PAOS

Release 0.0.2

Andrea Bocchieri, Enzo Pascale

Dec 14, 2021

CONTENTS

Welcome to the *PAOS* documentation, last updated Dec 14, 2021.

CHAPTER

ONE

TABLE OF CONTENTS

1.1 Introduction

1.1.1 What is PAOS?

PAOS, the Physical Ariel Optics Simulator, is an End-to-End physical optics propagation model of the Ariel Telescope and subsystems. *PAOS* was developed to demonstrate that even at wavelengths where it is not diffraction-limited Ariel still delivers high quality data for scientific analysis. *PAOS* simulates the complex wavefront along the propagation axis, and the Point Spread Function (PSF) at the focal planes. To do so, *PAOS* implements the Paraxial theory described in Lawrence et al., Applied Optics and Optical Engineering, Volume XI (1992). *PAOS* has been validated using the physical optics propagation library PROPER (see John E. Krist, SPIE (2007)). In short, *PAOS* can study the effect of diffraction and aberrations impacting the optical performance and related systematics. This allows performing a large number of detailed analyses, both on the instrument side and on the optimization of the Ariel mission. Having a generic input system which mimics the Zemax OpticStudio interface, *PAOS* allows the user expert in CAD modeling to simulate other optical systems, as well.

Note: *PAOS* v 0.0.2 works on Python 3+

Warning: *PAOS* is still under development. If you have any issue or find any bug, please report it to the developers.

1.1.2 Citing

If you use this software or its products, please cite (Bocchieri A. - *PAOS* - *in prep*).

1.1.3 Changelog

Tip: Please note that *PAOS* does not implement an automatic updating system. Be always sure that you are using the most updated version by monitoring GitHub.

Version	Date	Changes
0.0.2	15/09/2021	setting up new *PAOS* repository
0.0.2	20/10/2021	first documented *PAOS* release

1.2 Installation

The following notes guide you toward the installation of PAOS.

1.2.1 Install from git

You can clone *PAOS* from our main git repository.

```
$ git clone https://github.com/arielmission-space/PAOS
```

Then, move into the *PAOS* folder.

```
$ cd /your_path/PAOS
```

1.2.2 Prepare the run

If you want to use *PAOS* in a python shell or jupyter notebook, you may need to add to PYTHONPATH the path to local *PAOS* path.

This can be done as in the below code example.

```
import os, sys
paospath = "~/git/PAOS"
if not os.path.expanduser(paospath) in sys.path:
    sys.path.append( os.path.expanduser(paospath) )

import paos
```

1.3 User guide

This user guide will walk you through the main *PAOS* functionalities.

For clarity, it is divided into several sections:

1. ***Quick start*** How to launch *PAOS* from terminal shell
2. ***Input system*** The input system used by *PAOS*
3. ***ABCD description*** ABCD matrix theory and how it is implemented in *PAOS*

4. ***POP description*** Physical Optics Propagation and how it is implemented in *PAOS*
5. ***Aberration description*** Wavefront aberrations and how they are implemented in *PAOS*
6. ***Materials description*** Optical materials and how they are implemented in *PAOS*
7. ***Plotting results*** How to plot POP results
8. ***Saving results*** How to save POP results
9. ***Automatic pipeline*** An automated way to run *PAOS* from python console or jupyter notebook

Each functionality has a self-consistent example to run in python console or jupyter notebook.

Have a good read. Feedback is highly appreciated.

1.3.1 Table of Contents

Quick start

Short explanation on how to quickly run *PAOS* and have its output stored in a convenient file.

Running PAOS from terminal

The quickest way to run *PAOS* is from terminal.

Run it with the *help* flag to read the available options:

```
$ python paos.py --help
```

The main command line flags are listed below.

flag	description
-c, –configuration	Input configuration file to pass
-o, –output	Output file
-p, –plot	Save output plots
-n, –nThreads	Number of threads for parallel processing
-d, –debug	Debug mode screen
-l, –log	Store the log output on file

Where the configuration file shall be an *.xlsx* file and the output file an *.h5* file (see later in *The output file*). *-n* must be followed by an integer. To activate *-p*, *-d* and *-l* no argument is needed.

Other option flags may be given to run specific simulations.

flag	description
-wl, –wavelength	A list of specific wavelengths at which to run the simulation
-wlg, –wavelength_grid	A list with min wl, max wl, spectral resolution to build a wavelength grid
-wfe, –wfe_simulation	A list with wfe realization file and column to simulate an aberration

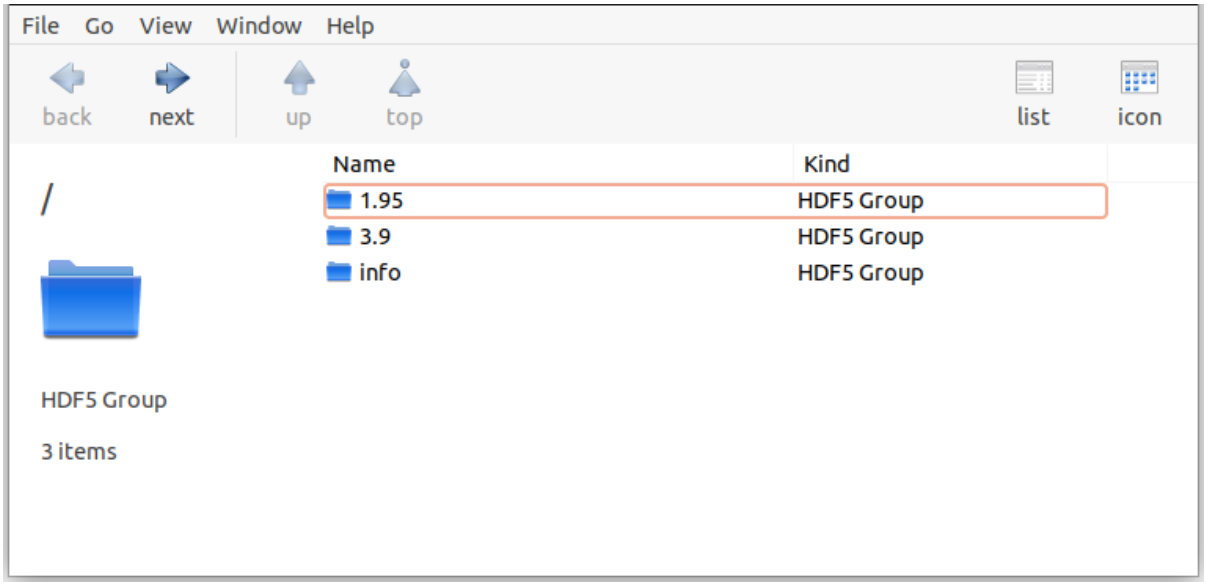
To have a lighter output please use the option flags listed below.

flag	description
-keys, –keys_to_keep	A list with the output dictionary keys to save
-lo, –light_output	Save only at last optical surface

To activate *-lo* no argument is needed.

