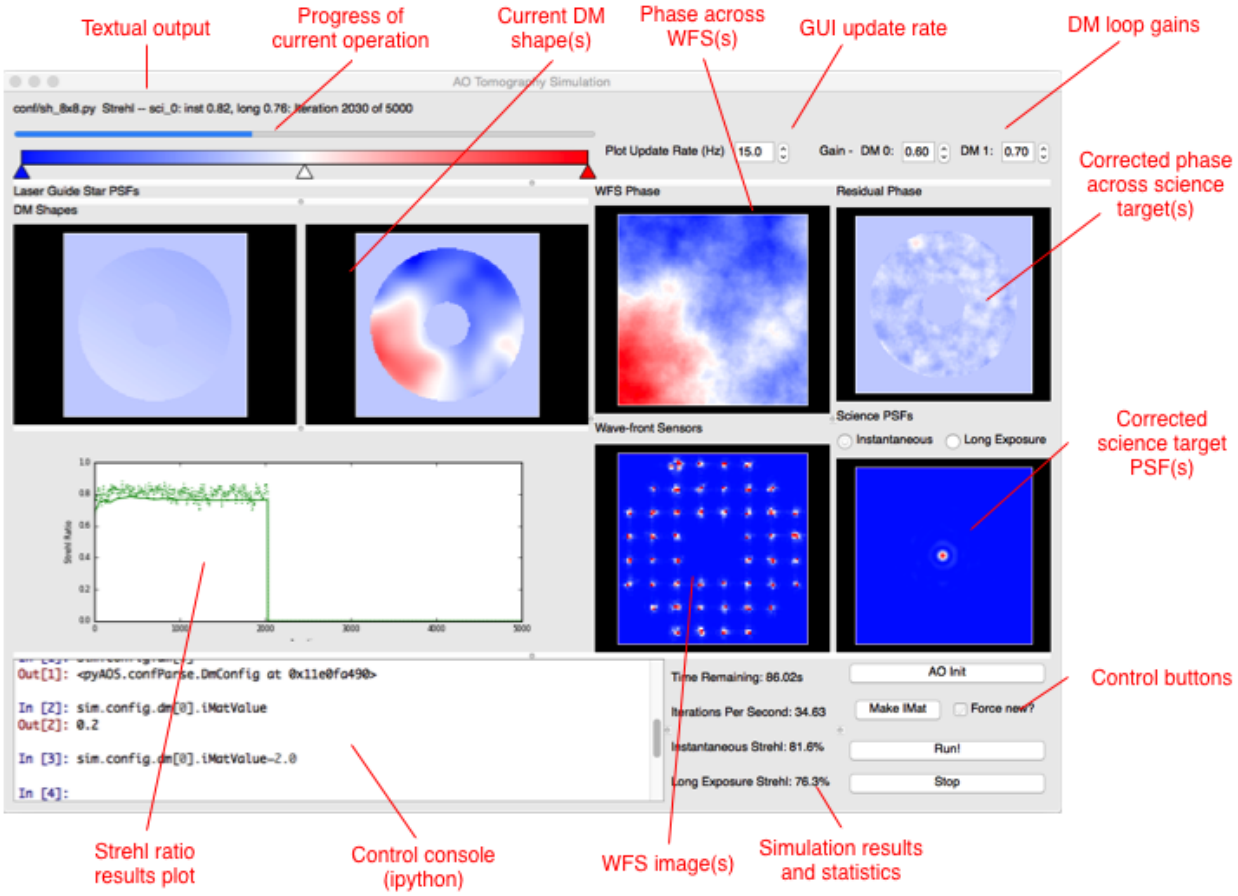
To open the Graphical User Interface (GUI), type in the command line:

```
soapy --gui conf/sh_8x8.yaml
```

This relies on `soapy` being in you're `PATH`. If thats not the case, run:

```
python <path/to/soapy>/bin/soapy --gui conf/sh8x8.yaml
```

You should see a window which looks a bit like this pop up:



If you don't want to run the GUI, then open a python terminal and run:

```
import soapy
sim = soapy.Sim("conf/sh8x8.yaml")
```

Before the simulation can be started, some initialisation routines must be run. If running the GUI, then this will automatically when you start it up. In the command line, to initialise run:

```
sim.aoinit()
```

Next, the interaction matrixes between the DMs and the WFSs. In the GUI this is achieved by clicking "makIMat", and in the command line with:

```
sim.makeIMat()
```

This simulation will save command matrices, interaction matrices and DM influence functions for a simulation, so that it doesn't alway have to remake them. If you'd like to override the loading them from file and make them from scratch, tick the "force new" button in the GUI, or pass the argument `forceNew=True` to the `makeIMat` command.

To actually run the simulation, click "aoloop" in the GUI, or type:

```
sim.aoloop()
```

at the command line. This will run the simulation for the configured number of iterations, and estimate the performance of the specified AO system.

Creating a new SCAO configuration file

Now the simulation is working, lets start to simulate CANARY. We'll use the `sh_8x8.yaml` configuration file as a template. Copy it to another file called `CANARY_SCAO.yaml`, and open this file in your favourite text editor. The configuration file contains all the parameters which determine the configuration of the simulated AO system. All the parameters are held in a YAML configuration file and parameters are grouped into sub-dictionaries depending on which components they control. Descriptions of all possible parameters are given in the *Configuration* section.

`Sim` Parameters

The first of these groups are parameters are those which have a system wide effect, so-called `Sim` parameters. They should have no indendation in the YAML file.

The first parameter to change is the `simName`, this is the directory where data will be saved during and after an AO run. Set it to `CANARY_SCAO`. The `logFile` is the filename of a log which records all text output from the simulation, set it to `CANARY_SCAO.log`. The value of `loopTime` specifies the frame rate of the simulation, which is usually, though not always, also the frame rate of the WFSs and DMs. More accurately though, it is the time between movements of the atmosphere. For CANARY, make the system run at 200Hz, so set this to `0.005`. For the purposes of this tutorial, lets also set the number of iterations which will be run, `nIters` to around 500 so that it will run quickly.

The `Sim` group also contains parameters which determine the data which will be stored and saved from the simulation. Set values to `True` if you'd like them to be continually saved in a memory buffer before being written to disk in a AO run specific, time-stamped directory within the `simName` directory.

`Atmosphere` Parameters

As would be expected, this group of parameters describe the nature of the atmospheric turbulence. Currently, this configuration file features an atmosphere with 4 discrete turbulence layers, increase that to 5 by setting `scrnNo` to `5`. The `r0` parameter is the Fried parameter in metres and controls the integrated seeing strength, set this to `0.14`. `screenHeights`, `scrnStrengths`, `windDirs` and `windSpeed` control the layer heights, relative C_N^2 strengths, wind directions and wind velocities. These must be formatted as a list at least as long as `scrnNo`, so add another value to each.

Phase screens can be either created on each simulation run, or can be loaded from file. To load screens from file a parameter, `scrnNames`, must be set with the filename of each phase screen in a list.

`Telescope` Parameters

The diameter of the simulated telescope and its central obscuration are determined by the `telDiam` and `obsDiam` parameters in the `Telescope` parameters. The `mask` value determines the shape if the pupil mask. If set to `circle`, this will simple be a circular telescope pupil, with a circular obscuration cut out the centre. If something more complex is desired, this value should be set to filename of 2-d fits file with shape `(sim.pupilSize, sim.pupilSize)`, set to `0` at opaque parts of the pupil and `1` at transparent parts.

CANARY is hosted by the WHT, which is a 4.2 metre diameter telescope with a central obscuration of approximately 1.2 metres. Set these values, and keep `mask` set to `circle`.

`WFS` Parameters

Each WFS must be specified seperately, with an index or 0, 1, 2...etc. Set `nxSubaps`, the number of Shack-Hartmann sub-apertures in a single dimension to `7` and `pxlsPerSubap` to `14`. The pixel scale is defined by the parameter

`subapFOV`, which is actually the FOV of the entire sub-aperture, set this to `2.5`.

`DM` Parameters

As with `WFS` parameters, each DM is specified seperately, with an integer index. There must be at least `sim.nDM``s specified. The first DM will be a Tip-tilt mirror, hence the ``type` is set to `TT`. The second is a higher spatial order stack array type denoted in the simulation as `Piezo`. These names correspond to classes which are defined in the `DM.py` module. Set the number of actuators in one dimension to 8, by setting the second value in `nxActuators` to `8`.

`Science` Parameters

The final group of parameters which define the simulation are the `Science` parameters which define the science targets and detectors to be used to measure AO performance. Again, multiple science cameras can be specified, so each requires an index. There must be at least `sim.nSci` science cameras specified. Change the Field of View of the science detector by setting `FOV` to `3.0`.

Run it!

Run the simulation as before, either in the GUI or in the command line with either:

```
soapy --gui conf/CANARY_SCAO.yaml
```

click makeIMat click aoloop

or:

```
import soapy
sim = soapy.Sim("conf/CANARY_SCAO.yaml")
sim.makeIMat()
sim.aoloop()
```

The resulting Strehl ratio should be around 0.65, though there will be some variation due to the random generation of the phase screens.

Examining data and changing parameters

Once a simulation has been completed, the task then turns to extracting an analysing the resulting data. Many data sources can be saved from Soapy, they are listed in *Data Sources*. Whether they are saved or not is a result of the parameters set in the `Sim` section. If so, they will be saved to a directory of `<simName>/<timestamp>/` in the FITS standard format. They can also be accessed from the simulation object using `sim.<dataSource>`. For example, to plot the long exposure Strehl ratio recorded on the first science detector over the course of the simulation, type either in a command line or in the GUI terminal:

```
from matplotlib import pyplot
pyplot.plot(sim.longStrehl[0])
pyplot.show()
```

The first science detector image can be retrieved with:

```
imshow(sim.sciImgs[0])
```

and the measurements recored on all WFSs with:

```
imshow(sim.allSlopes)
```

The parameters which were originally defined in the configuration file can also be accessed and altered. The variables holding the parameters have the same name as the configuration file parameters, though the names of the groups may be shortened. Assuming that the simulation object is called `sim` (as in this tutorial), any configuration parameter can be access with:

```
sim.config.<configGroup>.<param>
```

So to check or change the `pupilSize` parameter, one could do the following:

```
print(sim.config.sim.pupilSize)
sim.config.sim.pupilSize = 256
```

For the parameter groups `WFS`, `DM` and `Science`, which are set as lists, access of the parameter for item `n` is through `sim.config.wfss[n].<param>`, `sim.config.dms[n].<param>` and `sim.config.scis[n].<param>`. For example, to check, then change the 1st WFS centroiding method:

```
print(sim.config.wfss[0].centMethod)
sim.config.wfss[0].centMethod = "simple"
```

or to set the number of DM actuators on the high order DM:

```
print(sim.config.dms[1].nxActuators)
sim.config.dms[1].nxActuators[1] = 16
```

After changing these values, click aoinit or type `sim.aoinit`, then makeImat or `sim.makeIMat()` and finally aoloop or `sim.aoloop` to run the simulation and observe the effect of the change parameters. Some parameters can be changed while the simulation is running. This is useful when using the GUI and optimising parameters for an AO system. Parameters which are safe to change during AO operation are denoted in the *Configuration* section with ** at the end of the parameter description.

