OOPAO: Object Oriented Python Adaptive Optics

C.-T. Héritier[a,b,c*], C. Vérinaud[c], and C. Correia[d]

[a]DOTA, ONERA, F-13661 Salon cedex Air - France
[b]Aix Marseille Univ, CNRS, CNES, LAM, Marseille, France
[c]European Southern Observatory (ESO), Karl Schwarzschild-str 2, D-85748 Garching-bei-Muenchen Germany
[d]Space ODT - Optical Deblurring Technologies Lda, Porto, Portugal

ABSTRACT

The list of Adaptive Optics (AO) simulators in the community has constantly been growing, guided by different needs and purposes (Compass, HCIPY, OOMAO, SOAPY, YAO...). In this paper, we present OOPAO (Object Oriented Python Adaptive Optics), a simulation tool based on the Matlab distribution OOMAO to adapt its philosophy to the Python language. This code was initially intended for internal use but the choice was made to make it public as it can benefit the community since it is fully developed in Python. The OOPAO repository is available in free access on GitHub (https://github.com/cheritier/OOPAO) with several tutorials. The tool consists of a full end-to-end simulator designed for AO analysis purposes. The principle is that the light from a given light source can be propagated through multiple objects (Atmosphere, Telescope, Deformable Mirror, Wave-Front Sensors...) among which experimental features can be input, in the spirit of OOMAO. This paper provides an overview of the main capabilities of the code and can be used as a user manual for interested users.

Keywords: Adaptive Optics ; Python, Numerical modelling

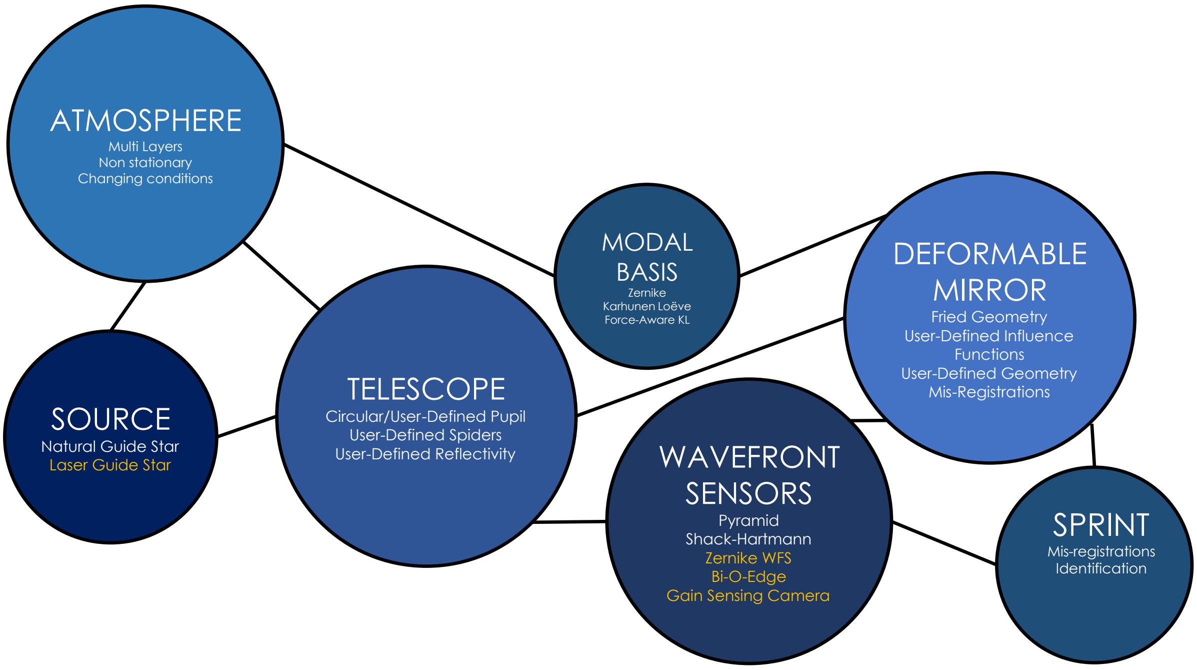
1. INTRODUCTION

OOPAO is a Python tool developed to perform Adaptive Optics simulation using a similar philosophy as OOMAO. The motivation of the development of OOPAO is to provide an open-source tool to the community that completes the long list of existing simulation tools [5, 8, 3, 1, 7, 4, 6]. The purpose of this paper is to provide a user manual to describe the philosophy of the code. The documentation of the code provides more details if required.

The source code of OOPAO is available on GitHub https://github.com/cheritier/OOPAO. This page provides all the informations necessary to install the required packages.

```
import matplotlib.pyplot as plt
import numpy as np
import OOPAO
```

*At the time of this work development
Further author information:
C. T. Heritier, E-mail: cedric.heritier-salama@onera.fr & cedric-taissir.heritier@lam.fr



Summary of the main features of OOPAO. The yellow font indicates features under development.

2. THE SOURCE OBJECT

The source object carries the wavelength information as well as the flux information It is initialized specifying the optical band and the magnitude of the star.

```
from OOPAO.Source import Source
# to create a natural guide star in I band of magnitude 5
ngs = Source(magnitude = 5,
             optBand   ='I')
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% SOURCE %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
 Source Wavelength Zenith  Azimuth   Altitude Magnitude    Flux
           [m]     [arcsec]  [deg]      [m]             [phot/m2/s]
--------------------------------------------------------------------
  NGS     7.9e-07     0         0         inf        5.0    73369565.2
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

The optical bands are defined within the `Source` object inside the `photometry` method. The default photometry have been imported from the OOMAO toolkit. You can edit this function to add your own optical bands.

```
                   [ Wavelength , Bandwidth, Zero-Point ]
        phot.U      = [ 0.360e-6 , 0.070e-6 , 2.0e12 ]
        phot.V      = [ 0.550e-6 , 0.090e-6 , 3.3e12 ]
        ...
```

You can access a few properties of the `ngs` object:

```
print('The source Wavelenght is '+str(ngs.wavelength) + ' [m] ')
print('The source Magnitude is '+str(ngs.magnitude))
print('The source Altitude is '+str(ngs.altitude) + ' [m] ')
print('The source Coordinates are '+str(ngs.coordinates) + ' [arcsec,deg] ')
print('The source Flux is '+str(ngs.nPhoton) + ' [phot/m2/s] ')
print('The source Type is '+str(ngs.type))
print('The source Tag is '+str(ngs.tag))
```

The main properties of an OOPAO class can be displayed simply entering its name in a terminal.

3. THE TELESCOPE OBJECT

You can import the Telescope class from OOPAO using:

```
from OOPAO.Telescope import Telescope
```

The Telescope is a central object in OOPAO, it is required to initialize many of the OOPAO classes as it carries the pupil definition and pixel size. A Source object is associated to the Telescope that carries the flux and wavelength information. An Atmosphere object can be paired to the Telescope to propagate the light through turbulent phase screens. To initialize a Telescope object, you need to specify a few parameters, the diameter and the resolution of the telescope. The pixel size associated will define the pixel size of the phase-screens. The sampling of the atmospheric phase screens is driven by different factors:

- **Sampling of the Wave-Front Sensor sub-apertures:** the WFS object requires a resolution that is `n` times the number of subaperture where `n` should be an even number >= 4.
- **Sampling of the turbulence:** It is recommanded to use at least 3 pixels per r_0, the Fried Parameter expressed at the wavelength of interest.
- **Sampling of the influence functions of the Deformable Mirror:** Two pixels per influence functions is typically not enough and could lead to numerical errors when trying to apply a shape on the Deformable Mirror.