The output file

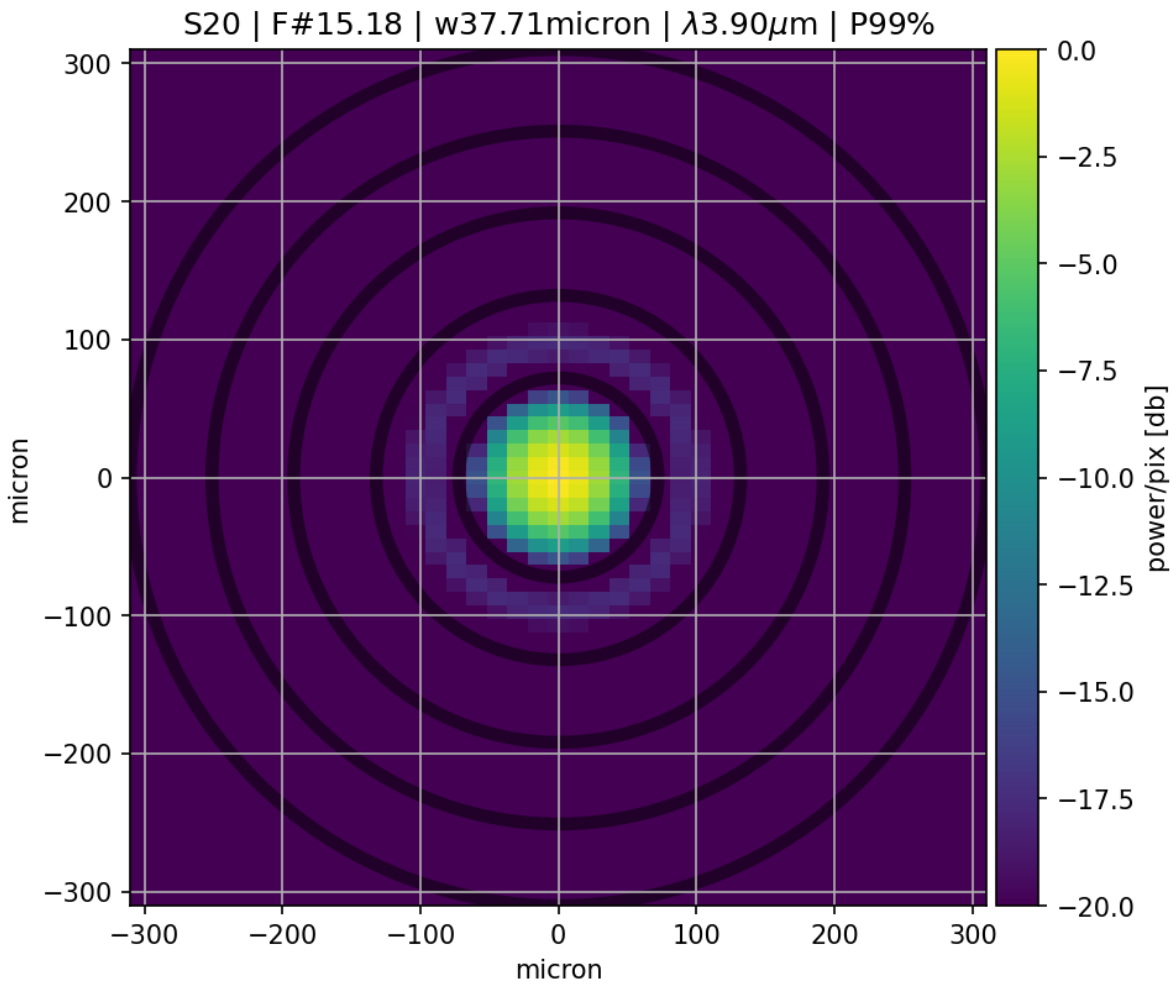
PAOS stores its main output product to a HDF5 *.h5* file. To open it, please choose your favourite viewer (e.g. HDFView, HDFCompass) or API (e.g. Cpp, FORTRAN and Python).



For more information on how to produce a similar output file, see *Saving results*.

The default plot

An important part of understanding the *PAOS* output is often to look at the default plot, as in the below figure.



This plot shows the PSF, i.e. the squared amplitude of the complex wavefront, at the *AIRS-CH0* focal plane.

The color scale can be either linear or logarithmic. The x and y axes are in physical units, e.g. micron. For reference, dark circular rings are superimposed on the first five zeros of the circular Airy function.

The title of the plot features the optical surface name, the focal number, the Gaussian beam width, the simulation wavelength and the total optical throughput that reaches the surface.

For more information on how to produce a similar plot, see *Plotting results*.

Input system

PAOS has a generic input system that tries to mimic the Zemax OpticStudio interface to allow the user expert in Computer Aided Design (CAD) modeling to simulate also optical systems other than *Ariel*.

For instance, *PAOS* is currently used to simulate the optical performance of the stratospheric balloon-borne experiment EXCITE (Tucker et al., The Exoplanet Climate Infrared TElescope (EXCITE) (2018)).

Tip: The interested reader may refer to the section *Plotting results* to see an example of *PAOS* results for EXCITE.

Having a generic input system was also advantageous to validate the *PAOS* code, allowing to use the Hubble optical system (see *Validation*).

Configuration file

The configuration file is an Excel spreadsheet containing three data sheets named *General*, *Fields* and *Lens Data*.

General (see Fig. 1.1) has the simulation wavelength in micron, grid size in pixel and zoom, defined as the ratio of grid size to initial beam size in unit of pixel.

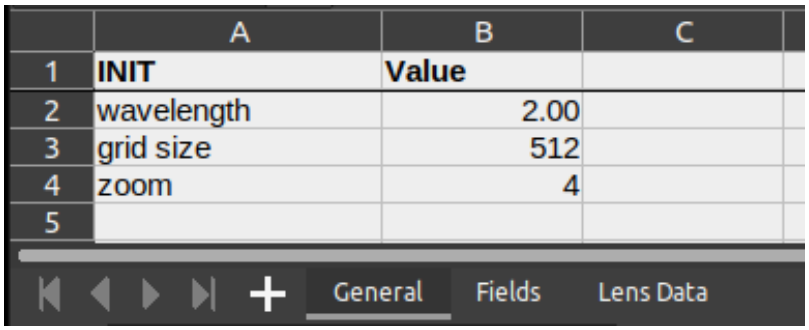


Fig. 1.1: *General*

Fields (see Fig. 1.2) has the input field angles in degrees.

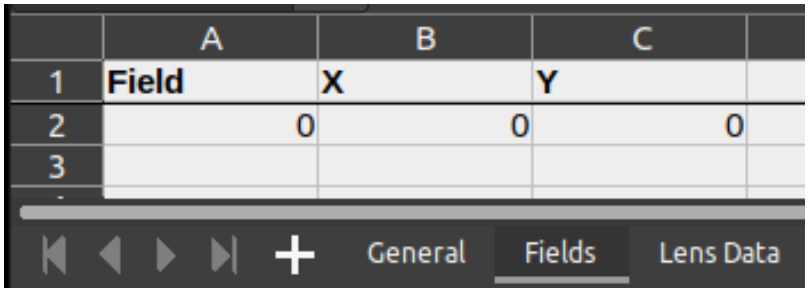


Fig. 1.2: *Fields*

Lens Data (see Fig. 1.3 and Fig. 1.4) is the lens data and contains the sequence of surfaces for the simulation. Supported surface types include Coordinate Break, Standard, Obscuration, Paraxial Lens, Prism, Slit and Zernike.

Setting the *Ignore* flag to 1 skips the current surface, while setting the *Stop* flag to 1 makes the current surface a Stop (see *Stops*). To save the current surface, set the *Save* flag to 1.