GLAO Example

CANARY is an experimental AO system which has been designed to explore tomographic AO. As such it would be thoroughly rude not to simulate it in a tomographic configuration. As tomographic AO often involves complex reconstructors out of the scope of this tutorial, it shall be run in the simplest tomographic case, Ground Layer AO (GLAO). This is where the measurements of several WFSs observing off-axis are effectively averaged, which corrects well when the WFS field of views overlap, such as at low-layers, but not so well when they have diverged, such as at high layers. This mode of AO can be performed using the `MVM` reconstructor used previously without modification.

Copy the `CANARY_SCAO` configuration file to another file name `CANARY_GLAO`. The only parameters which require changing are the number and position of WFSs. In the `Simulation` group set `nGS` to `3`. Copy the first WFS set of parameters and paste them below it twice. Change the index, currently set at `0` to `1` and `2` respectively. The `GSPosition` values may be set to an asterism such as `[0, 30]`, `[-24.5, -25]`, `[24.5, -15]` which forms a triangle around the science target.

Run this new configuration file. The AO performance should have decreased significantly as only the lowest turbulence layer will be corrected effectively, but extra off-axis science targets would show that the performance is more consistent across a wide-field.

CHAPTER 5

Configuration

Configuration of the system is handled by the `confParse` module, that reads the simulation parameters from a given configuration file. This file should be a YAML file, which contains groups for each simulation sub-module. Where a sub-module may consist of multiple components i.e. Wave-front sensors, each WFS must be specified seperately, with an integer index, for example:

```
WFS:
        0:
                GSMag: 0
                GSPosition: (0, 0)
        1:
                GSMag: 1
                GSPosition: (1, 0)
```

Example configuration files can be found in the `conf` directory of the soapy package. (Note: Previously, a Python file was used for configuration. This format is still supported but can lead to messy configuration files! There are still examples of these in the source repository if you prefer.)

Below is a list of all possible simulation parameters. Parameters which have a description ending in ** can be altered while the simulation is running. When others are changed and `aoinit` must be run before they will take effect and they may break a running simulation.

Simulation Parameters

class `soapy.confParse.`**`SimConfig`**(*N=None*)

Configuration parameters relavent for the entire simulation. These should be held at the beginning of the parameter file with no indendation.

Required:

Parameter	Description
`pupilSize`	int: Number of phase points across the simulation pupil
`nIters`	int: Number of iteration to run simulation
`loopTime`	float: Time between simulation frames (1/framerate)

Optional:

Parameter	Description	Default
`nGS`	int: Number of Guide Stars and WFS	`0`
`nDM`	int: Number of deformable Mirrors	`0`
`nSci`	int: Number of Science Cameras	`0`
`reconstructor`	str: name of reconstructor class to use. See `reconstructor` module for available reconstructors.	`"MVM"`
`simName`	str: directory name to store simulation data	`None`
`wfsMP`	bool: Each WFS uses its own process	`False`
`verbosity`	int: debug output for the simulation ranging from 0 (no-ouput) to 3 (all debug output)	`2`
`logfile`	str: name of file to store logging data,	`None`
`learnIters`	int: Number of *learn* iterations for Learn & Apply reconstructor	`0`
`learnAtmos`	str: if `random`, then random phase screens used for *learn*	`random`
`simOversize`	float: The fraction to pad the pupil size with to reduce edge effects	`1.2`
`loopDelay`	int: loop delay in integer count of `loopTime`	`0`
`threads`	int: Number of threads to use for multithreaded operations	`1`
`photometric_zp`	float: Photometric zeropoint - number of photons/meter/second from a magnitude 0 star	`2e9`

Data Saving (all default to False):

Parameter	Description
`saveSlopes`	Save all WFS slopes. Accessed from sim with `sim.allSlopes`
`saveDmCommands`	Saves all DM Commands. Accessed from sim with `sim.allDmCom`
`saveWfsFrames`	Saves all WFS pixel data. Saves to disk a after every frame to avoid u much memory
`saveStrehl`	Saves the science camera Strehl Ratio. Accessed from sim with `sim.longStrehl` and `sim.instStrehl`
`saveWfe`	Saves the science camera wave front error. Accessed from sim with s
`saveSciPsf`	Saves the science PSF.
`saveInstPsf`	Saves the instantenous science PSF.
`saveInstScieField`	Saves the instantaneous electric field at focal plane.
`saveSciRes`	Save Science residual phase

Telescope Parameters

class `soapy.confParse.`**`TelConfig`**(*N=None*)

Configuration parameters characterising the Telescope. These should be held in the `Telescope` group in the parameter file.

Required:

Parameter	Description
`telDiam`	float: Diameter of telescope pupil in metres

Optional:

Parameter	Description	Default
`obsDiam`	float: Diameter of central obscuration	`0`
`mask`	str: Shape of pupil (only accepts `circle` currently)	`circle`

Atmosphere Parameters

class `soapy.confParse.`**`AtmosConfig`**(*N=None*)

Configuration parameters characterising the atmosphere. These should be held in the `Atmosphere` group in

the parameter file.

Required:

Parameter	Description
`scrnNo`	int: Number of turbulence layers
`scrnHeights`	list, int: Phase screen heights in metres
`scrnStrength`	list, float: Relative layer scrnStrength
`windDirs`	list, float: Wind directions in degrees.
`windSpeeds`	list, float: Wind velocities in m/s
`r0`	float: integrated seeing strength (metres at 500nm)

Optional:

Parameter	Description	Default
`scrnNames`	list, string: filenames of phase if loading from fits files. If `None` will make new screens.	`None`
`subHarmonics`	bool: Use sub-harmonic screen generation algorithm for better tip-tilt statistics - useful for small phase screens.	`False`
`L0`	list, float: Outer scale of each layer. Kolmogorov turbulence if `None`.	`None`
`randomScrns`	bool: Use a random set of phase phase screens for each loop iteration?	`False`
`infinite`	bool: Use infinite phase screens?	`False`
`tau0`	float: Turbulence coherence time, if set wind speeds are scaled.	`None`
`wholeScrnSize`	int: Size of the phase screens to store in the `atmosphere` object. Required if large screens used.	`None`

Wave-front Sensor Parameters

class `soapy.confParse.`**`WfsConfig`**(*N=None*)

Configuration parameters characterising Wave-front Sensors. These should be held in the `WFS` group in the parameter file. Each WFS is specified by first specifying an index, then the WFS parameters. Any entries above `sim.nGS` will be ignored.

Required:

Parameter	Description
`GSPosition`	tuple: position of GS on-sky in arc-secs
`wavelength`	float: wavelength of GS light in metres
`nxSubaps`	int: number of SH sub-apertures

Optional:

Parameter	Description	Default
`type`	string: Which WFS object to load from WFS.py?	`ShackHartmann`
`GSMag`	float: Apparent magnitude of the guide star	`0`
`photonNoise`	bool: Include photon (shot) noise.	`False`
`eReadNoise`	float: Electrons of read noise	`0`
`throughput`	float: Throughput of the entire optical and electronic system from guide star photons to recorded WFS detector counts. Includes atmospheric effects, the optical train and detector gain.	`1.`
`propagationMode`	string: Mode of light propogation from GS. Can be "Physical" or "Geometric"**.	`"Geometric"`
`subapFieldStop`	bool: if True, add a field stop to the wfs to prevent spots wandering into adjacent sub-apertures. if False, oversample subap FOV by a factor of 2 to allow into adjacent subaps.	`False`
`removeTT`	bool: if True, remove TT signal from WFS slopes before reconstruction.**	`False`
`fftOversamp`	int: Multiplied by the number of of phase points required for FOV to increase fidelity from FFT.	`3`
`GSHeight`	float: Height of GS beacon. `0` if at infinity.	`0`
`subapThreshold`	float: How full should subap be to be used for wavefront sensing?	`0.5`
`lgs`	bool: is WFS an LGS?	`False`
`centMethod`	string: Method used for Centroiding. Can be `centreOfGravity`, `brightestPxl`, or `correlation`.**	`centreOfGravity`
`referenceImage`	array: Reference images used in the correlation centroider. Full image plane image, each subap has a separate reference image	`None`
`angleEquivNoise`	float: width of gaussian noise added to slopes measurements in arc-secs	`0`
`centThreshold`	float: Centroiding threshold as a fraction of the max subap value.**	`0.1`
`exposureTime`	float: Exposure time of the WFS camera - must be higher than loopTime. If None, will be set to loopTime.	`None`
`wvlBandWidth`	float: Width of wavelength band sent to WFS in nm	`100`
`extendedObject`	ndarray or str: The object used as extended source for WFS, of size 2*fftOversamp*pxlsPerSubap. The FOV of the object should be twice the FOV of the sub-aperture.	`None`
`fftwThreads`	int: number of threads for fftw to use. If `0`, will use system processor number.	`1`
`fftwFlag`	str: Flag to pass to FFTW when preparing plan.	`FFTW_PATIENT`
`pxlsPerSubap`	int: number of pixels per sub-apertures	`10`
`subapFOV`	float: Field of View of sub-aperture in arc-secs	`5`
`correlationPadding`	int: Padding for correlation WFS	`None`
`nx_guard_pixels`	int: Guard Pixels between Shack-Hartmann sub-apertures (Not currently operational)	`0`

Laser Guide Star Parameters

class `soapy.confParse.`**`LgsConfig`**(*N=None*)

Configuration parameters characterising the Laser Guide Stars. These should be held in the `LGS` sub-group of the WFS parameter group.

	Parameter	Description	Default
	uplink	bool: Include LGS uplink effects	False
	pupilDiam	float: Diameter of LGS launch aperture in metres.	0.3
	wavelength	float: Wavelength of laser beam in metres	600e-9
	propagationMode	str: Mode of light propogation from GS. Can be "Physical" or "Geometric".	"Phsyical"
	height	float: Height to use physical propogation of LGS (does not effect cone-effect) in metres	90000
Optional:	elongationDepth	float: Depth of LGS elongation in metres	0
	elongationLayers	int: Number of layers to simulate for elongation.	10
	launchPosition	tuple: The launch position of the LGS in units of the pupil radii, where (0,0) is the centre launched case, and (1,0) is side-launched.	(0,0)
	fftwThreads	int: number of threads for fftw to use. If 0, will use system processor number.	1
	fftwFlag	str: Flag to pass to FFTW when preparing plan.	FFTW_PATIENT
	naProfile	list: The relative sodium layer strength for each elongation layer. If None, all equal.	None

Deformable Mirror Parameters

class soapy.confParse.**DmConfig**(*N=None*)

Configuration parameters characterising Deformable Mirrors. These should be held in the DM sub-group of the parameter file. Each DM is specified seperately, by first specifying an index, then the DM parameters. Any entries above sim.nGS will be ignored.