```
# telescope parameters
sensing_wavelength = ngs.wavelength      # sensing wavelength of the WFS
n_subaperture      = 20                  # number of subap accross the diameter
diameter           = 8                   # diameter of the phase screens in [m]
resolution         = n_subaperture*8     # resolution of the phase screens in pix
pixel_size         = diameter/resolution # size of the pixels in [m]
obs_ratio          = 0.1                 # central obstruction ratio
sampling_time      = 1/1000              # sampling time of the AO loop in [s]
# initialize the telescope object
tel = Telescope(diameter          = diameter,
               resolution         = resolution,
               centralObstruction = obs_ratio,
               samplingTime       = sampling_time)
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% TELESCOPE %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
     Diameter                8                 [m]
    Resolution              160              [pixels]
    Pixel Size              0.05               [m]
     Surface                50.0               [m2]
Central Obstruction          10.0          [% of diameter]
Pixels in the pupil         19900             [pixels]
      Source                None
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
No light propagated through the telescope
```

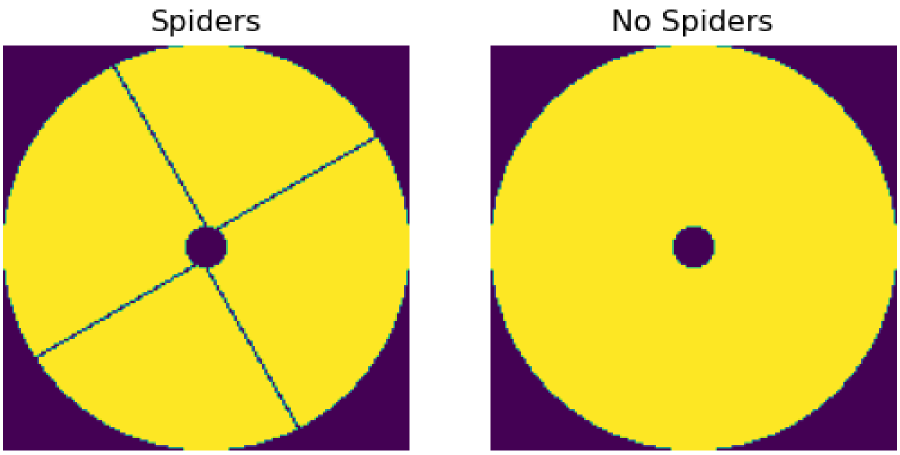
We can notice that, as indicated, so far no light is propagated through the telescope. See the section **Propagating light to the Telescope** A few properties of the `tel` object can be accessed: * `tel.pupil` The pupil mask of the telescope * `tel.pixelArea` The number of valid pixels in the pupil * `tel.pupilReflectivity` User-defined pupil reflectivity (default: uniform)

Methods

Apply spider to the pupils You can defin a list of angle for each spider of the telescope. A spider is defined from the center to the edge of the pupil support. If necessqry, a shift offset can be applied to the spiders.

```
n_spider = 4
spider_angle = np.linspace(0+30,360+30,n_spider,endpoint=False)
tel.apply_spiders(spider_angle,thickness_spider=0.05,
   offset_X=[0.15,-0.15,-0.15,0.15],
   offset_Y=[0.15,0.15,-0.15,-0.15])
pupil_spiders =  tel.pupil.copy()
# reset to initial pupil
tel.apply_spiders(angle = [0],thickness_spider=0)
plt.subplot(1,2,1),plt.imshow(pupil_spiders),plt.axis('off'),plt.title('Spiders')
plt.subplot(1,2,2),plt.imshow(tel.pupil),plt.axis('off'),plt.title('No Spiders');
```

```
Warning!: A new pupil is now considered, its reflectivity is considered to be
uniform. Assign the proper reflectivity map to tel.pupilReflectivity if
required.
```



Left: `tel.pupil` applying spiders, Right: `tel.pupil` after initialization.

Propagating light to the Telescope

Coupling a Source to the Telescope To propagate the light of the Source through the Telescope, it is necessary to attach the `ngs` object to the `tel` object using the `*` operator.

```
ngs*tel
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% TELESCOPE %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
     Diameter               8                 [m]
    Resolution             160              [pixels]
    Pixel Size             0.05               [m]
     Surface               50.0               [m2]
Central Obstruction         10.0         [% of diameter]
Pixels in the pupil        19900             [pixels]
    Source NGS            790.0               [nm]
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
NGS(I) ~~> telescope
```

This time, the bottom part indicated that the NGS has been properly propagated to the telescope. The optical path can be displayed using the `print_optical_path()` method

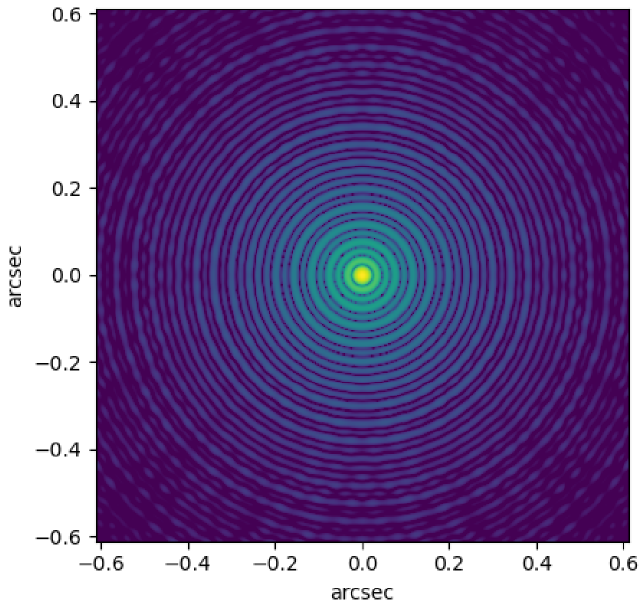
```
tel.print_optical_path()
```

```
NGS(I) ~~> telescope
```

The `tel` object now has an `src` attribute that corresponds to the `ngs` object and the flux properties are now adapted to the pupil of the telescope. The `tel.src` has a `fluxMap` property that displays a 2D map of the **number of photons per pixel per temporal iteration** The Optical Path Difference in [m] of the wave-front can be accessed using the `tel.OPD` and the phase in [rad] wave-front can be accessed using the `ngs.phase` or the `tel.src.phase`

Compute the PSF of the system

```
# set back the reflectivity 1
tel.pupilReflectivity = tel.pupil.astype(float)
ngs*tel