Depending on the surface, optical materials are supported (see *Materials description*).

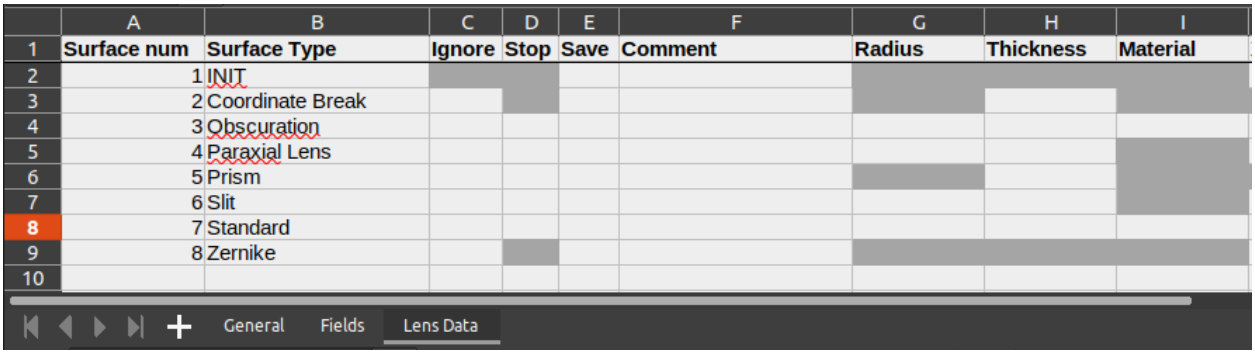


Fig. 1.3: *Lens Data (1)*

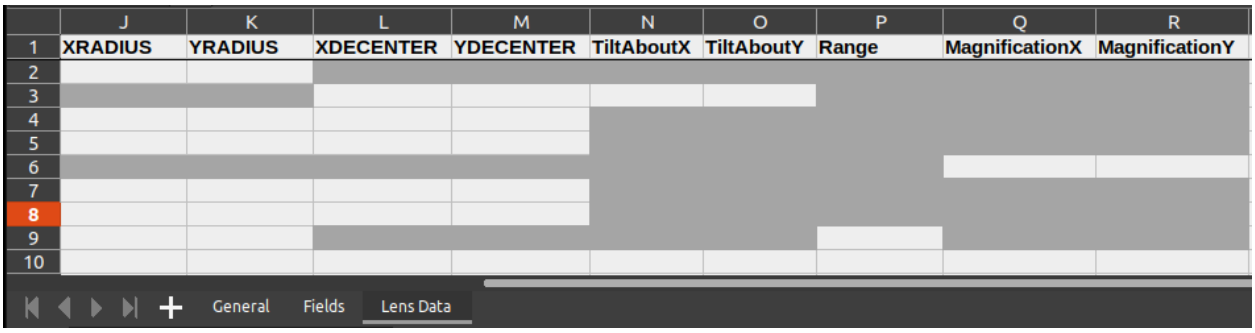


Fig. 1.4: *Lens Data (2)*

Read configuration file

Given the input file name, *PAOS* implements *ReadConfig*, a function that opens the file and returns the simulation parameters, as shown in the example below.

```
from paos.paos_parseconfig import ReadConfig
simulation_parameters = ReadConfig('path/to/conf/file')
```

Parse configuration file

Then, the simulation parameters are parsed by *ParseConfig* that prepares the simulation run. This function then returns the input pupil diameter along with the general parameters, the input fields and the optical chain, as shown below.

```
from paos.paos_parseconfig import ParseConfig
pupil_diameter, general, fields, optical_chain = ParseConfig('path/to/conf/file')
```

ABCD description

PAOS implements the paraxial theory described in Lawrence et al., Applied Optics and Optical Engineering, Volume XI (1992). In *PAOS*, this is handled by the *ABCD* class.

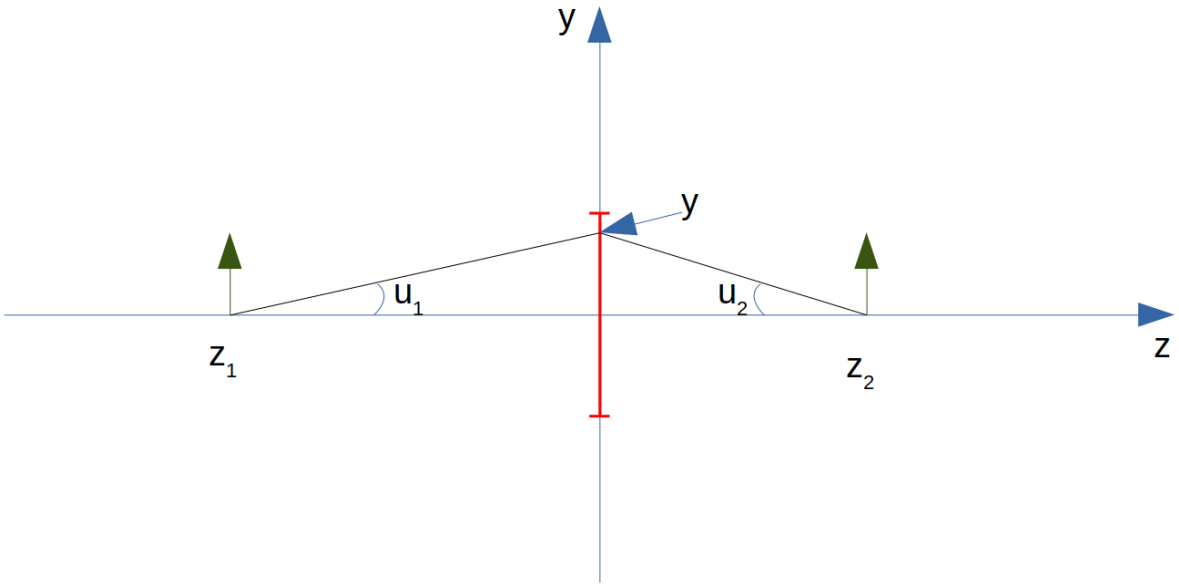
The paraxial region

For self-consistency, we give a definition for paraxial region, following Smith, Modern Optical Engineering, Third Edition (2000).

The paraxial region of an optical system is a thin threadlike region about the optical axis where all the slope angles and the angles of incidence and refraction may be set equal to their sines and tangents.

Optical coordinates

The *PAOS* code implementation assumes optical coordinates as defined in the diagram below.



where

1. z_1 is the coordinate of the object (< 0 in the diagram)
2. z_2 is the coordinate of the image (> 0 in the diagram)
3. u_1 is the slope, i.e. the tangent of the angle = angle in paraxial approximation; $u_1 > 0$ in the diagram.
4. u_2 is the slope, i.e. the tangent of the angle = angle in paraxial approximation; $u_2 < 0$ in the diagram.
5. y is the coordinate where the rays intersect the thin lens (coloured in red in the diagram).

The (thin) lens equation is

$$-\frac{1}{z_1} + \frac{1}{z_2} = \frac{1}{f} \tag{1.1}$$

where f is the lens focal length: $f > 0$ causes the beam to be more convergent, while $f < 0$ causes the beam to be more divergent.

The tangential plane is the YZ plane and the sagittal plane is the XZ plane.