	Parameter	Description
	type	string: Type of DM. This must the name of a class in the DM module.
Required:	nxActuators	int: Number independent DM shapes. e.g., for stack-array DMs this is number of actuators in one dimension, for Zernike DMs this is number of Zernike modes.
	gain	float: The loop gain for the DM.**
	svdConditioning	float: The conditioning parameter used in the pseudo inverse of the interaction matrix. This is performed by numpy.linalg.pinv.

Optional:

Reconstructor Parameters

class soapy.confParse.**ReconstructorConfig**(*N=None*)

Configuration parameters describing the reconstructor that will be used to calculate DM commands from WFS measurements. The type must be an object in the soapy.reconstruction module. Other parameters may be specific to this reconstructor

	Parameter	Description	Default
	type	string: Type of reconstructor to use. Must be a class in reconstruction module.	MVM
Optional:	svdConditioning	float: Conditioning parameter to be using in Least Squares reconstructor inversion SVD to cut off unwanted DM modes. See numpy.linalg.pinv for details about the inversion.	0
	gain	float: Gain of the integrator loop.	0.6
	imat_noise	bool: include WFS noise when making in interaction matrix	True

Science Camera Parameters

class `soapy.confParse.`**`SciConfig`**(*N=None*)

Configuration parameters characterising Science Cameras.

These should be held in the `Science` of the parameter file. Each Science target is created seperately with an integer index. Any entries above `sim.nSci` will be ignored.

Required:

Parameter	Description
`position`	tuple: The position of the science camera in the field in arc-seconds
`FOV`	float: The field of fiew of the science detector in arc-seconds
`wavelength`	float: The wavelength of the science detector light
`pxls`	int: Number of pixels in the science detector

Optional:

Parameter	Description	Default
`pxlScale`	float: Pixel scale of science camera, in arcseconds. If set, overwrites `FOV`.	`None`
`type`	string: Type of science camera This must the name of a class in the `SCI` module.	`PSF`
`fftOversamp`	int: Multiplied by the number of of phase points required for FOV to increase fidelity from FFT.	`2`
`fftwThreads`	int: number of threads for fftw to use. If `0`, will use system processor number.	`1`
`fftwFlag`	str: Flag to pass to FFTW when preparing plan.	`FFTW_MEASURE`
`height`	float: Altitude of the object. 0 denotes infinity.	`0`
`propagationMode`	str: Mode of light propogation from object. Can be “Physical” or “Geometric”.	`"Geometric"`
`instStrehlWithTT`	bool: Whether or not to include tip/tilt in instantaneous Strehl calculations.	`False`

CHAPTER 6

Data Sources

In this section, the data sources which are stored in soapy are listed and a description of how they are obtained is given.

Simulation Run Data

The following sources of data are recorded for each simulation run and are saved as a fits file in a time stamped run specific directory inside the `simName` directory. They can be accessed by `sim.<data>`, where `<data>` is listed in the "Internal data structure" column. As the storing of some of these data sources can increase memory usage significantly, they are not all saved by default, and the flag must be set in the configuration file.

Data	Saved filename	Internal data structure	Description
Instantaneous Strehl ratio	`instStrehl.fits`	`instStrehl`	The instantaneous strehl ratio for each science target frame
Long exposure Strehl ratio	`longStrehl.fits`	`longStrehl`	The long exposure strehl ratio for each science target frame
Wavefront Error	`WFE.fits`	`WFE`	The corrected wave- front error for each science target in nm
Science PSF	`sciPsf_n.fits`	`sciImgs[n]`	The science camera PSFs where `n` indicates the camera number
Residual Science phase	`sciResidual_n.fits`	`sciPhase[n]`	The residual uncorrected phase across science target `n`
WFS measurements	`slopes.fits`	`allSlopes`	All WFS measurements stored in a numpy array of size (nIters, totalSlopes)
WFS Frames	`wfsFPFrames/wfs-n_frame-i.fits`	`sim.wfss[n].wfsDetectorPlane`	WFS detector image, only last frame stored in memory. Can save each frame, `i`, from wfs `n`
DM Commands	`dmCommands.fits`	`allDmCommands`	DM commands for all DMs present in numpy of size (nIters, totaldmCommands)

CHAPTER 7

Simulation Design

Data flow and modularity

Soapy has been designed from the beginning to be extremely modular, where each AO component can be used individually. In fact, the file *simulation.py*, really only acts as a shepherd, moving data around between the components, with some fancy bits for saving data and printing nice outputs. A simple control loop to replace that file could be written from scratch in only 5-10 lines of Python!

This modularity is well illustrated by a data flow diagram describing the simulations, show in Figure 1, below.

Figure 1. Soapy Data Flow

Class Hierarchy

Pythons Object Orientated nature has also been exploited. Categories of AO component have a *base class*, which deals with most of the interfaces to the main simulation module and other boiler-plate style code. The classes which represent actual AO modules inherit this base class, and hopefully need only add interesting functionality specific to that new component. This is illustrated in the class diagram in Figure 2, with some example methods and attributes of each class.

Figure 2. Class diagram with example attributes and methods

It is aimed that in future developments of Soapy, this philosophy will be extended. Currently the WFS, science camera and LGS modules all deal with optical propagation through turbulence separately, clearly this should be combined into one place to ease code readability and maintenance. This work is currently under development. Figure 3 shows all the Soapy classes in a simplified class diagram, including the new *LineOfSight* class currently under construction.

Figure 3. Full, simplified class diagram with the lineOfSight class under construction.

CHAPTER 8

Simulation

High level interface to run and examine a simulation The main Soapy Simulation module

This module contains the `Sim` class, which can be used to run an end-to-end simulation. Initally, a configuration file is read, the system is initialised, interaction and command matrices calculated and finally a loop run. The simulation outputs some information to the console during the simulation.

The `Sim` class holds all configuration information and data from the simulation.

Examples

To initialise the class:

```
import soapy
sim = soapy.Sim("sh_8x8_4.2m.py")
```

Configuration information has now been loaded, and can be accessed through the `config` attribute of the `sim` class. In fact, each sub-module of the system has a configuration object accessed through this config attribute:

```
print(sim.config.sim.pupilSize)
sim.config.wfss[0].pxlsPerSubap = 10
```

Next, the system is initialised, this entails calculating various parameters in the system sub-modules, so must be done after changing some simulation parameters:

```
sim.aoinit()
```

DM Interation and command matrices are calculated now. If `sim.config.sim.simName` is not `None`, then these matrices will be saved in `data/simName` (data will be saved here also in a time-stamped directory):

```
sim.makeIMat()
```

Finally, the loop is run with the command:

```
sim.aoloop()
```

Some output will be printed to the console. After the loop has finished, data specified to be saved in the config file will be saved to `data/simName` (if it is not set to `None`). Data can also be accessed from the simulation class, e.g. `sim.allSlopes`, `sim.longStrehl`

Author Andrew Reeves

class soapy.simulation.**DelayBuffer**
Bases: list

A delay buffer.

Each time delay() is called on the buffer, the input value is stored. If the buffer is larger than count, the oldest value is removed and returned. If the buffer is not yet full, a zero of similar shape as the last input is returned.

delay(*value*, *count*)

class soapy.simulation.**Sim**(*configFile=None*)
Bases: object

The soapy Simulation class.

This class holds all configuration information, data and control methods of the simulation. It contains high level methods dealing with initialising all component objects, making reconstructor control matrices, running the loop and saving data after the loop has run.

Can be sub-classed and the 'aoloop' method overwritten for different loops to be used

Parameters **configFile** (*string*) – The filename of the AO configuration file

addToGuiQueue()
Adds data to a Queue object provided by the soapy GUI.

The soapy GUI doesn't need to plot every frame from the simulation. When it wants a frame, it will request if by setting `waitingPlot = True`. As this function is called on every iteration, data is passed to the GUI only if `waitingPlot = True`. This allows efficient and abstracted interaction between the GUI and the simulation

aoinit()
Initialises all simulation objects.

Initialises and passes relevant data to sim objects. This does important pre-run tasks, such as creating or loading phase screens, determining WFS geometry, setting propagation modes and pre-allocating data arrays used later in the simulation.

aoloop()
Main AO Loop

Runs a WFS iteration, reconstructs the phase, runs DMs and finally the science cameras. Also makes some nice output to the console and can add data to the Queue for the GUI if it has been requested. Repeats for nIters.

finishUp()
Prints a message to the console giving timing data. Used on sim end.

getTimeStamp()
Returns a formatted timestamp

Returns nicely formatted timestamp of current time.

Return type string

initSaveData()
Initialise data structures used for data saving.

Initialise the data structures which will be used to store data which will be saved or analysed once the simulation has ended. If the `simName = None`, no data is saved, other wise a directory called `simName` is created, and data from simulation runs are saved in a time-stamped directory inside this.

loopFrame()
Runs a single from of the entire AO system.

Moves the atmosphere, runs the WFSs, finds the corrective DM shape and finally runs the science cameras. This can be called over and over to form the "loop"

makeIMat(*forceNew=False*, *progressCallback=None*)
Creates interaction and control matrices for simulation reconstruction

Makes and inverts Interaction matrices for each DM in turn to create a DM control Matrix for each DM. Each DM's control Matrix is independent of the others, so care must be taken so DM correction modes do not "overlap". Some reconstruction modes may require WFS frames to be taken for the creation of a control matrix. Depending on set parameters, can load previous control and interaction matrices.

Parameters

- **forceNew** (*bool*) – if true, will force making of new iMats and cMats, otherwise will attempt to load previously made matrices from same simName
- **progressCallback** (*func*) – function called to report progress of interaction matrix construction

makeSaveHeader()
Forms a header which can be used to give a header to FITS files saved by the simulation.

printOutput(*iter*, *strehl=False*)
Prints simulation information to the console

Called on each iteration to print information about the current simulation, such as current strehl ratio, to the console. Still under development :param label: Simulation Name :type label: str :param iter: simulation frame number :type iter: int :param strehl: current strehl ration if science cameras are present to record it. :type strehl: float, optional

readParams(*configFile=None*)
Reads configuration file parameters

Calls the radParams function in confParse to read, parse and if required set reasonable defaults to AO parameters

reset_loop()
Resets parameters in the system to zero, to restart an AO run wihtout reinitialising

runDM(*dmCommands*, *closed=True*)
Runs a single frame of the deformable mirrors

Calculates the total combined shape of all deformable mirrors (DMs), given an array of DM commands. DM commands correspond to shapes generated during the making of interaction matrices, the final DM shape for each DM is a combination of these. The DM commands will have already been calculated by the systems reconstructor.