# compute PSF method
zeroPaddingFactor = 6
tel.computePSF(zeroPaddingFactor=zeroPaddingFactor)

# get current pixel scale
size_pixel_arcsec = 206265*(tel.src.wavelength/tel.D)/zeroPaddingFactor
N                 = 300
normalized_PSF    = tel.PSF_norma[N:-N,N:-N]# crop to zoom on the center
fov               = normalized_PSF.shape[0]*size_pixel_arcsec # get the corresponding fov
plt.imshow(np.log10(normalized_PSF),extent=[-fov/2,fov/2,-fov/2,fov/2])
plt.clim([-6,0]),plt.xlabel('arcsec'),plt.ylabel('arcsec');
```



Normalized Point Spread Function obtained using the `tel.computePSF()` method.
The PSF is provided at the `tel.src` wavelength and the pixel scale depends on the `zeroPaddingFactor`.

4. THE ATMOSPHERE OBJECT

You can import the Atmosphere class from OOPAO using:

```
from OOPAO.Atmosphere import Atmosphere
```

An atmosphere is made of one or several layer of turbulence that follow the Van Karmann statistics. The phase-screens evolution are computed using the method developed in [2] to efficiently generate non stationary phase screens. Each layer is considered to be independent to the other ones and has its own properties (direction, speed, etc.). The Atmosphere object can be defined for a single Source object (default) or multi Source Object (see Asterism class). The Source coordinates allow to span different areas in the field (defined as well by `tel.fov`). If the source type is an LGS the cone effect is considered using an interpolation. NGS and LGS can be combined together in the Asterism object. The convention chosen is that all the wavelength-dependant atmosphere parameters are expressed at 500 nm.

```
# setting a two layers atmosphere
atm = Atmosphere(telescope      = tel,
                 r0             = 0.15,# Fried parameter @500 nm in [m]
                 L0             = 25,# Outer scale in [m]
                 fractionalR0   = [0.7, 0.3  ],# Cn2 profile
                 altitude       = [0  , 10000],
                 windDirection  = [0  , 20   ],
                 windSpeed      = [5  , 10   ])
```

The atmosphere has to be initialized using the `atm.initializeAtmosphere()` command

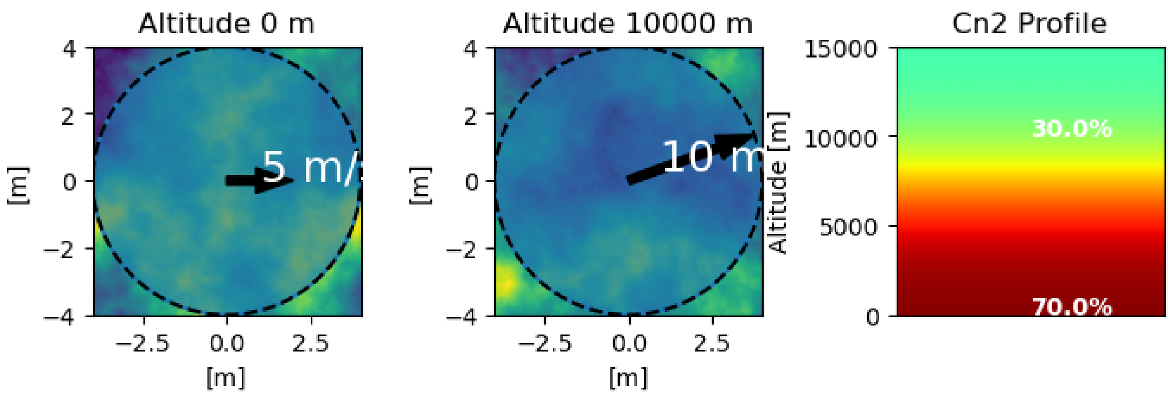
```
atm.initializeAtmosphere(telescope=tel)
```

```
Creation of layer1/2 ...
-> Computing the initial phase screen...
initial phase screen : 0.03470635414123535 s
ZZt.. : 1.593867301940918 s
ZXt.. : 0.9020891189575195 s
XXt.. : 0.4502260684967041 s
Done!
Creation of layer2/2 ...
-> Computing the initial phase screen...
initial phase screen : 0.01602458953857422 s
ZZt.. : 0.0 s
ZXt.. : 0.0 s
XXt.. : 0.0 s
```

```
SCAO system considered: covariance matrices were already computed!
Done!
%%%%%%%%%%%%%%%%%%%%%%%%%%%%% ATMOSPHERE %%%%%%%%%%%%%%%%%%%%%%%%%%%%%
   Layer      Direction      Speed      Altitude      Cn2
                [deg]        [m/s]        [m]       [m-2/3]
----------------------------------------------------------------------
     1           0            5            0          0.7
------------------------------------------------------------------
     2           20           10         10000        0.3
------------------------------------------------------------------
    r0 @500 nm          0.15 [m]
        L0               25 [m]
  Seeing @500nm         0.69 ["]
    Frequency         1000.0 [Hz]
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

The atm layers can be displayed using the `atm.display_atm_layers()` command

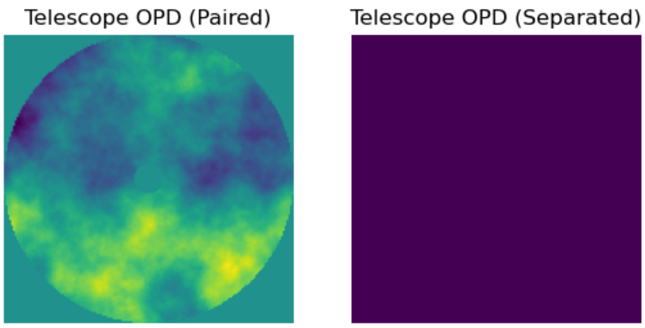
```
atm.display_atm_layers()
```



The atmosphere `atm` can be combined to the telescope using the `+`. In that case, the OPD of the telescope is copied from the atmosphere OPD:

```
tel+atm
OPD_tel_paired = tel.OPD.copy()
tel-atm
OPD_tel_separated = tel.OPD.copy()
plt.subplot(1,2,1),plt.imshow(OPD_tel_paired),plt.axis('off'),
plt.title('Telescope OPD (Paired)')
plt.subplot(1,2,2),plt.imshow(OPD_tel_separated),plt.axis('off'),
plt.title('Telescope OPD (Separated)');
```

```
Telescope and Atmosphere combined!
Telescope and Atmosphere separated!
```



Temporal evolution of the phase screen

By default, the phase screens are generated based on a random seed value. The evolution of the phase screens will also follow a given trajectory defined by a random vector associated to another seed value.

To update the phase screens of one time step, defined by `tel.samplingTime`, the following command must be used:

```
atm.update()
```

This operation will trigger a shift of each `atm.layer` of a value dictated by the layer speed and direction. If the `tel` and `atm` objects are combined, this command will trigger an update of the `tel.OPD` to copy the new `atm.OPD`.