Ray tracing

Paraxial ray tracing in the tangential plane (YZ) can be done by defining the vector $\vec{v_t} = (y, u_y)$ which describes a ray propagating in the tangential plane. Paraxial ray tracing can be done using ABCD matrices (see later in *Optical system equivalent*).

Note: In the sagittal plane, the same equation apply, modified when necessary when cylindrical symmetry is violated. The relevant vector is $\vec{v_s} = (x, u_x)$.

PAOS implements the function *raytrace* to perform a diagnostic ray tracing of an optical system, given the input fields and the optical chain. This function then prints the ray positions and slopes in the tangential and sagittal planes for each surface of the optical chain.

Below is an example to call *raytrace*, provided you already have the optical chain (if not, back to *Parse configuration file*).

```
from paos.paos_raytrace import raytrace
raytrace(field={'us': 0.0, 'ut': 0.0}, opt_chain=optical_chain)
```

Propagation

Either in free space or in a refractive medium, propagation over a distance t (positive left $\rightarrow$ right) is given by

$$\begin{pmatrix} y_2 \\ u_2 \end{pmatrix} = \begin{pmatrix} 1 & t \\ 0 & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ u_1 \end{pmatrix} = \hat{T} \begin{pmatrix} y_1 \\ u_1 \end{pmatrix} \tag{1.2}$$

Below is an example to use *ABCD* to propagate a light ray over a thickness $t = 50.0$ mm.

```
from paos.paos_abcd import ABCD
thickness = 50.0  # mm
abcd = ABCD(thickness=thickness)
(A, B), (C, D) = abcd.ABCD
```

Thin lenses

A thin lens changes the slope angle and this is given by

$$\begin{pmatrix} y_2 \\ u_2 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ -\Phi & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ u_1 \end{pmatrix} = \hat{L} \begin{pmatrix} y_1 \\ u_1 \end{pmatrix} \tag{1.3}$$

where $\Phi = \frac{1}{f}$ is the lens optical power.

Below is an example to use *ABCD* to simulate the effect of a thin lens with radius of curvature $R = 20.0$ mm on a light ray.

```
from paos.paos_abcd import ABCD
radius = 20.0  # mm
abcd = ABCD(curvature=1.0/radius)
(A, B), (C, D) = abcd.ABCD
```

Dioptre

When light propagating from a medium with refractive index n_1 enters in a dioptre of refractive index n_2, the slope varies as

$$\begin{pmatrix} y_2 \\ u_2 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ -\frac{\Phi}{n_2} & \frac{n_1}{n_2} \end{pmatrix} \begin{pmatrix} y_1 \\ u_1 \end{pmatrix} = \hat{D} \begin{pmatrix} y_1 \\ u_1 \end{pmatrix} \tag{1.4}$$

with the dioptre power $\Phi = \frac{n_2 - n_1}{R}$, where R is the dioptre radius of curvature.

Note: $R > 0$ if the centre of curvature is at the right of the dioptre and $R < 0$ if at the left.

Below is an example to use *ABCD* to simulate the effect of a dioptre with radius of curvature $R = 20.0$ mm that causes a change of medium from $n_1 = 1.0$ to $n_2 = 1.5$ on a light ray.

```
from paos.paos_abcd import ABCD
n1, n2 = 1.0, 1.5
radius = 20.0  # mm
abcd = ABCD(curvature = 1.0/radius, n1 = n1, n2 = n2)
(A, B), (C, D) = abcd.ABCD
```

Medium change

The limiting case of a dioptre with $R \to \infty$ represents a change of medium.

$$\begin{pmatrix} y_2 \\ u_2 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & \frac{n_1}{n_2} \end{pmatrix} \begin{pmatrix} y_1 \\ u_1 \end{pmatrix} = \hat{N} \begin{pmatrix} y_1 \\ u_1 \end{pmatrix} \tag{1.5}$$

Below is an example to use *ABCD* to simulate the effect of a change of medium from $n_1 = 1.0$ to $n_2 = 1.5$ on a light ray.

```
from paos.paos_abcd import ABCD
n1, n2 = 1.0, 1.5
abcd = ABCD(n1 = n1, n2 = n2)
(A, B), (C, D) = abcd.ABCD
```

Thick lenses

A real (thick) lens is modelled as

$$\begin{pmatrix} y_2 \\ u_2 \end{pmatrix} = \hat{D}_b \hat{T} \hat{D}_a \begin{pmatrix} y_1 \\ u_1 \end{pmatrix} \tag{1.6}$$

i.e. propagation through the dioptre D_a (first encountered by the ray), then a propagation in the medium, followed by the exit dioptre D_b.

Note: When the thickness of the dioptre, t, is negligible and can be set to zero, this gives back the thin lens ABCD matrix.

Note: If a dioptre has $R \to \infty$, this gives a plano-concave or plano-convex lens, depending on the curvature of the other dioptre.

Below is an example to use *ABCD* to simulate the effect of a thick lens on a light ray. The lens is $t_c = 5.0$ mm thick and is plano-convex, i.e. the first dioptre has $R = \infty$ and the second has $R = -20.0$ mm, causing the beam to converge. The index of refraction in object space and in image space is that of free space $n_{os} = n_{is} = 1.0$, while the lens medium has $n_l = 1.5$.

```
import numpy as np
from paos.paos_abcd import ABCD

radius1, radius2 = np.inf, -20.0  # mm
```

(continues on next page)

(continued from previous page)

```
n_os, n_l, n_is = 1.0, 1.5, 1.0
center_thickness = 5.0
abcd = ABCD(curvature = 1.0/radius1, n1 = n_os, n2 = n_l)
abcd = ABCD(thickness = center_thickness) * abcd
abcd = ABCD(curvature = 1.0/radius2, n1 = n_l, n2 = n_is) * abcd
(A, B), (C, D) = abcd.ABCD
```

Magnification

A magnification is modelled as

$$\begin{pmatrix} y_2 \\ u_2 \end{pmatrix} = \begin{pmatrix} M & 0 \\ 0 & 1/M \end{pmatrix} = \hat{M} \begin{pmatrix} y_1 \\ u_1 \end{pmatrix} \tag{1.7}$$

Below is an example to use *ABCD* to simulate the effect of a magnification $M = 2.0$ on a light ray.

```
from paos.paos_abcd import ABCD
from paos.paos_abcd import ABCD
abcd = ABCD(M=2.0)
(A, B), (C, D) = abcd.ABCD
```

Prism

The prism changes both the slope and the magnification. Following J. Taché, "Ray matrices for tilted interfaces in laser resonators," Appl. Opt. 26, 427-429 (1987) we report the ABCD matrices for the tangential and sagittal transfer:

$$P_t = \begin{pmatrix} \frac{cos(\theta_4)}{cos(\theta_3)} & 0 \\ 0 & \frac{ncos(\theta_3)}{cos(\theta_4)} \end{pmatrix} \begin{pmatrix} 1 & L \\ 0 & 1 \end{pmatrix} \begin{pmatrix} \frac{cos(\theta_2)}{cos(\theta_1)} & 0 \\ 0 & \frac{cos(\theta_1)}{ncos(\theta_2)} \end{pmatrix} \tag{1.8}$$

$$P_s = \begin{pmatrix} 1 & \frac{L}{n} \\ 0 & 1 \end{pmatrix} \tag{1.9}$$

where n is the refractive index of the prism, L is the geometrical path length of the prism, and the angles θ_i are as described in Fig.2 from the article, reported in the image below.