Parameters

- **dmCommands** (*ndarray*) – an array of dm commands corresponding to dm shapes
- **closed** (*bool*) – if True, indicates to DM that slopes are residual errors from previous frame, if False, slopes correspond to total phase error over pupil.

Returns the combined DM shape

Return type ndArray

runSciCams(*dmShape=None*)
Runs a single frame of the science Cameras

Calculates the image recorded by all science cameras in the system for the current phase over the telescope one frame. If a dmShape is present (which it usually will be in AO!) this correction is applied to the science phase before the image is calculated.

Parameters **correction** (*list or ndarray, optional*) – An array of the combined system DM shape to correct the science path. If not given science cameras are in open loop.

runWfs_MP(*scrns=None*, *dmShape=None*, *wfsList=None*, *loopIter=None*)
Runs all WFSs using multiprocessing

Runs a single frame for each WFS in wfsList, passing the given phase screens and optional dmShape (if WFS in closed loop). If LGSs are present it will also deals with LGS propagation. Finally, the slopes from all WFSs are returned. Each WFS is allocated a separate process to complete the frame, giving a significant increase in speed, especially for computationally heavy WFSs.

Parameters

- **scrns** (*list*) – List of phase screens passing over telescope
- **dmShape** (*ndarray, optional*) – 2-dimensional array of the total corrector shape
- **wfsList** (*list, optional*) – A list of the WFSs to be run, if not set, runs all WFSs
- **loopIter** (*int, optional*) – The loop iteration number

Returns The slope data return from the WFS frame (may not be actual slopes if WFS other than SH used)

Return type ndarray

runWfs_noMP(*scrns=None*, *dmShape=None*, *wfsList=None*, *loopIter=None*)
Runs all WFSs

Runs a single frame for each WFS in wfsList, passing the given phase screens and optional dmShape (if WFS in closed loop). The WFSs are only read out if the wfs frame time co-incides with the WFS frame rate, else old slopes are provided. If iter is not given, then all WFSs are run and read out. If LGSs are present it will also deals with LGS propagation. Finally, the slopes from all WFSs are returned.

Parameters

- **scrns** (*list*) – List of phase screens passing over telescope
- **dmShape** (*ndarray, optional*) – 2-dim array of the total corrector shape
- **wfsList** (*list, optional*) – A list of the WFSs to be run
- **loopIter** (*int, optional*) – The loop iteration number

Returns The slope data return from the WFS frame (may not be actual slopes if WFS other than SH used)

Return type ndarray

saveData()
Saves all recorded data to disk

Called once simulation has ended to save the data recorded during the simulation to disk in the directories created during initialisation.

setLoggingLevel(*level*)
sets which messages are printed from logger.

if logging level is set to 0, nothing is printed. if set to 1, only warnings are printed. if set to 2, warnings and info is printed. if set to 3 detailed debugging info is printed.

Parameters **level** (*int*) – the desired logging level

storeData(*i*)
Stores data from each frame in an appropriate data structure.

Called on each frame to store the simulation data into various data structures corresponding to different data sources in the system.

Parameters **i** (*int*) – The system iteration number

soapy.simulation.**make_mask**(*config*)
Generates a Soapy pupil mask

Parameters **config** (*SoapyConfig*) – Config object describing Soapy simulation

Returns 2-d pupil mask

Return type ndarray

soapy.simulation.**multiWfs**(*scrns*, *wfsObj*, *dmShape*, *read*, *queue*)
Function to run the WFS in multiprocessing mode.

Function is called by each of the new WFS processes spawned to run each WFS. Does the same job as the sim runWfs_noMP method of running LGS, then getting slopes from each WFS.

Parameters

- **scrns** (*list*) – list of the phase screens over the WFS
- **wfsObj** (*WFS object*) – the WFS object being run
- **dmShape** (*ndArray*) – shape of system DMs for WFS phase correction
- **queue** (*Queue object*) – a multiprocessing Queue object used to pass data back to host process.

CHAPTER 9

Atmosphere

The Soapy module used to simulate the atmosphere.

This module contains an `atmos` object, which can be used to create or load a specified number of phase screens corresponding to atmospheric turbulence layers. The layers can then be moved with the `moveScrns` method, at a specified wind velocity and direction, where the screen is interpolated if it does not fall on an integer number of pixels. Alternatively, random screens with the same statistics as the global phase screens can be generated using the `randomScrns` method.

The module also contains a number of functions used to create the phase screens, many of these are ported from the book *Numerical Simulation of Optical Propagation*, Schmidt, 2010. It is possible to create a number of phase screens using the *`makePhaseScreens()`* function which are saved to file in a format which can be read by the simulation.

Examples

To get the configuration objects:

```
from soapy import confParse, atmosphere

config = confParse.loadSoapyConfig("configfile.yaml")
```

Initialise the amosphere (creating or loading phase screens):

```
atmosphere = atmosphere.atmos(config)
```

Run the atmosphere for 10 time steps:

```
for i in range(10):
    phaseScrns = atmosphere.moveScrns()
```

or create 10 sets of random screens:

```
for i in range(10):
    randomPhaseScrns = atmosphere.randomScrns()
```

Atmosphere Class

class soapy.atmosphere.**atmos**(*soapyConfig*)

Class to simulate atmosphere above an AO system.

On initialisation of the object, new phase screens can be created, or others loaded from .fits file. The atmosphere is created with parameters given in ConfigObj.sim and ConfigObj.atmos. These are soapy configuration objects, which can be created by the :ref:confParse module, or could be created manually. If created manually, check the :ref: confParse section to see which attributes the configuration objects must contain.

If loaded from file, the screens should have a header with the parameter R0 specifying the r0 fried parameter of the screen in pixels.

The method moveScrns can be called on each iteration of the AO system to move the scrns forward by one time step. The size of this is defined by parameters given in

The method randomScrns returns a set of random phase screens with the smame statistics as the atmos object.

Parameters **soapyConfig** (*ConfigObj*) – The Soapy config object

moveScrns()

Moves the phase screens one time-step, defined by the atmosphere object parameters.

Returned phase is in units of nana-meters

Returns a dictionary containing the new set of phase screens

Return type dict

randomScrns(*subHarmonics=True*, *l0=0.01*)

Generated random phase screens defined by the atmosphere object parameters.

Returned phase is in units of nana-meters

Returns a dictionary containing the new set of phase screens

Return type dict

saveScrns(*DIR*)

Saves the currently loaded phase screens to file, saving the r0 value in the fits header (in units of pixels). Saved phase data is in radians @500nm

Parameters **DIR** (*string*) – The directory to save the screens

Phase Screen Creation and Saving

soapy.atmosphere.**makePhaseScreens**(*nScrns*, *r0*, *N*, *pxlScale*, *L0*, *l0*, *returnScrns=True*, *DIR=None*, *SH=False*)

Creates and saves a set of phase screens to be used by the simulation.

Creates nScrns phase screens, with the required parameters, then saves them to the directory specified by DIR. Each screen is given a FITS header with its value of r0, which will be scaled by on simulation when its loaded.

Parameters

- **nScrns** (*int*) – The number of screens to make.
- **r0** (*float*) – r0 value of the phase screens in metres.
- **N** (*int*) – Number of elements across each screen.

- **pxlScale** (*float*) – Size of each element in metres.
- **L0** (*float*) – Outer scale of each screen.
- **l0** (*float*) – Inner scale of each screen.
- **returnScrns** (*bool, optional*) – Whether to return a list of screens. True by default, but if screens are very large, False might be preferred so they aren't kept in memory if saving to disk.
- **DIR** (*str, optional*) – The directory to save the screens.
- **SH** (*bool, optional*) – If True, add sub-harmonics to screens for more accurate power spectra, though screens no-longer periodic.

Returns A list containing all the screens.

Return type list

CHAPTER 10

Line Of Sight

soapy.lineofsight module

A generalised module to provide phase or the EField through a "Line Of Sight"

Line of Sight Object

The module contains a 'lineOfSight' object, which calculates the resulting phase or complex amplitude from propogating through the atmosphere in a given direction. This can be done using either geometric propagation, where phase is simply summed for each layer, or physical propagation, where the phase is propagated between layers using an angular spectrum propagation method. Light can propogate either up or down.

The Object takes a 'config' as an argument, which is likely to be the same config object as the module using it (WFSs, ScienceCams, or LGSs). It should contain paramters required, such as the observation direction and light wavelength. The *config* also determines whether to use physical or geometric propagation through the 'propagationMode' parameter.

Examples:

```
from soapy import confParse, lineofsight

# Initialise a soapy conifuration file
config = confParse.loadSoapyConfig('conf/sh_8x8.py')

# Can make a 'LineOfSight' for WFSs
los = lineofsight.LineOfSight(config.wfss[0], config)

# Get resulting complex amplitude through line of sight
EField = los.frame(some_phase_screens)
```

class soapy.lineofsight.**LineOfSight**(*config*, *soapyConfig*, *propagation_direction='down'*, *out_pixel_scale=None*, *nx_out_pixels=None*, *mask=None*, *metaPupilPos=None*)

Bases: object

A "Line of sight" through a number of turbulence layers in the atmosphere, observing ing a given direction.

Parameters

- **config** – The soapy config for the line of sight
- **simConfig** – The soapy simulation config object
- **propagationDirection** (*str, optional*) – Direction of light propagation, either *"up"* or *"down"*
- **outPxlScale** (*float, optional*) – The EField pixel scale required at the output (m/pxl)
- **nOutPxls** (*int, optional*) – Number of pixels to return in EFIeld
- **mask** (*ndarray, optional*) – Mask to apply at the *beginning* of propagation
- **metaPupilPos** (*list, dict, optional*) – A list or dictionary of the meta pupil position at each turbulence layer height ub metres. If None, works it out from GS position.

allocDataArrays()
Allocate the data arrays the LOS will require

Determines and allocates the various arrays the LOS will require to avoid having to re-alloc memory during the running of the LOS and keep it fast. This includes arrays for phase and the E-Field across the LOS

calcInitParams(*out_pixel_scale=None*, *nx_out_pixels=None*)
Calculates some parameters required later

Parameters

- **outPxlScale** (*float*) – Pixel scale of required phase/EField (metres/pxl)
- **nOutPxls** (*int*) – Size of output array in pixels

calculate_altitude_coords(*layer_altitude*)
Calculate the co-ordinates of vertices of fo the meta-pupil at altitude given a guide star direction and source altitude

Paramters: layer_altitude (float): Altitude of phase layer

frame(*scrns=None*, *correction=None*)
Runs one frame through a line of sight

Finds the phase or complex amplitude through line of sight for a single simulation frame, with a given set of phase screens and some optional correction.