Generating a new initial phase screen

To generate a different phase screen it is possible to use the `atm.generateNewPhaseScreen` method, specifying a given seed value:

```
atm.generateNewPhaseScreen(seed=5)
```

Updating the atmosphere properties on the fly

Most of the Atomosphere properties can be updated on the fly (`atm.r0, atm.windSpeed,atm.windDirection`). In this case, only the new pixels coming inside the pupil will follow the new statistics. If you want to re-initialize the phase screen with the updated statistics, it is necessary to use the `atm.generateNewPhaseScreen` command.

5. THE DEFORMABLE MIRROR OBJECT

The deformable mirror is mainly characterized with its influence functions. They can be user-defined and loaded in the model but the default case is a cartesian DM with gaussian influence functions and normalized to 1 m. The DM is always defined in the pupil plane and can be conjugated to different altitude.

Fried Geometry

By default, the Deformable Mirror geometry follows the Fried's definition where each actuator is located at the corner of 4 WFS sub-aperture. The number of sub-aperture is then a required parameter of the Deformable Mirror class.

```
from OOPAO.DeformableMirror import DeformableMirror
mechanical_coupling = 0.45

dm_fried = DeformableMirror(  telescope    = tel,
                        nSubap       = n_subaperture, # by default Fried Geometry
                        mechCoupling = mechanical_coupling)
```

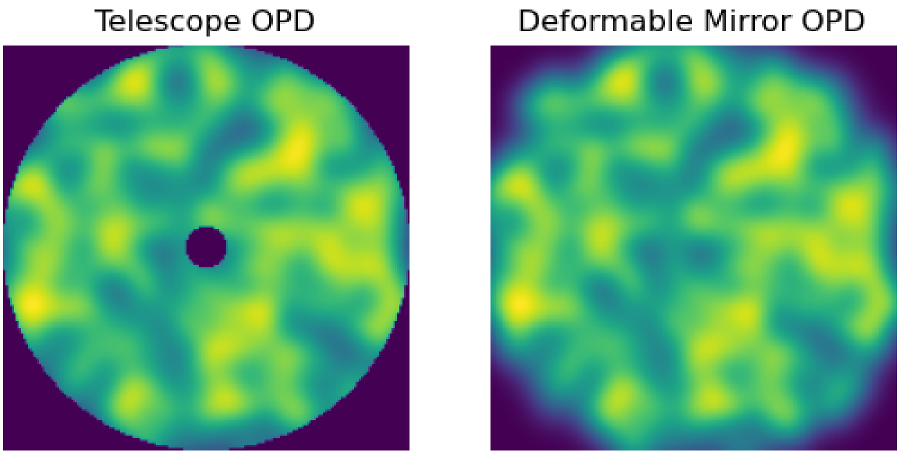
```
No coordinates loaded.. taking the cartesian geometry as a default
Generating a Deformable Mirror:
Computing the 2D zonal modes...
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% DEFORMABLE MIRROR %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Controlled Actuators        356
         M4                False
        Pitch               0.4              [m]
 Mechanical Coupling        0.45             [%]
-------------------------------------------------------------------------------
Mis-registration:
Rotation [deg]  Shift X [m]     Shift Y [m]     Radial Scaling [%]
Tangential Scaling [%]
      0              0               0                  0
0
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

The DM commands can be applied as a linear combination of the influence functions `dm.modes` setting the `dm.coefs` property:

```
dm_fried.coefs = np.random.rand(dm_fried.nValidAct)
```

The light can be propagated through the telescope, passing by the deformable mirror using the `*` operator:

```
ngs*tel*dm_fried
plt.subplot(1,2,1),plt.imshow(tel.OPD),plt.axis('off'),plt.title('Telescope OPD')
plt.subplot(1,2,2),plt.imshow(dm_fried.OPD),plt.axis('off'),
plt.title('Deformable Mirror OPD');
```



Mis-Registration

A mis-registration can be input when initialising the DM object, only if the default influence functions are considered (gaussian). To apply mis-registration to an existing objet with user-defined influence functions, consider using the more complete`applyMisRegistration` function available in `OOPAO/mis_registration_identification_algorithm/`.

By default, the Deformable Mirror geometry follows the Fried's definition where each actuator is located at the corner of 4 WFS sub-aperture. The number of sub-aperture is then a required parameter of the Deformable Mirror class.

```
from OOPAO.MisRegistration import MisRegistration
m = MisRegistration()
m.shiftX = 0.05*tel.D/n_subaperture
m.shiftY = 0.1*tel.D/n_subaperture
m.rotationAngle = 15
m.radialScaling = 0.1
m.tangentialScaling = 0
m.anamorphosisAngle = 45

dm_mis_registered = DeformableMirror(  telescope    = tel,
                        nSubap       = n_subaperture, # by default Fried Geometry
                        mechCoupling = mechanical_coupling,
                        misReg       = m)
```

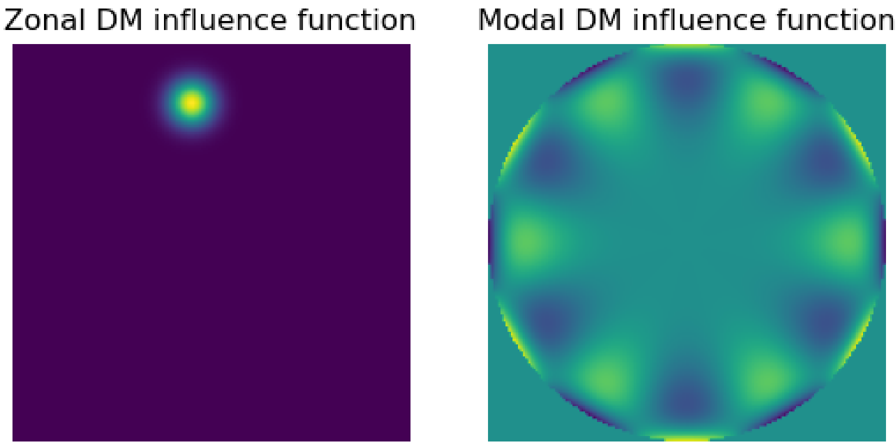
User-defined geometry

```
# create a circular DM
n_ring = 11; n_start = 5; cx = []; cy = []
for i in range(1,n_ring+1):
    r = i*(tel.D/2)/(n_ring)
    theta = np.linspace(0,2*np.pi,n_start,endpoint=False)
    cx = np.hstack((cx,(r*np.cos(theta))))
    cy = np.hstack((cy,(r*np.sin(theta))))
    n_start += 6
# input coordinates must be provided in a 2D array [nAct,2]
circular_coordinates = np.vstack((cx,cy)).T
dm_circular = DeformableMirror(  telescope    = tel,
                        nSubap       = n_subaperture, # by default Fried Geometry
                        mechCoupling = mechanical_coupling,
                        coordinates=circular_coordinates)
```

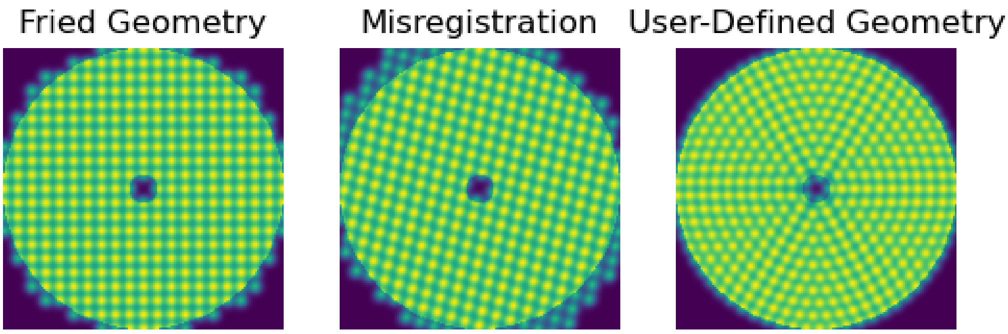
User-Defined influence functions

It is possible to input the influence functions of the DM directly. In this example, we consider a modal DM that corresponds to zernike polynomials.