After some algebra, the ABCD matrix for the tangential transfer can be rewritten as:

$$P_t = \begin{pmatrix} A & B \\ C & D \end{pmatrix} \tag{1.10}$$

where

$$\begin{aligned} A &= \frac{cos(\theta_2)cos(\theta_4)}{cos(\theta_1)cos(\theta_3)} \\ B &= \frac{L}{n}\frac{cos(\theta_1)cos(\theta_4)}{cos(\theta_2)cos(\theta_3)} \\ C &= 0.0 \\ D &= 1.0/A \end{aligned} \tag{1.11}$$

Below is an example to use *ABCD* to simulate the effect of a prism on a collimated light ray. The prism is $t = 8.0$ mm thick and has a refractive index of $n_p = 1.5$. The prism angles θ_i are selected in conformity with the ray propagation in Fig. 1.5.

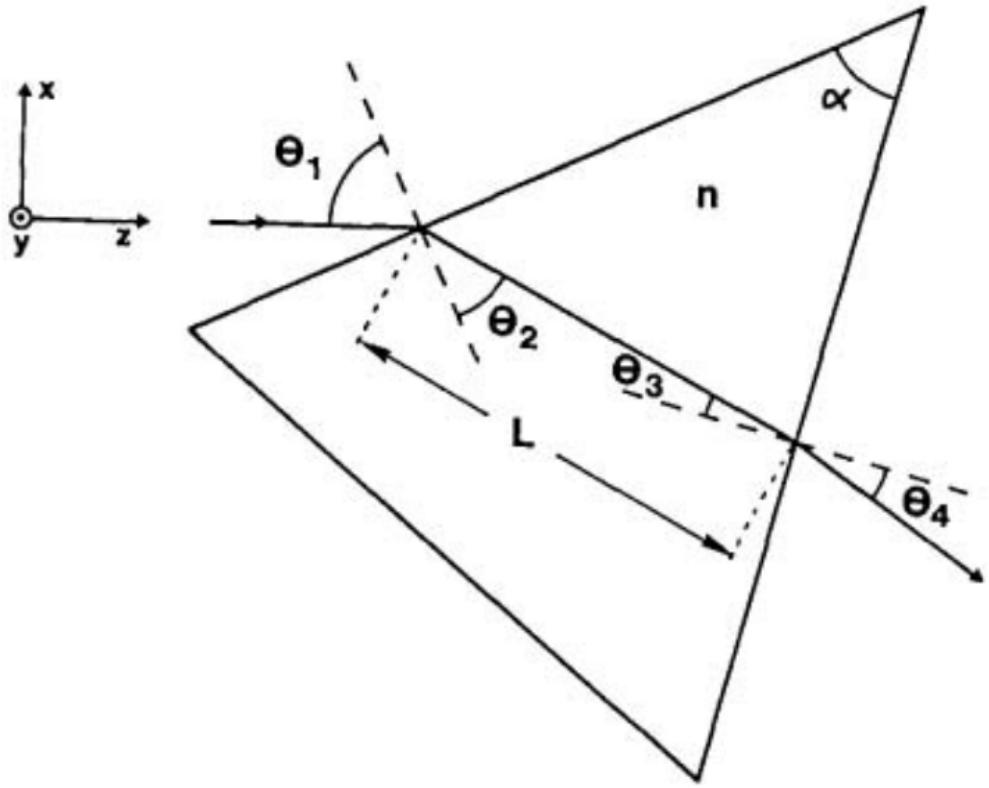


Fig. 1.5: *Ray propagation through a prism (from the article cited in the text)*

```
import numpy as np
from paos.paos_abcd import ABCD

thickness = 8.0e-3  # m
n = 1.5

theta_1 = np.deg2rad(60.0)
theta_2 = np.deg2rad(-30.0)
theta_3 = np.deg2rad(20.0)
theta_4 = np.deg2rad(-30.0)

A = np.cos(theta_2)*np.cos(theta_4)/(np.cos(theta_1)*np.cos(theta_3))
B = np.cos(theta_1)*np.cos(theta_4)/(np.cos(theta_2)*np.cos(theta_3))/n
C = 0.0
D = 1.0/A

abcdt = ABCD()
abcdt.ABCD = np.array([[A,B], [C,D]])
abcds = ABCD()
abcds.ABCD= np.array([[1, thickness/n], [0, 1]])
```

Optical system equivalent

The ABCD matrix method is a convenient way of treating an arbitrary optical system in the paraxial approximation. This method is used to describe the paraxial behavior, as well as the Gaussian beam properties and the general diffraction behaviour.

Any optical system can be considered a black box described by an effective ABCD matrix. This black box and its matrix can be decomposed into four, non-commuting elementary operations (primitives):

1. magnification change
2. change of refractive index
3. thin lens
4. translation of distance (thickness)

Explicitly:

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} = \begin{pmatrix} 1 & t \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ -\Phi & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & n_1/n_2 \end{pmatrix} \begin{pmatrix} M & 0 \\ 0 & 1/M \end{pmatrix} = \hat{T}\hat{L}\hat{N}\hat{M} \tag{1.12}$$

where the four free parameters t, Φ, n_1/n_2, M are, respectively, the effective thickness, power, refractive index ratio, and magnification. Not to be confused with thickness, power, refractive index ratio, and magnification of the optical system under study and its components.

All diffraction propagation effects occur in the single propagation step of distance t. Only this step requires any substantial computation time.

The parameters are estimated as follows:

$$\begin{aligned} M &= \frac{AD - BC}{D} \\ n_1/n_2 &= MD \\ t &= \frac{B}{D} \\ \Phi &= -\frac{C}{M} \end{aligned} \tag{1.13}$$

With these definitions, the effective focal length is

$$f_{eff} = \frac{1}{\Phi M} \tag{1.14}$$

Below is an example to use *ABCD* to simulate an optical system equivalent for a magnification $M = 2.0$, a change of medium from $n_1 = 1.0$ to $n_2 = 1.5$, a thin lens with radius of curvature $R = 20.0$ mm, and a propagation over a thickness $t = 5.0$ mm.

```
from paos.paos_abcd import ABCD

radius = 20.0  # mm
n1, n2 = 1.0, 1.5
thickness = 5.0  # mm
magnification = 2.0

abcd = ABCD(thickness = thickness, curvature = 1.0/radius, n1=n1, n2=n2, M=magnification)
(A, B), (C, D) = abcd.ABCD
```

POP description

Brief description of some concepts of physical optics wavefront propagation (POP) and how they are implemented in *PAOS*.

General diffraction

Diffraction is the deviation of a wave from the propagation that would be followed by a straight ray, which occurs when part of the wave is obstructed by the presence of a boundary. Light undergoes diffraction because of its wave nature. The Huygens-Fresnel principle is often used to explain diffraction intuitively. Each point on the wavefront propagating from a single source can be though of as being the source of spherical secondary wavefronts (wavelets). The combination of all wavelets cancels except at the boundary, which is locally parallel to the initial wavefront. However, if there is an object or aperture which obstructs some of the wavelets, changing their phase or amplitude, these wavelets interfere with the unobstructed wavelets, resulting in the diffraction of the wave.