Parameters

- **scrns** (*list*) – A list or dict containing the phase screens
- **correction** (*ndarray, optional*) – The correction term to take from the phase screens before the WFS is run.
- **read** (*bool, optional*) – Should the WFS be read out? if False, then WFS image is calculated but slopes not calculated. defaults to True.

Returns WFS Measurements

Return type ndarray

height

makePhase(*radii=None*, *apos=None*)
Generates the required phase or EField. Uses difference approach depending on whether propagation is geometric or physical (makePhaseGeometric or makePhasePhys respectively)

Parameters

- **radii** (*dict, optional*) – Radii of each meta pupil of each screen height in pixels. If not given uses pupil radius.
- **apos** (*ndarray, optional*) – The angular position of the GS in radians. If not set, will use the config position

makePhaseGeometric(*radii=None*, *apos=None*)

Creates the total phase along line of sight offset by a given angle using a geometric ray tracing approach

Parameters

- **radii** (*dict, optional*) – Radii of each meta pupil of each screen height in pixels. If not given uses pupil radius.
- **apos** (*ndarray, optional*) – The angular position of the GS in radians. If not set, will use the config position

makePhasePhys(*radii=None*, *apos=None*)

Finds total line of sight complex amplitude by propagating light through phase screens

Parameters

- **radii** (*dict, optional*) – Radii of each meta pupil of each screen height in pixels. If not given uses pupil radius.
- **apos** (*ndarray, optional*) – The angular position of the GS in radians. If not set, will use the config position

performCorrection(*correction*)

Corrects the aberrated line of sight with some given correction phase

Parameters **correction** (*list or ndarray*) – either 2-d array describing correction, or list of correction arrays

position

zeroData(***kwargs*)

Sets the phase and complex amp data to zero

soapy.lineofsight.**physical_atmosphere_propagation**(*phase_screens*, *output_mask*, *layer_altitudes*, *source_altitude*, *wavelength*, *output_pixel_scale*, *propagation_direction='up'*)

Finds total line of sight complex amplitude by propagating light through phase screens

Parameters

- **radii** (*dict, optional*) – Radii of each meta pupil of each screen height in pixels. If not given uses pupil radius.
- **apos** (*ndarray, optional*) – The angular position of the GS in radians. If not set, will use the config position

CHAPTER 11

Wave-front Sensors

WFS Module

The Soapy WFS module.

This module contains a number of classes which simulate different adaptive optics wavefront sensor (WFS) types. All wavefront sensor classes can inherit from the base `WFS` class. The class provides the methods required to calculate phase over a WFS pointing in a given WFS direction and accounts for Laser Guide Star (LGS) geometry such as cone effect and elongation. This is If only pupil images (or complex amplitudes) are required, then this class can be used stand-alone.

Example

Make configuration objects:

```
from soapy import WFS, confParse

config = confParse.Configurator("config_file.py")
config.loadSimParams()
```

Initialise the wave-front sensor:

```
wfs = WFS.WFS(config, 0 mask)
```

Set the WFS scrns (these should be made in advance, perhaps by the *soapy.atmosphere* module). Then run the WFS:

```
wfs.scrns = phaseScrnList
wfs.makePhase()
```

Now you can view data from the WFS frame:

```
frameEField = wfs.EField
```

A Shack-Hartmann WFS is also included in the module, this contains further methods to make the focal plane, then calculate the slopes to send to the reconstructor.

Example

Using the config objects from above...:

```
shWfs = WFS.ShackHartmann(config, 0, mask)
```

As we are using a full WFS with focal plane making methods, the WFS base classes `frame` method can be used to take a frame from the WFS:

```
slopes = shWfs.frame(phaseScrnList)
```

All the data from that WFS frame is available for inspection. For instance, to obtain the electric field across the WFS and the image seen by the WFS detector:

```
EField = shWfs.EField
wfsDetector = shWfs.wfsDetectorPlane
```

Adding new WFSs

New WFS classes should inherit the `WFS` class, then create methods which deal with creating the focal plane and making a measurement from it. To make use of the base-classes `frame` method, which will run the WFS entirely, the new class must contain the following methods:

```
calcFocalPlane(self)
makeDetectorPlane(self)
calculateSlopes(self)
```

The Final `calculateSlopes` method must set `self.slopes` to be the measurements made by the WFS. If LGS elongation is to be used for the new WFS, create a `detectorPlane`, which is added to for each LGS elongation propagation. Have a look at the code for the `Shack-Hartmann` and experimental `Pyramid` WFSs to get some ideas on how to do this.

Author Andrew Reeves

Base WFS Class

class `soapy.wfs.base.`**`WFS`**(*soapy_config*, *n_wfs=0*, *mask=None*)

A WFS class.

This is a base class which contains methods to initialise the WFS, and calculate the phase across the WFSs input aperture, given the WFS guide star geometry.

Parameters

- **`soapy_config`** (*`ConfigObj`*) – The soapy configuration object
- **`nWfs`** (*`int`*) – The ID number of this WFS
- **`mask`** (*`ndarray, optional`*) – An array or size (simConfig.simSize, simConfig.simSize) which is 1 at the telescope aperture and 0 else-where.

addPhotonNoise()
Add photon noise to `wfsDetectorPlane` using `numpy.random.poisson`

addReadNoise()
Adds read noise to `wfsDetectorPlane using ``numpy.random.normal`. This generates a normal (guassian) distribution of random numbers to add to the detector. Any CCD bias is assumed to have been removed, so the distribution is centred around 0. The width of the distribution is determined by the value *eReadNoise* set in the WFS configuration.

calcElongPhaseAddition(*elongLayer*)
Calculates the phase required to emulate layers on an elongated source

For each 'elongation layer' a phase addition is calculated which accounts for the difference in height from the nominal GS height where the WFS is focussed, and accounts for the tilt seen if the LGS is launched off-axis.

Parameters **elongLayer** (*int*) – The number of the elongation layer

Returns The phase addition required for that layer.

Return type ndarray

calcElongPos(*elongLayer*)
Calculates the difference in GS position for each elongation layer only makes a difference if LGS launched off-axis

Parameters **elongLayer** (*int*) – which elongation layer

Returns The effective position of that layer GS on the simulation phase grid

Return type float

frame(*scrns*, *phase_correction=None*, *read=True*, *iMatFrame=False*)
Runs one WFS frame

Runs a single frame of the WFS with a given set of phase screens and some optional correction. If elongation is set, will run the phase calculating and focal plane making methods multiple times for a few different heights of LGS, then sum these onto a `wfsDetectorPlane`.

Parameters

- **scrns** (*list*) – A list or dict containing the phase screens
- **correction** (*ndarray, optional*) – The correction term to take from the phase screens before the WFS is run.
- **read** (*bool, optional*) – Should the WFS be read out? if False, then WFS image is calculated but slopes not calculated. defaults to True.
- **iMatFrame** (*bool, optional*) – If True, will assume an interaction matrix is being measured. Turns off some AO loop features before running

Returns WFS Measurements

Return type ndarray

initLGS()
Initialises the LGS objects for the WFS

Creates and initialises the LGS objects if the WFS GS is a LGS. This included calculating the phases additions which are required if the LGS is elongated based on the depth of the elongation and the launch position. Note that if the GS is at infinity, elongation is not possible and a warning is logged.

initLos()
Initialises the `LineOfSight` object, which gets the phase or EField in a given direction through turbulence.

makeElongationFrame(*correction=None*)
Find the focal plane resulting from an elongated guide star, such as LGS.

Runs the phase stacking and propagation routines multiple times with different GS heights, positions and/or aberrations to simulation the effect of a number of points in an elongation guide star.

setMask(*mask*)
Sets the pupil mask as seen by the WFS.

This method can be called during a simulation

class `soapy.wfs.shackhartmann.`**ShackHartmann**(*soapy_config*, *n_wfs=0*, *mask=None*)
Class to simulate a Shack-Hartmann WFS

addPhotonNoise()
Add photon noise to `wfsDetectorPlane` using `numpy.random.poisson`

addReadNoise()
Adds read noise to `wfsDetectorPlane using ``numpy.random.normal`. This generates a normal (guassian) distribution of random numbers to add to the detector. Any CCD bias is assumed to have been removed, so the distribution is centred around 0. The width of the distribution is determined by the value *eReadNoise* set in the WFS configuration.

allocDataArrays()
Allocate the data arrays the WFS will require

Determines and allocates the various arrays the WFS will require to avoid having to re-alloc memory during the running of the WFS and keep it fast.

applyLgsUplink()
A method to deal with convolving the LGS PSF with the subap focal plane.

calcFocalPlane(*intensity=1*)
Calculates the wfs focal plane, given the phase across the WFS

Parameters **intensity** (*float*) – The relative intensity of this frame, is used when multiple WFS frames taken for extended sources.

calcInitParams()
Calculate some parameters to be used during initialisation

calcTiltCorrect()
Calculates the required tilt to add to avoid the PSF being centred on only 1 pixel

calculateSlopes()
Calculates WFS slopes from wfsFocalPlane

Returns array of all WFS measurements

Return type ndarray

findActiveSubaps()
Finds the subapertures which are not empty space determined if mean of subap coords of the mask is above threshold.

getStatic()
Computes the static measurements, i.e., slopes with flat wavefront

initFFTs()
Initialise the FFT Objects required for running the WFS

Initialised various FFT objects which are used through the WFS, these include FFTs to calculate focal planes, and to convolve LGS PSFs with the focal planes

initLos()
Initialises the `LineOfSight` object, which gets the phase or EField in a given direction through turbulence.

makeDetectorPlane()
Scales and bins intensity data onto the detector with a given number of pixels.

If required, will first convolve final PSF with LGS PSF, then bin PSF down to detector size. Finally puts back into `wfsFocalPlane` array in correct order.

zeroData(*detector=True*, *FP=True*)
Sets data structures in WFS to zero.

Parameters

- **detector** (*bool, optional*) – Zero the detector? default:True
- **FP** (*bool, optional*) – Zero intermediate focal plane arrays? default: True

CHAPTER 12

Deformable Mirrors

The module simulating Deformable Mirrors in Soapy

DMs in Soapy

DMs are represented in Soapy by python objects which are initialised at startup with some configuration parameters given, as well as a list of one or more WFS objects which can be used to measure an interaction matrix.