```
from OOPAO.Zernike import Zernike
Z = Zernike(tel,300) # create a Zernike object considering 300 polynomials
Z.computeZernike(tel) # compute the Zernike
Z_2D = Z.modesFullRes.reshape((tel.resolution**2,Z.nModes))# reshape in 2D
dm_Zernike = DeformableMirror( telescope = tel,
                               nSubap    = n_subaperture, # required
                               modes     = Z_2D)
```



We can display the cube of the influence functions to display the position of the actuators with respect to the pupil.



6. WAVE-FRONT SENSORS OBJECTS

Pyramid WFS

The pyramid object consists mainly in defining the PWFS phase mask to apply the filtering of the electro-magnetic field. Many parameters can allow to tune the pyramid model:

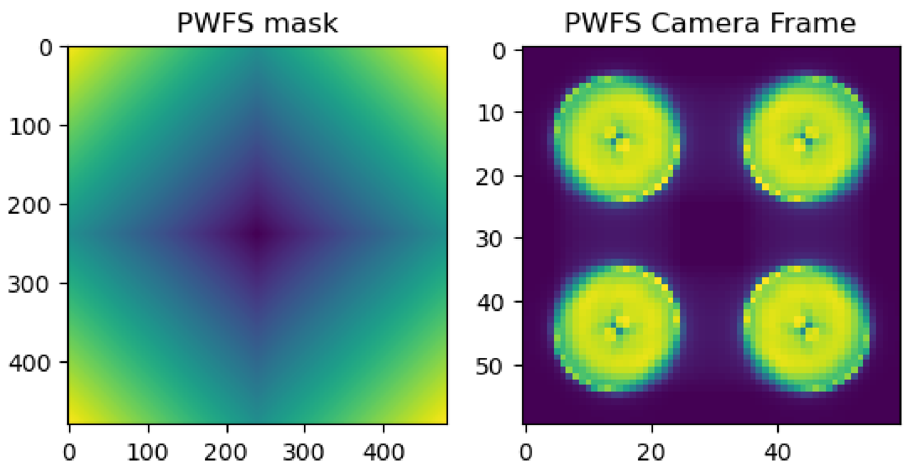
- Number of pixels `nSubap` along the telescope pupil support. `tel.resolution` should be a multiple of `nSubap` and >=4
- Cicular modulation radius in λ/D `modulation`. By default the number of modulation points ensures to have one point every λ/D on the circular trajectory but this sampling can be modified by the user. The number of modulation points is a multiple of 4 to ensure that each quadrant has the same number of modulation points. A user-defined modulation path can be input.
- The type of post-processing of the PWFS signals (slopes-maps, full-frame,etc). To be independent from this choice, the pyramid signals are named `wfs.signal_2D` for either the Slopes-Maps or the camera frame and `signal` for the signal reduced to the valid pixels considered.
- The intensity threshold to select the valid pixel
- Centering of the mask and of the FFT on 1 or 4 pixels `psfCentering`
- Number of pixels separating the PWFS pupils `n_pix_separation`
- Number of pixels on the edge of the Pyramid pupils`n_pix_edge`
- The modulation value for the calibration and selection of the valid pixels `calibModulation`

In addition, the Pyramid object has a Detector object as a child-class that provides the pyramid signals. It can be accessed through `pwfs.cam`

Different properties can be user-defined (modulation path, valid pixel map, number of modulation point, etc.). More details are provided in the description of the pyramid class.

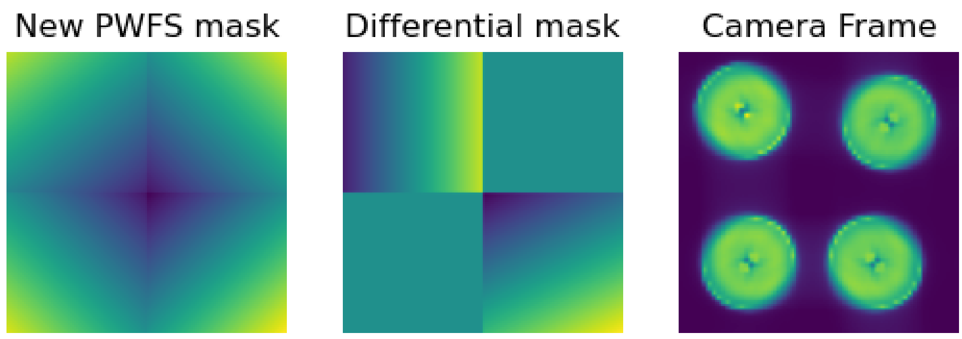
```python
from OOPAO.Pyramid import Pyramid

pwfs = Pyramid(telescope         = tel,
               nSubap            = n_subaperture,
               modulation        = 3,
               lightRatio        = 0.1,
               postProcessing    = 'slopesMaps',
               n_pix_separation = 10,
               n_pix_edge        = 5)
plt.figure(),
plt.subplot(1,2,1),plt.imshow(pwfs.m),plt.title('PWFS mask') # 2D phase mask
plt.subplot(1,2,2),plt.imshow(pwfs.cam.frame),plt.title('PWFS Camera Frame');
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% PYRAMID WFS %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
 Pupils Diameter          20              [pixels]
Pupils Separation         10              [pixels]
       FoV               3.26             [arcsec]
  TT Modulation           3              [lamda/D]
PSF Core Sampling         4              [pixel(s)]
   Valid Pixels          664             [pixel(s)]
Signal Computation    slopesMaps
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```