Fresnel diffraction theory

Fresnel diffraction theory requires the following conditions to be met (see e.g. Lawrence et al., Applied Optics and Optical Engineering, Volume XI (1992)):

1. aperture sized significantly larger than the wavelength
2. modest numerical apertures
3. thin optical elements

PAOS is implemented assuming that Fresnel diffraction theory holds.

Coordinate breaks

Coordinate breaks are implemented as follows:

1. Decenter by x_{dec}, y_{dec}
2. Rotation XYZ (first X, then Y, then Z)

The rotation is intrinsic (X, then around new Y, then around new Z).

To transform the sagittal coordinates (x, u_x) and the tangential coordinates (y, u_y), define the position vector

$$\vec{R_0} = (x - x_{dec}, y - y_{dec}, 0) \tag{1.15}$$

and the unit vector of the light ray

$$\vec{n_0} = (zu_x, zu_y, z) \tag{1.16}$$

where z is an appropriate projection of the unit vector such that u_x and u_y are the tangent of the angles (though we are in the paraxial approximation and this might not be necessary).

Note: z does not need to be calculated because it gets normalised away.

The position on the rotated x', y' plane would be $\vec{R_0}' = (x', y', 0)$ and the relation is

$$U^T \vec{R_0} + \rho U^T \vec{n_0} = \vec{R_0}' \tag{1.17}$$

that can be solved as

$$\begin{aligned} U^T \vec{n_0} = \vec{n_0}' = z'(u'_x, u'_y, 1) \\ \rho = -\frac{z'_0}{z'} \end{aligned} \tag{1.18}$$

Below is an example of a coordinate break where the input field is centered on the origin and has null angles u_s and u_t and is subsequently decentered on the Y axis by $y_{dec} = 10.0$mm and rotated around the X axis by $x_{rot} = 0.1°$.

```
from paos.paos_coordinatebreak import CoordinateBreak

field={'us': 0.0, 'ut': 0.0}
vt = np.array([0.0, field['ut']])
vs = np.array([0.0, field['us']])

xdec, ydec = 0.0, 10.0e-3  # m
xrot, yrot, zrot = 0.1, 0.0, 0.0  # deg
vt, vs = CoordinateBreak(vt, vs, xdec, ydec, xrot, yrot, zrot, order=0.0)
```

Gaussian beams

For a Gaussian beam, i.e. a beam with an irradiance profile that follows an ideal Gaussian distribution (see e.g. Smith, Modern Optical Engineering, Third Edition (2000))

$$I(r) = I_0 e^{-\frac{2r^2}{w(z)^2}} = \frac{2P}{\pi w(z)^2} e^{-\frac{2r^2}{w(z)^2}} \tag{1.19}$$

where I_0 is the beam intensity on axis, r is the radial distance and w is the radial distance at which the intensity falls to I_0/e^2, i.e., to 13.5 percent of its value on axis.

Note: $w(z)$ is the semi-diameter of the beam and it encompasses 86.5% of the beam power.

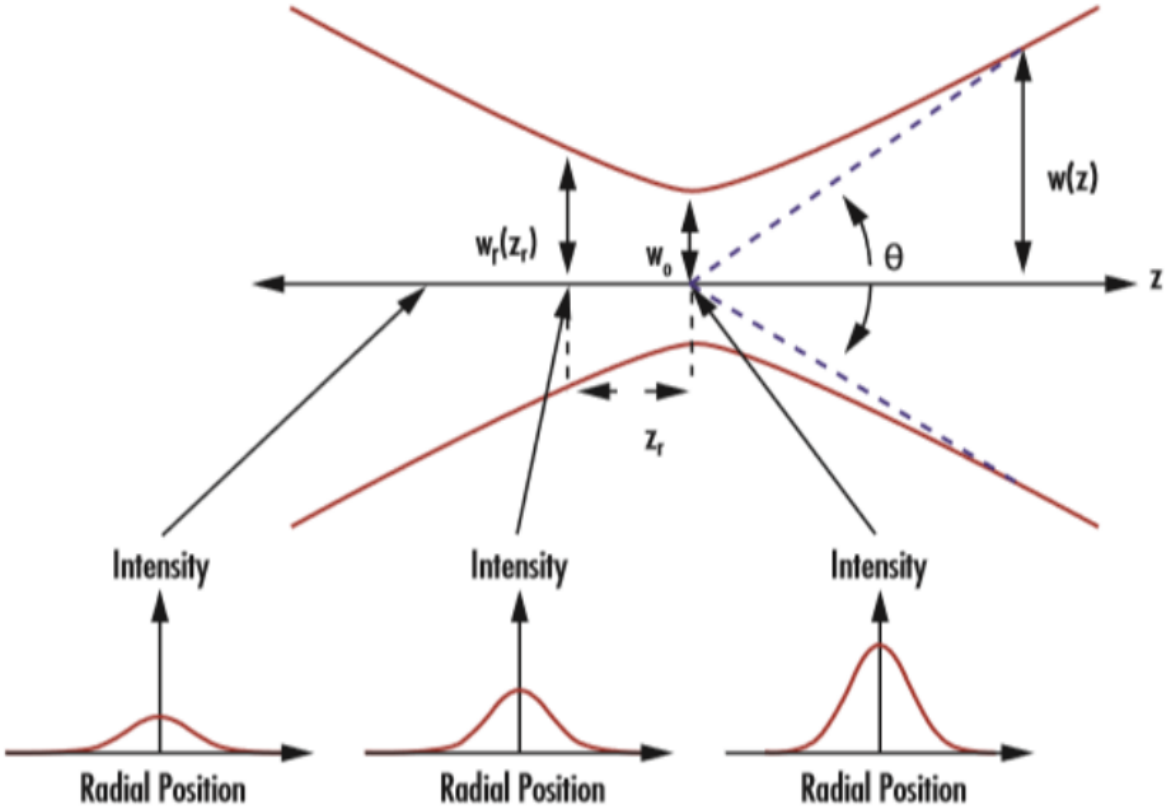
Due to diffraction, a Gaussian beam will converge and diverge from the beam waist w_0, an area where the beam diameter reaches a minimum size, hence the dependence of $w(z)$ on z, the longitudinal distance from the waist w_0 to the plane of $w(z)$, henceforward "distance to focus".

A Gaussian beam spreads out as

$$w(z)^2 = w_0^2 \left[1 + \left(\frac{\lambda z}{\pi w_0^2}\right)^2\right] = w_0^2 \left[1 + \left(\frac{z}{z_R}\right)^2\right] \tag{1.20}$$

where z_R is the *Rayleigh distance*.

A Gaussian beam is defined by just three parameters: w_0, z_R and the divergence angle θ, as in the figure below (from Edmund Optics, Gaussian beam propagation).



The complex amplitude of a Gaussian beam is of the form (see e.g. Lawrence et al., Applied Optics and Optical Engineering, Volume XI (1992))

$$a(r,0) = e^{-\frac{r^2}{w_0^2}} e^{-\frac{jkr^2}{R}} \tag{1.21}$$

where k is the wavenumber and R is the radius of the quadratic phase factor, henceforward "phase radius". This reduces to

$$a(r,0) = e^{-\frac{r^2}{w_0^2}} \tag{1.22}$$

at the waist, where the wavefront is planar ($R \to \infty$).