Upon creation of an interaction matrix, the object first generations all the possible independant shapes which the DM may form, known as "influence functions". Then each influence function is passed to the specified WFS(s) and the response noted to form an interaction matrix. The interaction matrix may then be used to forma reconstructor.

During the AO loop, commands corresponding to the required amplitude of each DM influence function are sent to the *DM.dmFrame()* method, which returns an array representing the DMs shape.

Adding New DMs

New DMs are easy to add into the simulation. At its simplest, the *DM* class is inherited by the new DM class. Only a ‘‘makeIMatShapes‘ method need be provided, which creates the independent influence function the DM can make. The base class deals with the rest, including making interaction matrices and loop operation.

Base DM Class

class soapy.DM.**DM**(*soapy_config*, *n_dm=0*, *wfss=None*, *mask=None*)

Bases: object

The base DM class

This class is intended to be inherited by other DM classes which describe real DMs. It provides methods to create DM shapes and then interaction matrices, given a specific WFS or WFSs.

Parameters

- **soapy_config** (*ConfigObj*) – The soapy configuration object
- **n_dm** (*int*) – The ID number of this DM
- **wfss** (*list, optional*) – A list of Soapy WFS object with which to record the interaction matrix
- **mask** (*ndarray, optional*) – An array or size (simConfig.simSize, simConfig.simSize) which is 1 at the telescope aperture and 0 else-where. If None then a circle is generated.

dmFrame(*dmCommands*)

Uses DM commands to calculate the final DM shape.

Multiplies each of the DM influence functions by the corresponding DM command, then sums to create the final DM shape. Lastly, the mean value is subtracted to avoid piston terms building up.

Parameters

- **dmCommands** (*ndarray*) – A 1-dimensional vector of the multiplying factor of each DM influence function
- **closed** (*bool, optional*) – Specifies how to great gain. If "True" (closed) then "dmCommands" are multiplied by gain and summed with previous commands. If "False" (open), then "dmCommands" are multiplied by gain, and summed withe previous commands multiplied by (1-gain).

Returns A 2-d array with the DM shape

Return type ndarray

getActiveActs()

Method returning the total number of actuators used by the DM - May be overwritten in DM classes

Returns number of active DM actuators

Return type int

makeIMatShapes()

Virtual method to generate the DM influence functions

Real DM Classes

class soapy.DM.**TT**(*soapy_config*, *n_dm=0*, *wfss=None*, *mask=None*)

Bases: *soapy.DM.DM*

A class representing a tip-tilt mirror.

This can be used as a tip-tilt mirror, it features two actuators, where each influence function is simply a tip and a tilt.

getActiveActs()

Returns the number of active actuators on the DM. Always 2 for a TT.

makeIMatShapes()

Forms the DM influence functions, in this case just a tip and a tilt.

class soapy.DM.**Zernike**(*soapy_config*, *n_dm=0*, *wfss=None*, *mask=None*)

Bases: *soapy.DM.DM*

A DM which corrects using a provided number of Zernike Polynomials

makeIMatShapes()
Creates all the DM shapes which are required for creating the interaction Matrix. In this case, this is a number of Zernike Polynomials

class soapy.DM.**Piezo**(*soapy_config*, *n_dm=0*, *wfss=None*, *mask=None*)
Bases: *soapy.DM.DM*

A DM emulating a Piezo actuator style stack-array DM.

This class represents a standard stack-array style DM with push-pull actuators behind a continuous phase sheet. The number of actuators is given in the configuration file.

Each influence function is created by started with an N x N grid of zeros, where N is the number of actuators in one direction, and setting a single value to `1`, which corresponds with a "pushed" actuator. This grid is then interpolated up to the `pupilSize`, to form the shape of the DM when that actuator is activated. This is repeated for all actuators.

getActiveActs()
Finds the actuators which will affect phase whithin the pupil to avoid reconstructing for redundant actuators.

makeIMatShapes()
Generate Piezo DM influence functions

Generates the shape of each actuator on a Piezo stack DM (influence functions). These are created by interpolating a grid on the size of the number of actuators, with only the 'poked' actuator set to 1 and all others set to zero, up to the required simulation size. This grid is actually padded with 1 extra actuator spacing to avoid strange edge effects.

class soapy.DM.**GaussStack**(*soapy_config*, *n_dm=0*, *wfss=None*, *mask=None*)
Bases: *soapy.DM.Piezo*

A Stack Array DM where each influence function is a 2-D Gaussian shape.

This class represents a Stack-Array DM, similar to the *Piezo* DM, where each influence function is a 2-dimensional Gaussian function. Though not realistic, it provides a known influence function which can be useful for some analysis.

makeIMatShapes()
Generates the influence functions for the GaussStack DM.

Creates a number of Guassian distributions which are centred at points across the pupil to act as DM influence functions. The width of the guassian is determined from the configuration file.

CHAPTER 13

Laser Guide Stars

Classes simulating Laser guide stars - usually contained by a *WFS* object.

soapy.LGS module

class soapy.LGS.**LGS**(*wfsConfig*, *soapyConfig*, *nOutPxls=None*, *outPxlScale=None*)
Bases: object

A class to simulate the propogation of a laser up through turbulence. Given a set of phase screens, this will return the PSF which would be present on-sky.

Parameters

- **simConfig** – The Soapy simulation config
- **wfsConfig** – The relavent Soapy WFS configuration
- **atmosConfig** – The relavent Soapy atmosphere configuration
- **nOutPxls** (*int*) – Number of pixels required in output LGS
- **outPxlScale** (*float*) – The pixel scale of the output LGS PSF in arcsecs per pixel

calcInitParams()

getLgsPsf(*scrns*)

initFFTs()
Virtual Method as many LGS implentations will require extra FFTs

initLos()
Initialises the LineOfSight object, which gets the phase or EField in a given direction through turbulence.

class soapy.LGS.**LGS_Geometric**(*wfsConfig*, *soapyConfig*, *nOutPxls=None*, *outPxlScale=None*)
Bases: *soapy.LGS.LGS*

A class to simulate the propogation of a laser up through turbulence using a geometric algorithm. Given a set of phase screens, this will return the PSF which would be present on-sky.

Parameters

- **simConfig** – The Soapy simulation config
- **wfsConfig** – The relavent Soapy WFS configuration
- **atmosConfig** – The relavent Soapy atmosphere configuration
- **nOutPxls** (*int*) – Number of pixels required in output LGS
- **outPxlScale** (*float*) – The pixel scale of the output LGS PSF in arcsecs per pixel

calcInitParams()
Calculate some useful paramters to be used later

getLgsPsf(*scrns*)

initFFTs()

class soapy.LGS.**LGS_Physical**(*wfsConfig*, *soapyConfig*, *nOutPxls=None*, *outPxlScale=None*)
Bases: *soapy.LGS.LGS*

A class to simulate the propogation of a laser up through turbulence using a geometric algorithm. Given a set of phase screens, this will return the PSF which would be present on-sky.

Parameters

- **simConfig** – The Soapy simulation config
- **wfsConfig** – The relavent Soapy WFS configuration
- **atmosConfig** – The relavent Soapy atmosphere configuration
- **nOutPxls** (*int*) – Number of pixels required in output LGS
- **outPxlScale** (*float*) – The pixel scale of the output LGS PSF in arcsecs per pixel

calcInitParams()
Calculate some useful paramters to be used later

getLgsPsf(*scrns=None*)
Return the LGS PSF to be used in WFS calculation

soapy.LGS.**lgsOALaunchMetaPupilPos**(*gsPos*, *launchPos*, *lgsHt*, *layerHt*)
Finds the centre of a meta-pupil in the atmosphere sampled by an LGS launched from a position off-axis from the centre of the telescope.

Parameters

- **gsPos** (*ndarray*) – The X,Y position of the guide star in arcsecs
- **launchPos** (*ndarray*) – The X, Y launch position of the telescope in metres from the telescope centre
- **lgsHt** (*float*) – The altitude of the LGS beacon
- **layerHt** (*float*) – The height of the meta-pupil of interest

Returns Position in X,Y from the on-axis line-of-sight of the meta-pupil centre.

Return type ndarray

CHAPTER 14

Reconstructors

Classes simulating AO reconstructors.

soapy.RECON module

CHAPTER 15

Science Camera

A science camera class to measure system performance

soapy.SCI module

class soapy.SCI.**PSF**(*soapyConfig*, *nSci=0*, *mask=None*)
Bases: object

calcFocalPlane()
Takes the calculated pupil phase, scales for the correct FOV, and uses an FFT to transform to the focal plane.

calcInstStrehl()
Calculates the instantaneous Strehl, including TT if configured.

calc_wavefronterror()
Calculates the wavefront error across the telescope pupil

Returns RMS WFE across pupil in nm

Return type float

frame(*scrns*, *correction=None*)
Runs a single science camera frame with one or more phase screens

Parameters

- **scrns** (*ndarray, list, dict*) – One or more 2-d phase screens. Phase in units of nm.
- **phaseCorrection** (*ndarray*) – Correction phase in nm

Returns Resulting science PSF

Return type ndarray

setMask(*mask*)
Sets the pupil mask as seen by the WFS.

This method can be called during a simulation

soapy.SCI.**ScienceCam**
alias of *PSF*

soapy.SCI.**scienceCam**
alias of *PSF*

class soapy.SCI.**singleModeFibre**(*soapyConfig*, *nSci=0*, *mask=None*)
Bases: *soapy.SCI.PSF*

calcInstStrehl()

fibreEfield(*size*)

refCouplingLoss(*size*)

CHAPTER 16

Utilities

Modules containing some functions and classes commonly used throughout the simulation.

soapy.logger module

A module to provide a common logging interface for all simulation code.

Contains a `Logger` object, which can either, print information, save to file or both. The verbosity can also be adjusted between 0 and 3, where all is logged when verbosity is 3, debugging and warning information is logged when verbosity is 2, warnings logged when verbosity is 1 and nothing is logged when verbosity is 0.

`soapy.logger.`**`debug`**(*message*)
Logs messages if debug level is 3. Intended for very detailed debugging information.

Parameters **`message`** (*string*) – The message to log

`soapy.logger.`**`info`**(*message*)
Logs message if verbosity is 2 or higher. Useful for information which is not vital, but good to know.

Parameters **`message`** (*string*) – The message to log

`soapy.logger.`**`print_`**(*message*)
Always logs message, regardless of verbosity level

Parameters **`message`** (*str*) – The message to log

`soapy.logger.`**`setLoggingFile`**(*logFile*)

`soapy.logger.`**`setLoggingLevel`**(*level*)
sets which messages are printed from logger.

if logging level is set to 0, nothing is printed. if set to 1, only warnings are printed. if set to 2, warnings and info is printed. if set to 3 detailed debugging info is printed.