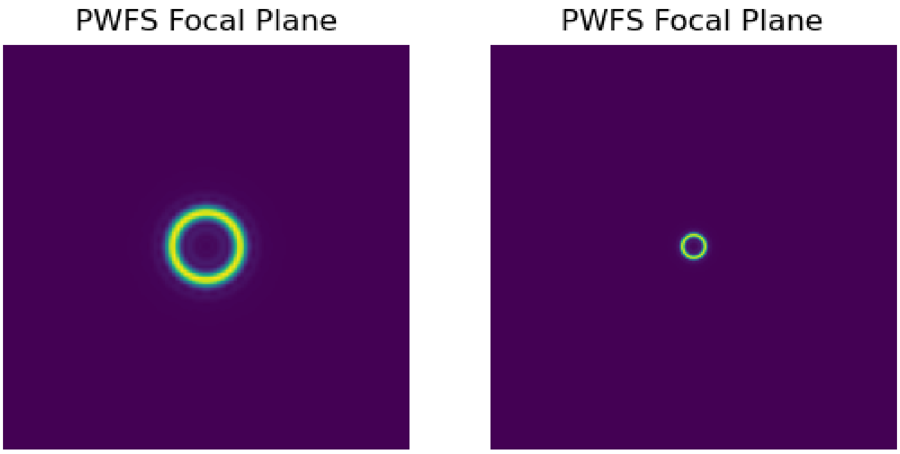
It is possible to shift the Pyramid pupils using the `apply_shift_wfs` method by applying an additional Tip/Tilt on each quadrant of the phase mask

```python
pwfs.apply_shift_wfs(sx=[5,0,2,0],sy=[0,0,4,0])
```



The focal plane electric field is accessible through the `modulation_camera_em` property. This is used by the `get_modulation_frame` method to compute the intensity in at the focal plane, afte the Tip/Tilt modulation mirror. The `radius` allows to select the field of view.

```
plt.figure(),
plt.subplot(1,2,1),plt.imshow(pwfs.get_modulation_frame(radius = 6)),
plt.axis('off'), plt.title('PWFS Focal Plane'); # radius in lambda/D
plt.subplot(1,2,2),plt.imshow(pwfs.get_modulation_frame(radius = 18)),
plt.axis('off'), plt.title('PWFS Focal Plane'); # radius in lambda/D
```



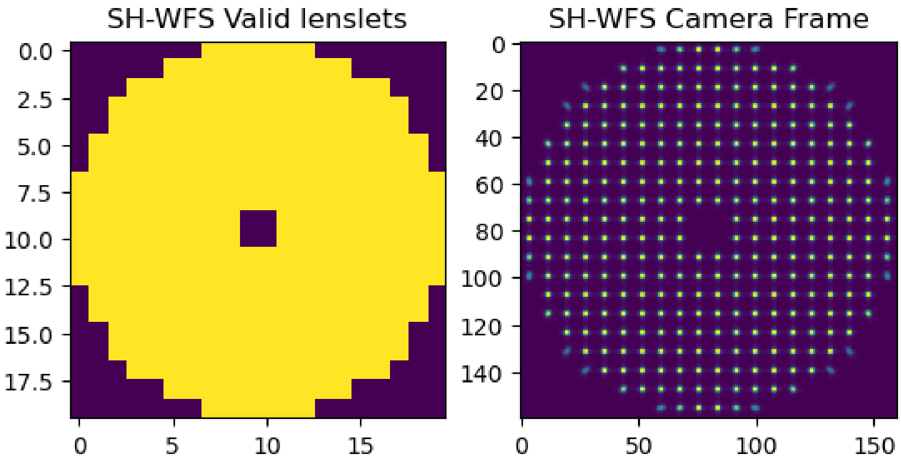
Shack Hartmann WFS

A Shack Hartmann object consists in defining a 2D grid of lenslet arrays located in the pupil plane of the telescope to estimate the local tip/tilt seen by each lenslet. By default the Shack Hartmann detector is considered to be noise-free (for calibration purposes). These properties can be switched on and off on the fly (see properties) It requires the following parameters:

- The number of lenslets `nSubap` accross the support of the telescope pupil
- The intensity threshold `lightRatio` to select the valid lenslets
- The flag `is_geometric` to switch from diffractive to geometric Shack Hartmann
- `shannon_sampling` to sample the diffractive spots with 2 pixels per lambd/D (`True`) or 1 pixel per lambda/D (`False`)
- theshold to compute the center of gravity `threshold_cog` with respect to the maximum value of the image

```
from OOPAO.ShackHartmann import ShackHartmann
shwfs = ShackHartmann(telescope    = tel,
              nSubap              = n_subaperture,
              lightRatio          = 0.5,
              is_geometric        = False,
              shannon_sampling    = True,
              threshold_cog       = 0.1)
plt.figure(),
plt.subplot(1,2,1),plt.imshow(shwfs.valid_subapertures),
plt.title('SH-WFS Valid lenslets') # 2D phase mask
plt.subplot(1,2,2),plt.imshow(shwfs.cam.frame),
plt.title('SH-WFS Camera Frame'); # camera detector frame
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% SHACK HARTMANN WFS %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    Subapertures           20
  Subaperture Size         0.4               [m]
     Pixel FoV             0.2             [arcsec]
   Subapertue FoV          1.63            [arcsec]
 Valid Subaperture         312
   Binning Factor           1
   Geometric WFS          False
  Shannon Sampling         True
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```



it is possible to switch from geometric to diffractive shack hartmann simply by setting `wfs.is_geometric` to `True` or `False`

```python
# make sure that the telescope OPD is initiliazed
tel.resetOPD()
ngs*tel*shwfs
shwfs.is_geometric = True
```

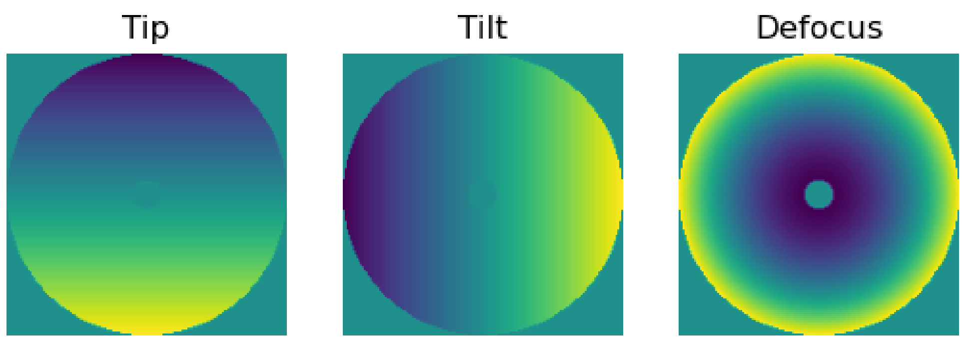
```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% SHACK HARTMANN WFS %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    Subapertures             20
  Subaperture Size          0.4               [m]
      Pixel FoV             0.2             [arcsec]
   Subapertue FoV           1.63            [arcsec]
 Valid Subaperture          312
   Binning Factor            1
   Geometric WFS            True
  Shannon Sampling          True
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

7. MODAL BASIS COMPUTATION

A few modal basis or available in OOPAO: Zernike and KL Modes.