Rayleigh distance

The Rayleigh distance of a Gaussian beam is defined as the value of z where the cross-sectional area of the beam is doubled. This occurs when w(z) has increased to $\sqrt{2}w_0$.

Explicitly:

$$z_R = \frac{\pi w_0^2}{\lambda} \tag{1.23}$$

The physical significance of the Rayleigh distance is that it indicates the region where the curvature of the wavefront reaches a minimum value. Since

$$R(z) = z + \frac{z_R^2}{z} \tag{1.24}$$

in the Rayleigh range, the phase radius is $R = 2z_R$.

From the point of view of the *PAOS* code implementation, the Rayleigh distance is used to develop a concept of near- and far-field, to define specific propagators (see *Wavefront propagation*).

Gaussian beam propagation

To the accuracy of Fresnel diffraction, a Gaussian beam propagates as (see e.g. Lawrence et al., Applied Optics and Optical Engineering, Volume XI (1992))

$$a(r,z) = e^{-j[kz-\theta(z)]} e^{-\frac{r^2}{w(z)^2}} e^{-\frac{jkr^2}{R(z)}} \tag{1.25}$$

where $\theta(z)$ is a piston term referred to as the phase factor, given by

$$\theta(z) = tan^{-1}\left(\frac{z_R}{z}\right) \tag{1.26}$$

$\theta(z)$ varies from π to $-\pi$ when propagating from $z = -\infty$ to $z = \infty$.

The Gaussian beam propagation can also be described using ABCD matrix optics. A complex radius of curvature $q(z)$ is defined as:

$$\frac{1}{q(z)} = \frac{1}{R(z)} - \frac{j\lambda}{\pi n w(z)^2} \tag{1.27}$$

Propagating a Gaussian beam from some initial position (1) through an optical system (ABCD) to a final position (2) gives the following transformation:

$$\frac{1}{q_2} = \frac{C + D/q_1}{A + B/q_1} \tag{1.28}$$

Gaussian beam magnification

The Gaussian beam magnification can also be described using ABCD matrix optics. Using the definition given in *Magnification*, in this case

$$\begin{aligned} A &= M \\ D &= 1/M \\ B &= C = 0 \end{aligned} \tag{1.29}$$

Therefore, for the complex radius of curvature we have that

$$q_2 = M^2 q_1. \tag{1.30}$$

Using the definition of $q(z)$ it follows that

1. $R_2 = M^2 R_1$
2. $w_2 = M w_1$

for the phase radius and the semi-diameter of the beam, while from the definition of Rayleigh distance it follows that

1. $z_{R,2} = M^2 z_{R,1}$
2. $w_{0,2} = M w_{0,1}$
3. $z_2 = M^2 z_1$

for the Rayleigh distance, the Gaussian beam waist and the distance to focus.

Note: In the current version of *PAOS*, the Gaussian beam width is set along x. So, only the sagittal magnification changes the Gaussian beam properties. A tangential magnification changes only the curvature of the propagating wavefront.

```
Ms, Mt = 1.0, 1.5
wfo.Magnification(Ms, Mt)
```

Wavefront propagation

The methods for propagation are the hardest part of the problem of modelling the propagation through a well-behaved optical system. A thorough discussion of this problem is presented in Lawrence et al., Applied Optics and Optical Engineering, Volume XI (1992). Here we discuss the relevant aspects for the *PAOS* code implementation.

Once an acceptable initial sampling condition is established and the propagation is initiated, the beam starts to spread due to diffraction. Therefore, to control the size of the array so that beam aliasing does not change much from the initial state it is important to choose the right propagator (far-field or near-field).

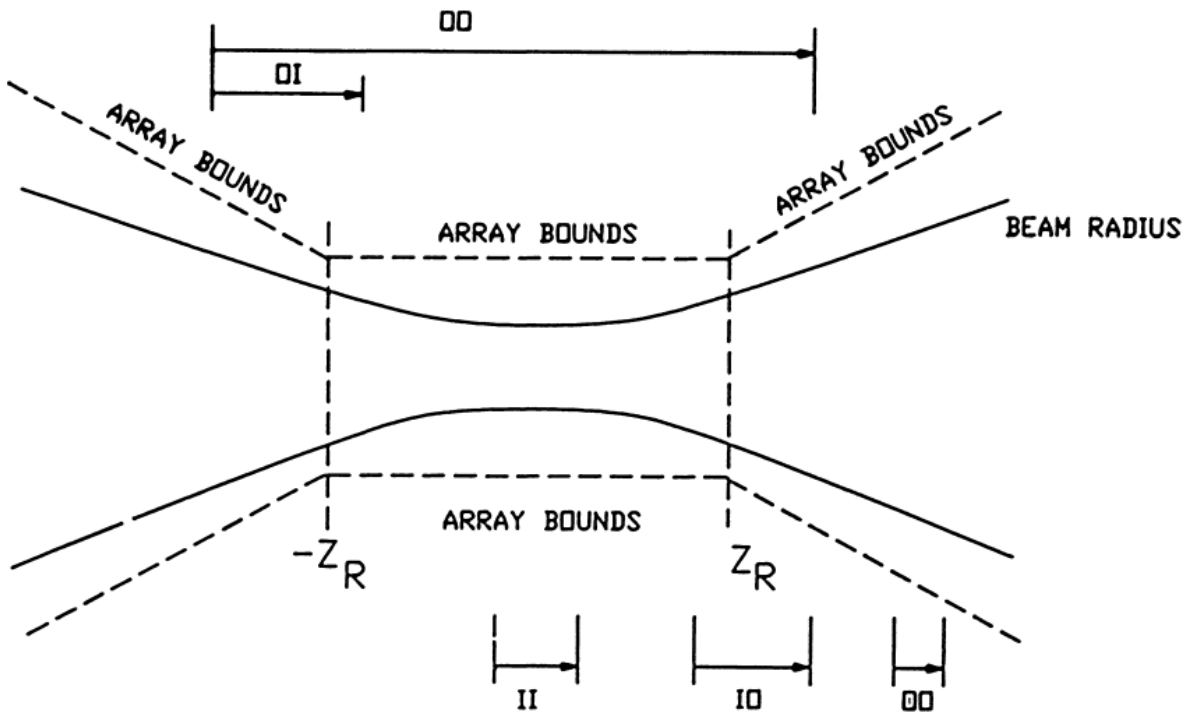
PAOS propagates the pilot Gaussian beam through all optical surfaces to calculate the beam width at all points in space. The Gaussian beam acts as a surrogate of the actual beam and the Gaussian beam parameters inform the POP simulation. In particular the *Rayleigh distance* z_R is used to inform the choice of specific propagators.

Aliasing occurs when the beam size becomes comparable to the array size. Instead of adjusting the sampling period to track exactly, it is more effective to have a region of constant sampling period near the beam waist (constant coordinates system of the form $\Delta x_2 = \Delta x_1$) and a linearly increasing sampling period far from the waist (expanding coordinates system of the form $\Delta x_2 = \lambda |z| / M \Delta x_1$).

For a given point, there are four possibilities in moving from inside or outside to inside or outside the Rayleigh range (RR), defined as the region between $-z_R$ and z_R from the beam waist:

$$\begin{aligned} \text{inside} &\leftrightarrow |z - z(w)| \leq z_R \\ \text{outside} &\leftrightarrow |z - z(w)| > z_R \end{aligned} \tag{1.31}$$

The situation is described in the below figure from Lawrence et al., Applied Optics and Optical Engineering, Volume XI (1992).