Parameters **`level`** (*int*) – the desired logging level

`soapy.logger.`**`setStatusFunc`**(*func*)

soapy.logger.**statusMessage**(*i*, *maxIter*, *message*)

soapy.logger.**warning**(*message*)
Logs messages if debug level is 1 or over. Intended for warnings

Parameters **message** (*string*) – The message to log

soapy.AOFFT module

A Module to perform FFTs, wrapping a variety of FFT Backends in a common interface. Currently supports either pyfftw (requires FFTW3), the scipy fftpack or some GPU algorithms

class soapy.AOFFT.**Convolve**(*shape1*, *shape2=None*, *mode='pyfftw'*, *fftw_FLAGS=('FFTW_MEASURE',)*, *threads=0*, *axes=(-2, -1)*)
Bases: object

class soapy.AOFFT.**FFT**(*inputSize*, *axes=(-1,)*, *mode='pyfftw'*, *dtype='complex64'*, *direction='FORWARD'*, *fftw_FLAGS=('FFTW_MEASURE', 'FFTW_DESTROY_INPUT')*, *THREADS=None*, *loggingLevel=None*)
Bases: object

Class for performing FFTs in a variety of ways, with the same API.

Once the class has been initialised, FFTs going in the same direction and using the same padding size can be performed with re-initialising. The inputSize set is actually the padding size, any array smaller than this can then be transformed.

Usually, its best to best to pass the data when performing the fft, either calling the class directly (`fftobj(fftData)`) or calling the `fft` method of the class. If though, you're certain the array to transform is C-contiguous, and its size is the same as `inputSize`, then you can set:

```
fftObj.inputData = fftData
```

then:

```
outputData = fftObj()
```

where `fftData` is the data to be transformed. This is faster, as it avoids an array copying operation, but is dangerous as the FFT may fail if the input data is not correct.

Parameters

- **inputSize** (*tuple*) – The size of the input array, including any padding
- **axes** (*tuple, optional*) – The axes to transform. defaults to the last.
- **mode** (*string, optional*) – Which FFT library to use, can by `'pyfftw'`, `'scipy'` or `'gpu'`. Defaults to `'pyfftw'`.
- **dtype** (*str, optional*) – The data type to transform, defaults to `'complex64'`
- **direction** (*str, optional*) – Forward or inverse FFT. Either *FORWARD* or *BACKWARD*. Default is *FORWARD*.
- **THREADS** (*int, optional*) – Number of threads to use for FFT. Defualt is 1

fft(*data=None*)
Perform the fft of *data*.

Parameters **data** (*ndarray, optional*) – The data to transform. Optional as sometimes it can be faster to access inputData directly, though if and only if the data will be c-contiguous.

Returns The transformed data

Return type ndarray

soapy.AOFFT.**convolve**(*img1*, *img2*, *mode='pyfftw'*, *fftw_FLAGS=('FFTW_MEASURE',)*, *threads=0*)
Convolves two, 2-dimensional arrays Uses the AOFFT library to do fast convolution of 2, 2-dimensional numpy ndarrays. The FFT mode, and some parameters can be set in the arguments. :param img1: 1st array to be convolved :type img1: ndarray :param img2: 2nd array to be convolved :type img2: ndarray :param mode: The fft mode used, defaults to fftw :type mode: string, optional :param fftw_FLAGS: flags for fftw, defaults to ("FFTW_MEASURE",) :type fftw_FLAGS: tuple, optional :param threads: Number of threads used if mode is fftw :type threads: int, optional

Returns The convolved 2-dimensional array

Return type ndarray

soapy.AOFFT.**ftShift2d**(*inputData*, *outputData=None*)
Helper function to shift an array of 2-D FFT data

Parameters

- **inputData** (*ndarray*) – array of data to be shifted. Will shift final 2 axes
- **outputData** (*ndarray, optional*) – array to place data. If not given, will overwrite inputData

class soapy.AOFFT.**mpFFT**(*inputSize*, *axes=(-1,)*, *mode='pyfftw'*, *dtype='complex64'*, *direction='FORWARD'*, *fftw_FLAGS=('FFTW_MEASURE',)*, *processes=None*)
Bases: object

Class to perform FFTs on a large number of problems, using the FFT class, and seperate processes for different problems. The input array will be split in the 0 axis onto different processes

doMpFFT(*fftObj*, *data*, *Q*)

fft()

soapy.aoSimLib module

soapy.opticalPropagationLib module

soapy.confParse module

A module to generate configuration objects for Soapy, given a parameter file.

This module defines a number of classes, which when instantiated, create objects used to configure the entire simulation, or just submodules. All configuration objects are stored in the Configurator object which deals with loading parameters from file, checking some potential conflicts and using parameters to calculate some other parameters used in parts of the simulation.

The ConfigObj provides a base class used by other module configuration objects, and provides methods to read the parameters from the dictionary read from file, and set defaults if appropriate. Each other module in the system has its own configuration object, and for components such as wave-front sensors (WFSs), Deformable Mirrors (DMs), Laser Guide Stars (LGSs) and Science Cameras, lists of the config objects for each component are created.

class soapy.confParse.**AtmosConfig**(*N=None*)
Bases: soapy.confParse.ConfigObj

Configuration parameters characterising the atmosphere. These should be held in the Atmosphere group in the parameter file.

Required:

Parameter	Description
scrnNo	int: Number of turbulence layers
scrnHeights	list, int: Phase screen heights in metres
scrnStrength	list, float: Relative layer scrnStrength
windDirs	list, float: Wind directions in degrees.
windSpeeds	list, float: Wind velocities in m/s
r0	float: integrated seeing strength (metres at 500nm)

Optional:

Parameter	Description	Default
scrnNames	list, string: filenames of phase if loading from fits files. If None will make new screens.	None
subHarmonics	bool: Use sub-harmonic screen generation algorithm for better tip-tilt statistics - useful for small phase screens.	False
L0	list, float: Outer scale of each layer. Kolmogorov turbulence if None.	None
randomScrns	bool: Use a random set of phase phase screens for each loop iteration?	False
infinite	bool: Use infinite phase screens?	False
tau0	float: Turbulence coherence time, if set wind speeds are scaled.	None
wholeScrnSize	int: Size of the phase screens to store in the atmosphere object. Required if large screens used.	None

allowedAttrs = ['scrnNo', 'scrnHeights', 'scrnStrengths', 'r0', 'windDirs', 'windSpeeds', 'normScrnStrengths', 'N', 's

calcParams()

calculatedParams = ['normScrnStrengths']

optionalParams = [('scrnNames', None), ('subHarmonics', False), ('L0', None), ('randomScrns', False), ('tau0', None)

p = ('wholeScrnSize', None)

requiredParams = ['scrnNo', 'scrnHeights', 'scrnStrengths', 'r0', 'windDirs', 'windSpeeds']

class soapy.confParse.**ConfigObj**(*N=None*)
Bases: object

calcParams()
Dummy method to be overidden if required

initParams()

loadParams(*configDict*)

warnAndDefault(*param*, *newValue*)

warnAndExit(*param*)

exception soapy.confParse.**ConfigurationError**
Bases: exceptions.Exception

soapy.confParse.**Configurator**
alias of PY_Configurator

class soapy.confParse.**DmConfig**(*N=None*)
Bases: soapy.confParse.ConfigObj

Configuration parameters characterising Deformable Mirrors. These should be held in the `DM` sub-group of the parameter file. Each DM is specified seperately, by first specifying an index, then the DM parameters. Any entries above `sim.nGS` will be ignored.

Required:

Parameter	Description
`type`	string: Type of DM. This must the name of a class in the `DM` module.
`nxActuators`	int: Number independent DM shapes. e.g., for stack-array DMs this is number of actuators in one dimension, for Zernike DMs this is number of Zernike modes.
`gain`	float: The loop gain for the DM.**
`svdConditioning`	float: The conditioning parameter used in the pseudo inverse of the interaction matrix. This is performed by numpy.linalg.pinv.

Optional:

`allowedAttrs` = ['type', 'N', 'nxActuators', 'svdConditioning', 'gain', 'closed', 'iMatValue', 'wfs', 'rotation', 'interpO

`calcParams`()

`calculatedParams` = []

`optionalParams` = [('nxActuators', None), ('svdConditioning', 0), ('gain', 0.6), ('closed', True), ('iMatValue', 10), ('wfs

`p` = ('gauss_width', 0.7)

`requiredParams` = ['type']

class `soapy.confParse.`**`LgsConfig`**(*N=None*)

Bases: `soapy.confParse.ConfigObj`

Configuration parameters characterising the Laser Guide Stars. These should be held in the `LGS` sub-group of the WFS parameter group.

Optional:

Parameter	Description	Default
`uplink`	bool: Include LGS uplink effects	`False`
`pupilDiam`	float: Diameter of LGS launch aperture in metres.	`0.3`
`wavelength`	float: Wavelength of laser beam in metres	`600e-9`
`propagationMode`	str: Mode of light propogation from GS. Can be "Physical" or "Geometric".	`"Phsyical"`
`height`	float: Height to use physical propogation of LGS (does not effect cone-effect) in metres	`90000`
`elongationDepth`	float: Depth of LGS elongation in metres	`0`
`elongationLayers`	int: Number of layers to simulate for elongation.	`10`
`launchPosition`	tuple: The launch position of the LGS in units of the pupil radii, where `(0,0)` is the centre launched case, and `(1,0)` is side-launched.	`(0,0)`
`fftwThreads`	int: number of threads for fftw to use. If `0`, will use system processor number.	`1`
`fftwFlag`	str: Flag to pass to FFTW when preparing plan.	`FFTW_PATIENT`
`naProfile`	list: The relative sodium layer strength for each elongation layer. If None, all equal.	`None`

`allowedAttrs` = ['position', 'N', 'uplink', 'pupilDiam', 'wavelength', 'propagationMode', 'height', 'fftwFlag', 'fftwThr

`calcParams`()

`calculatedParams` = ['position']

`optionalParams` = [('uplink', False), ('pupilDiam', 0.3), ('wavelength', 6e-07), ('propagationMode', 'Physical'), ('heigh

`p` = ('naProfile', None)

`requiredParams` = []

class soapy.confParse.**PY_Configurator**(*filename*)
Bases: object

The configuration class holding all simulation configuration information

This class is used to load the parameter dictionary from file, instantiate each configuration object and calculate some other parameters from the parameters given.

The configuration file given to this class must contain a python dictionary, named simConfiguration. This must contain other dictionaries for each sub-module of the system, Sim, Atmosphere, Telescope, WFS, LGS, DM, Science. For the final 4 sub-dictionaries, each entry must be formatted as a list (or numpy array) where each value corresponds to that component.