Zernike Polynomials

```python
from OOPAO.Zernike import Zernike
Z = Zernike(tel,9)      # first initialize the Zernike object
Z.computeZernike(tel)   # compute the Zernike for the desired telescope
plt.figure()
for i in range(3):
    plt.subplot(1,3,i+1)
    plt.imshow(Z.modesFullRes[:,:,i]),plt.axis('off'), plt.title(Z.modeName(i));
```

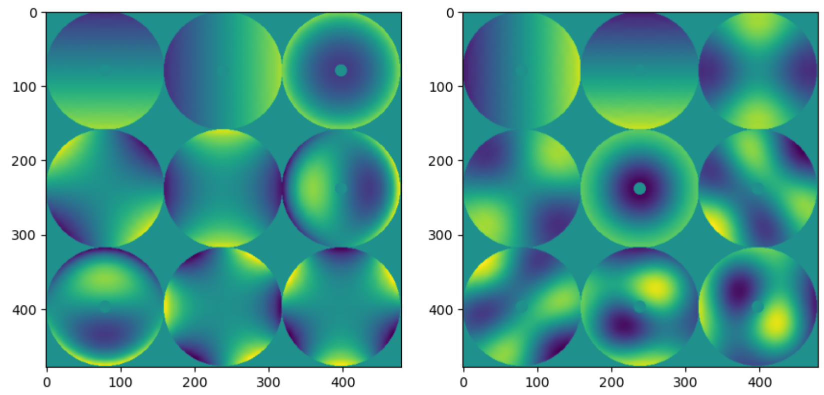


The Zernike can be projected on the `dm`using the dm influence functions `modes` from the `dm` object

```python
M2C_zernike = np.linalg.pinv(np.squeeze(dm_fried.modes[tel.pupilLogical,:]))@Z.modes
```

It is possible to display the modal basis using the `displayMap` function

```
from OOPAO.tools.displayTools import displayMap
dm_fried.coefs = M2C_zernike[:,:9]
ngs*tel*dm_fried
displayMap(tel.OPD)
```



Left: Zernike Modal Basis, Right: KL modal basis.

KL Modal Basis

A simplified function is available to provide KL modes orthogonal in the DM space, forcing the Tip/Tilt to be included in the Modal basis.

A more detailed fonction `compute_M2C` is available in the `compute_KL_modal_basis` if required, that provides different options to compute the KL modes (forcing modes in the basis, sampling of the FFT, minimization of the forces, etc).

```
from OOPAO.calibration.compute_KL_modal_basis import compute_KL_basis
M2C_KL = compute_KL_basis(tel=tel,atm =atm,dm=dm_fried)
```

```
TIME ELAPSED: 2 sec. COMPLETED: 100 %
NMAX =  300
RMS opd error = [[9.83402175e-09 1.55273321e-08 1.55273321e-08]]
RMS Positions = [[7.07072183e-08 3.24541557e-07 3.24541557e-07]]
MAX Positions = [[4.52512686e-07 8.08129457e-07 8.08129457e-07]]
WARNING: Number of modes requested too high, taking the maximum value possible!
KL WITH DOUBLE DIAGONALISATION: COVARIANCE ERROR =  4.866922429431739e-14
```

```
dm_fried.coefs = M2C_KL[:,:9]
ngs*tel*dm_fried
displayMap(tel.OPD)
```

8. CALIBRATION

The measurement of the interaction matrix is done using the `InteractionMatrix` class

```
tel.display_optical_path = False
from OOPAO.calibration.InteractionMatrix import InteractionMatrix
# modal interaction matrix
calib_modal = InteractionMatrix( ngs      = ngs,
                                 atm            = atm,
                                 tel            = tel,
                                 dm             = dm_fried,
                                 wfs            = pwfs,
                                 M2C            = M2C_KL[:,:300],
                                 stroke         = 1e-9,    # stroke in [m]
                                 nMeasurements  = 1,# to parallellize
                                 noise          = 'off')            #
```

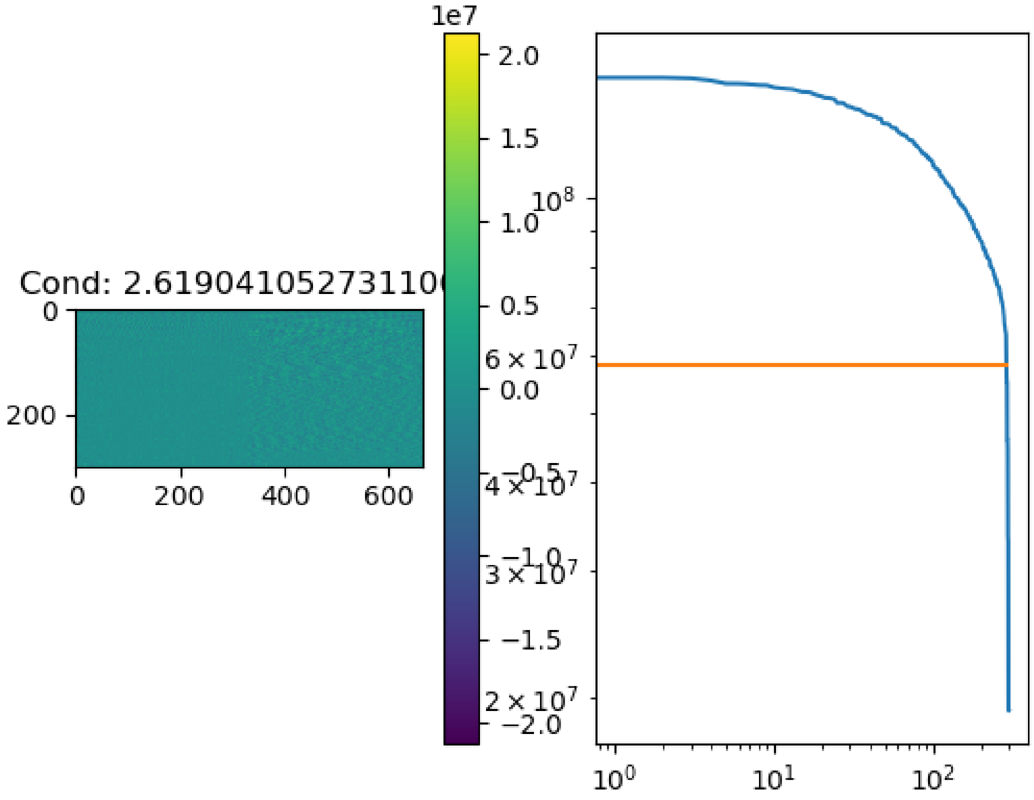
The details of the pseudo inverse of the interaction matrix are accessible within the properties of `calib_modal`.

The input matrix is denoted `calib_modal.D` and its pseudo inverse `calib_modal.M`.