The number of components on the module will only depend on the number set in the Sim dict. For example, if nGS is set to 2 in Sim, then in the WFS dict, each parameters must have at least 2 entries, e.g. subaps : [10,10]. If the parameter has more than 2 entries, then only the first 2 will be noted and any others discarded.

Descriptions of the available parameters for each sub-module are given in that that config classes documentation

Parameters **filename** (*string*) – The name of the configuration file

calcParams()
Calculates some parameters from the configuration parameters.

loadSimParams()

readfile()

class soapy.confParse.**ReconstructorConfig**(*N=None*)
Bases: soapy.confParse.ConfigObj

Configuration parameters describing the reconstructor that will be used to calculate DM commands from WFS measurements. The type must be an object in the soapy.reconstruction module. Other parameters may be specific to this reconstructor

Optional:

Parameter	Description	Default
type	string: Type of reconstructor to use. Must be a class in reconstruction module.	MVM
svdConditioning	float: Conditioning parameter to be using in Least Squares reconstructor inversion SVD to cut off unwanted DM modes. See numpy.linalg.pinv for details about the inversion.	0
gain	float: Gain of the integrator loop.	0.6
imat_noise	bool: include WFS noise when making in interaction matrix	True

allowedAttrs = ['N', 'type', 'svdConditioning', 'gain', 'imat_noise']

calculatedParams = []

optionalParams = [('type', 'MVM'), ('svdConditioning', 0.0), ('gain', 0.6), ('imat_noise', True)]

p = ('imat_noise', True)

requiredParams = []

class soapy.confParse.**SciConfig**(*N=None*)
Bases: soapy.confParse.ConfigObj

Configuration parameters characterising Science Cameras.

These should be held in the Science of the parameter file. Each Science target is created seperately with an integer index. Any entries above sim.nSci will be ignored.

Required:

Parameter	Description
`position`	tuple: The position of the science camera in the field in arc-seconds
`FOV`	float: The field of fiew of the science detector in arc-seconds
`wavelength`	float: The wavelength of the science detector light
`pxls`	int: Number of pixels in the science detector

Optional:

Parameter	Description	Default
`pxlScale`	float: Pixel scale of science camera, in arcseconds. If set, overwrites `FOV`.	`None`
`type`	string: Type of science camera This must the name of a class in the `SCI` module.	`PSF`
`fftOversamp`	int: Multiplied by the number of of phase points required for FOV to increase fidelity from FFT.	`2`
`fftwThreads`	int: number of threads for fftw to use. If `0`, will use system processor number.	`1`
`fftwFlag`	str: Flag to pass to FFTW when preparing plan.	`FFTW_MEASURE`
`height`	float: Altitude of the object. 0 denotes infinity.	`0`
`propagationMode`	str: Mode of light propogation from object. Can be "Physical" or "Geometric".	`"Geometric"`
`instStrehlWithTT`	bool: Whether or not to include tip/tilt in instantaneous Strehl calculations.	`False`

allowedAttrs = ['position', 'wavelength', 'pxls', 'N', 'pxlScale', 'FOV', 'type', 'fftOversamp', 'fftwFlag', 'fftwThreads

calcParams()

calculatedParams = []

optionalParams = [('pxlScale', None), ('FOV', None), ('type', 'PSF'), ('fftOversamp', 2), ('fftwFlag', 'FFTW_MEASU

p = ('propagationMode', 'Geometric')

requiredParams = ['position', 'wavelength', 'pxls']

class soapy.confParse.**SimConfig**(*N=None*)

Bases: soapy.confParse.ConfigObj

Configuration parameters relavent for the entire simulation. These should be held at the beginning of the parameter file with no indendation.

Required:

Parameter	Description
`pupilSize`	int: Number of phase points across the simulation pupil
`nIters`	int: Number of iteration to run simulation
`loopTime`	float: Time between simulation frames (1/framerate)

Optional:

Parameter	Description	Default
nGS	int: Number of Guide Stars and WFS	0
nDM	int: Number of deformable Mirrors	0
nSci	int: Number of Science Cameras	0
reconstructor	str: name of reconstructor class to use. See reconstructor module for available reconstructors.	"MVM"
simName	str: directory name to store simulation data	None
wfsMP	bool: Each WFS uses its own process	False
verbosity	int: debug output for the simulation ranging from 0 (no-ouput) to 3 (all debug output)	2
logfile	str: name of file to store logging data,	None
learnIters	int: Number of *learn* iterations for Learn & Apply reconstructor	0
learnAtmos	str: if random, then random phase screens used for *learn*	random
simOversize	float: The fraction to pad the pupil size with to reduce edge effects	1.2
loopDelay	int: loop delay in integer count of loopTime	0
threads	int: Number of threads to use for multithreaded operations	1
photometric_zp	float: Photometric zeropoint - number of photons/meter/second from a magnitude 0 star	2e9

Data Saving (all default to False):

Parameter	Description
saveSlopes	Save all WFS slopes. Accessed from sim with sim.allSlopes
saveDmCommands	Saves all DM Commands. Accessed from sim with sim.allDmCom
saveWfsFrames	Saves all WFS pixel data. Saves to disk a after every frame to avoid u much memory
saveStrehl	Saves the science camera Strehl Ratio. Accessed from sim with sim.longStrehl and sim.instStrehl
saveWfe	Saves the science camera wave front error. Accessed from sim with s
saveSciPsf	Saves the science PSF.
saveInstPsf	Saves the instantenous science PSF.
saveInstScieField	Saves the instantaneous electric field at focal plane.
saveSciRes	Save Science residual phase

allowedAttrs = ['pupilSize', 'nIters', 'loopTime', 'pxlScale', 'simPad', 'simSize', 'scrnSize', 'totalWfsData', 'totalActs

calculatedParams = ['pxlScale', 'simPad', 'simSize', 'scrnSize', 'totalWfsData', 'totalActs', 'saveHeader']

optionalParams = [('nGS', 0), ('nDM', 0), ('nSci', 0), ('gain', 0.6), ('reconstructor', 'MVM'), ('simName', None), ('sav

p = ('photometric_zp', 2000000000.0)

requiredParams = ['pupilSize', 'nIters', 'loopTime']

class soapy.confParse.**TelConfig**(*N=None*)

Bases: soapy.confParse.ConfigObj

Configuration parameters characterising the Telescope. These should be held in the Telescope group in the parameter file.

Required:

Parameter	Description
telDiam	float: Diameter of telescope pupil in metres

Optional:

Parameter	Description	Default
obsDiam	float: Diameter of central obscuration	0
mask	str: Shape of pupil (only accepts circle currently)	circle

allowedAttrs = ['telDiam', 'N', 'obsDiam', 'mask']

calculatedParams = []

optionalParams = [('obsDiam', 0), ('mask', 'circle')]

p = ('mask', 'circle')

requiredParams = ['telDiam']

class soapy.confParse.**WfsConfig**(*N=None*)
Bases: soapy.confParse.ConfigObj

Configuration parameters characterising Wave-front Sensors. These should be held in the WFS group in the parameter file. Each WFS is specified by first specifying an index, then the WFS parameters. Any entries above sim.nGS will be ignored.

Required:

Parameter	Description
GSPosition	tuple: position of GS on-sky in arc-secs
wavelength	float: wavelength of GS light in metres
nxSubaps	int: number of SH sub-apertures

Optional:

Parameter	Description	Default
`type`	string: Which WFS object to load from WFS.py?	`ShackHartmann`
`GSMag`	float: Apparent magnitude of the guide star	`0`
`photonNoise`	bool: Include photon (shot) noise.	`False`
`eReadNoise`	float: Electrons of read noise	`0`
`throughput`	float: Throughput of the entire optical and electronic system from guide star photons to recorded WFS detector counts. Includes atmospheric effects, the optical train and detector gain.	`1.`
`propagationMode`	string: Mode of light propogation from GS. Can be "Physical" or "Geometric"**.	`"Geometric"`
`subapFieldStop`	bool: if True, add a field stop to the wfs to prevent spots wandering into adjacent sub-apertures. if False, oversample subap FOV by a factor of 2 to allow into adjacent subaps.	`False`
`removeTT`	bool: if True, remove TT signal from WFS slopes before reconstruction.**	`False`
`fftOversamp`	int: Multiplied by the number of of phase points required for FOV to increase fidelity from FFT.	`3`
`GSHeight`	float: Height of GS beacon. `0` if at infinity.	`0`
`subapThreshold`	float: How full should subap be to be used for wavefront sensing?	`0.5`
`lgs`	bool: is WFS an LGS?	`False`
`centMethod`	string: Method used for Centroiding. Can be `centreOfGravity`, `brightestPxl`, or `correlation`.**	`centreOfGravity`
`referenceImage`	array: Reference images used in the correlation centroider. Full image plane image, each subap has a separate reference image	`None`
`angleEquivNoise`	float: width of gaussian noise added to slopes measurements in arc-secs	`0`
`centThreshold`	float: Centroiding threshold as a fraction of the max subap value.**	`0.1`
`exposureTime`	float: Exposure time of the WFS camera - must be higher than loopTime. If None, will be set to loopTime.	`None`
`wvlBandWidth`	float: Width of wavelength band sent to WFS in nm	`100`
`extendedObject`	ndarray or str: The object used as extended source for WFS, of size 2*fftOversamp*pxlsPerSubap. The FOV of the object should be twice the FOV of the sub-aperture.	`None`
`fftwThreads`	int: number of threads for fftw to use. If `0`, will use system processor number.	`1`
`fftwFlag`	str: Flag to pass to FFTW when preparing plan.	`FFTW_PATIENT`
`pxlsPerSubap`	int: number of pixels per sub-apertures	`10`
`subapFOV`	float: Field of View of sub-aperture in arc-secs	`5`
`correlationPadding`	int: Padding for correlation WFS	`None`
`nx_guard_pixels`	int: Guard Pixels between Shack-Hartmann sub-apertures (Not currently operational)	`0`

allowedAttrs = ['GSPosition', 'wavelength', 'nxSubaps', 'position', 'pxlsPerSubap2', 'dataStart', 'lgs', 'N', 'propagati

calcParams()

calculatedParams = ['position', 'pxlsPerSubap2', 'dataStart', 'lgs']

optionalParams = [('propagationMode', 'Geometric'), ('fftwThreads', 1), ('fftwFlag', 'FFTW_PATIENT'), ('angleEqu

p = ('nx_guard_pixels', 0)

requiredParams = ['GSPosition', 'wavelength', 'nxSubaps']

class soapy.confParse.**YAML_Configurator**(*filename*)

Bases: soapy.confParse.PY_Configurator

loadSimParams()

readfile()

soapy.confParse.**loadSoapyConfig**(*configfile*)

soapy.confParse.**test**()

CHAPTER 17

Indices and tables

- genindex
- modindex
- search

Python Module Index

S

Index

Z