By default, no singular values are truncated in the pseudo inverse: $M = (D^T.D)^{-1}D^T$

It is possible to recompute `calib_modal.M` truncating singular values using the `calib_modal.nTrunc` property:

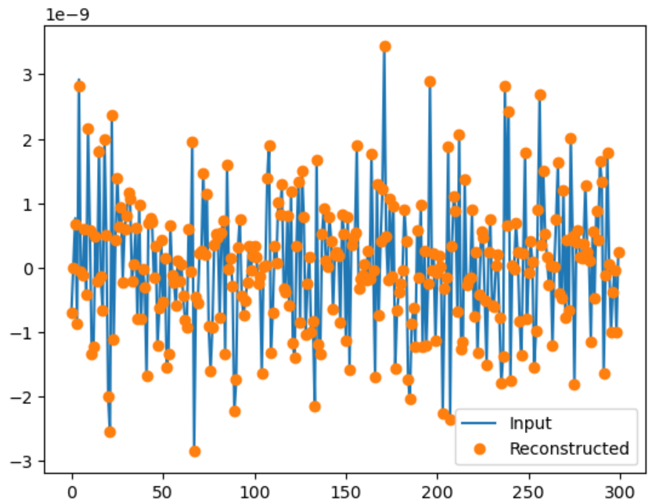
```
# truncate the 10 last singular values
calib_modal.nTrunc = 10
```



Left: Interaction Matrix. Right: Singular Values

It is now possible to reconstruct a given phase using wfs measurements using the `calib_modal.M` matrix:

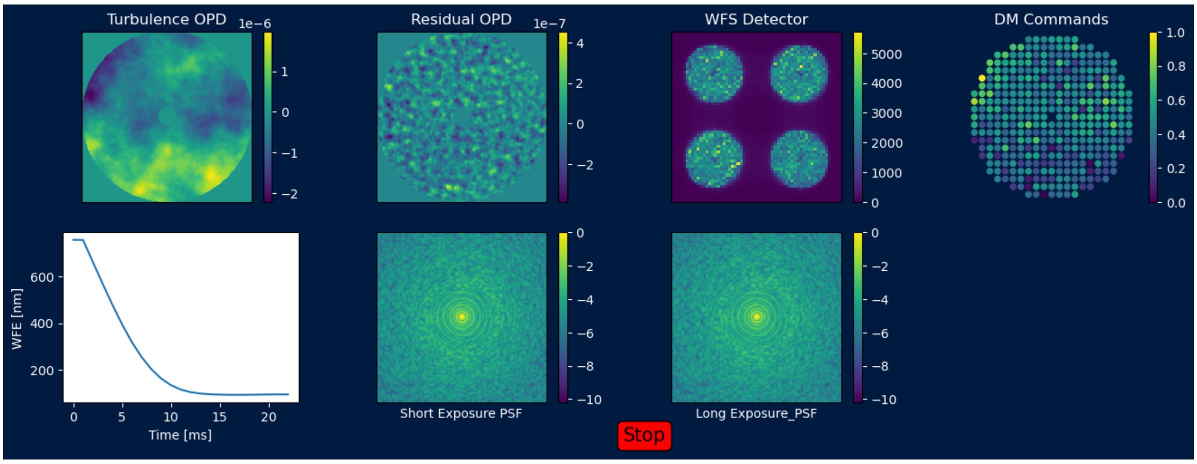
```
input_modes = np.random.randn(300)*1e-9
dm_fried.coefs = M2C_KL[:,:300]@ input_modes
ngs*tel*dm_fried*pwfs

plt.plot(input_modes,label='Input')
plt.plot(calib_modal.M@pwfs.signal,'o',label='Reconstructed')
plt.xlabel(''),plt.legend();
```



9. CLOSED LOOP TUTORIAL CODE

The following figure provides a tutorial code to do a closed-loop simulation. For the sake of clarity, the lines corresponding to the update of the display have been removed and are available in the tutorials codes.

```
src = Source('K',10)# Define a scientific source:
tel.resetOPD()  # initialize Telescope
dm_fried.coefs=0      # initialize DM commands
tel+atm # combine tel and atm to enable turbulence
# These are the calibration data used to close the loop
calib_CL    = calib_modal
M2C_CL      = M2C_KL[:,:300]
ngs*tel*dm_fried*pwfs
# allocate memory to save data
nLoop = 200
SR                          = np.zeros(nLoop)
total                       = np.zeros(nLoop)
residual                    = np.zeros(nLoop)
wfsSignal                   = np.arange(0,pwfs.nSignal)*0
SE_PSF = []
LE_PSF = np.log10(tel.PSF_norma_zoom)
# loop parameters
gainCL                      = 0.4
pwfs.cam.photonNoise         = True
display                     = True
reconstructor = M2C_CL@calib_CL.M
for i in range(nLoop):
    # update phase screens => overwrite tel.OPD and consequently tel.src.phase
    atm.update()
    # save phase variance
    total[i]=np.std(tel.OPD[np.where(tel.pupil>0)])*1e9
    # save turbulent phase
    turbPhase = tel.src.phase
    # propagate to the WFS with the CL commands applied
    ngs*tel*dm_fried*pwfs
    # propagate to the source with the CL commands applied
    src*tel
    tel.print_optical_path()
    dm.coefs=dm.coefs-gainCL*np.matmul(reconstructor,wfsSignal)
    # store the slopes after computing the commands => 2 frames delay
    wfsSignal=pwfs.signal
    SE_PSF.append(np.log10(tel.PSF_norma_zoom))
    LE_PSF = np.mean(SE_PSF, axis=0)
    SR=np.exp(-np.var(tel.src.phase[np.where(tel.pupil==1)]))
    residual=np.std(tel.OPD[np.where(tel.pupil>0)])*1e9
```



Snapshot of the OOPAO display function `cl_plot` allowing to mimic a GUI.
More details are provide in the tutorials

ACKNOWLEDGMENTS

This work was developed during C.T. Heritier's Engineering and Technology Research Fellowship at the European Southern Observatory. The author would like to warmly thank C. Vérinaud, B. Engler, M. Le Louarn, A. Kuznetsov and C. Correia for the numerous discussions and help with AO modelling and the use of Python. As well, the authors are thankful to the contributors of the current version that helped improving the code: J. Aveiro, J.-F. Sauvage and A. Striffling.

References

[1] GUIDO Agapito, A Puglisi, and Simone Esposito. "PASSATA: object oriented numerical simulation software for adaptive optics". In: *Adaptive Optics Systems V*. Vol. 9909. SPIE. 2016, pp. 2164–2172.

[2] François Assémat, Richard W Wilson, and Eric Gendron. "Method for simulating infinitely long and non stationary phase screens with optimized memory storage". In: *Optics express* 14.3 (2006), pp. 988–999.

[3] R Conan, C Correia, et al. "Object-oriented Matlab adaptive optics toolbox". In: SPIE. 2014.

[4] D Gratadour et al. "COMPASS: an efficient, scalable and versatile numerical platform for the development of ELT AO systems". In: *Adaptive Optics Systems IV*. Vol. 9148. SPIE. 2014, pp. 2173–2180.

[5] Miska Le Louarn et al. "Adaptive optics simulations for the European extremely large telescope". In: *Advances in Adaptive Optics II*. Vol. 6272. SPIE. 2006, pp. 1016–1024.

[6] Emiel H Por et al. "High Contrast Imaging for Python (HCIPy): an open-source adaptive optics and coronagraph simulator". In: *Adaptive Optics Systems VI*. Vol. 10703. SPIE. 2018, pp. 1112–1125.

[7] Andrew Reeves. "Soapy: an adaptive optics simulation written purely in Python for rapid concept development". In: *Adaptive optics systems V*. Vol. 9909. SPIE. 2016, pp. 2173–2183.

[8] François Rigaut1a and Marcos Van Dam. "Simulating astronomical adaptive optics systems using yao". In: (2013).