Explicitly, these possibilities are:

1. II(z_1, z_2): inside RR to inside RR
2. IO(z_1, z_2): inside RR to outside RR
3. OI(z_1, z_2): outside RR to inside RR
4. OO(z_1, z_2): outside RR to outside RR

To move from any point in space to any other, following Lawrence et al., Applied Optics and Optical Engineering, Volume XI (1992), *PAOS* implements three primitive operators:

1. plane-to-plane (PTP)
2. waist-to-spherical (WTS)
3. spherical-to-waist (STW)

Using these primitive operators, *PAOS* implements all possible propagations:

1. II(z_1, z_2) = PTP($z_2 - z_1$)
2. IO(z_1, z_2) = WTS($z_2 - z(w)$) PTP($z_2 - z(w)$)
3. OI(z_1, z_2) = PTP($z_2 - z(w)$) STW($z_2 - z(w)$)
4. OO(z_1, z_2) = WTS($z_2 - z(w)$) STW($z_2 - z(w)$)

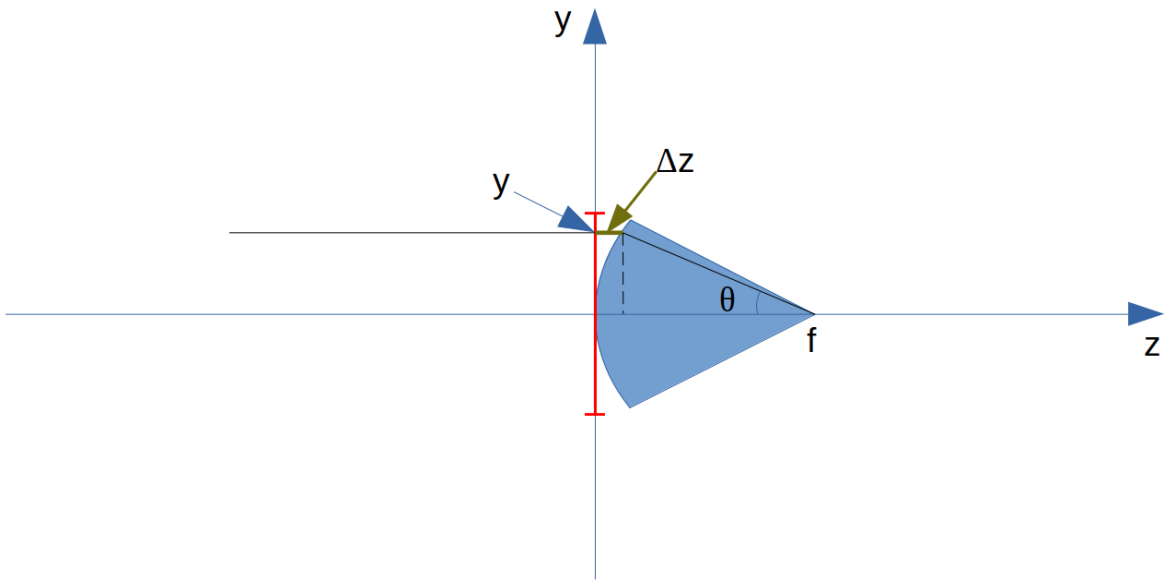
```
thickness = 10.0e-3  # m
wfo.propagate(dz = thickness)
```

Wavefront phase

A lens modifies the phase of an incoming beam.

Consider a monochromatic collimated beam travelling with slope $u = 0$, incident on a paraxial lens, orthogonal to the direction of propagation of the beam. The planar beam is transformed into a converging or diverging beam. That means, a spherical wavefront with curvature > 0 for a converging beam, or a < 0 for a diverging beam.

The convergent beam situation is described by the diagram below.



where:

1. the paraxial lens is coloured in red
2. the converging beam cone is coloured in blue
3. the incoming beam intersects the lens at a coordinate y

and

1. z is the propagation axis (> 0 at the right of the lens)
2. f is the optical focal length
3. Δz is the sag
4. θ is the angle corresponding to the sag

Δz depends from the x and y coordinates, and it introduces a delay in the complex wavefront $a_1(x, y, z) = e^{2\pi j z/\lambda}$ incident on the lens ($z = 0$ can be assumed). That is:

$$a_2(x, y, z) = a_1(x, y, z)e^{2\pi j \Delta z/\lambda} \tag{1.32}$$

The sag can be estimated using the Pythagoras theorem and evaluated in small angle approximation, that is

$$\Delta z = f - \sqrt{f^2 - y^2} \simeq \frac{y^2}{2f} \tag{1.33}$$

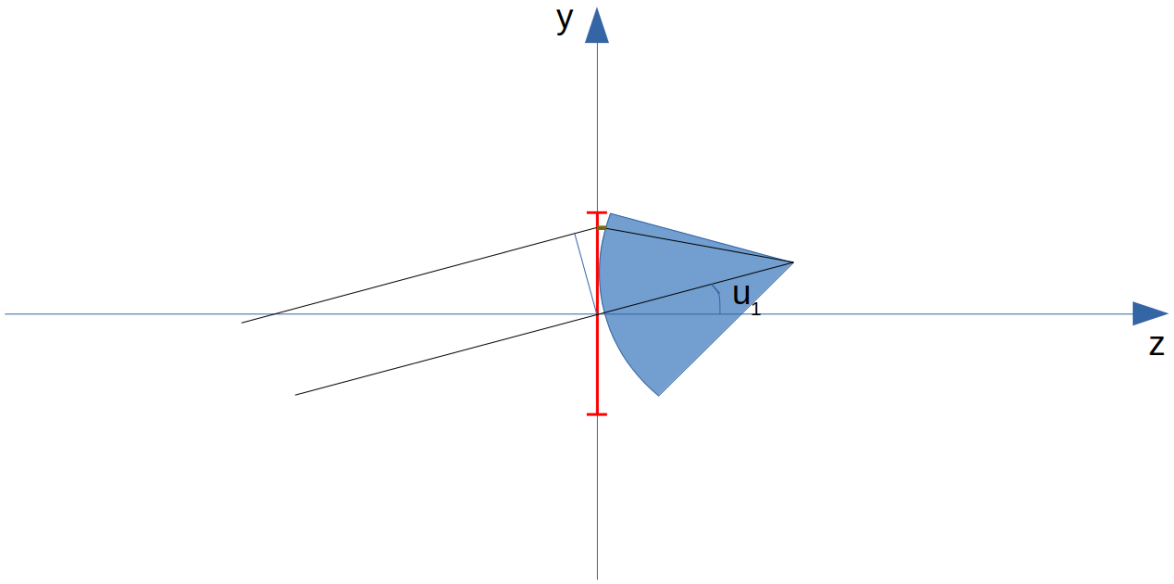
The phase delay over the whole lens aperture is then

$$\Delta\Phi = -\Delta z/\lambda = -\frac{x^2 + y^2}{2f\lambda} \tag{1.34}$$

Sloped incoming beam

When the incoming collimated beam has a slope u_1, its phase on the plane of the lens is given by $e^{2\pi j y u_1/\lambda}$ to which the lens adds a spherical sag.

This situation is described by the diagram below.



The total phase delay is then

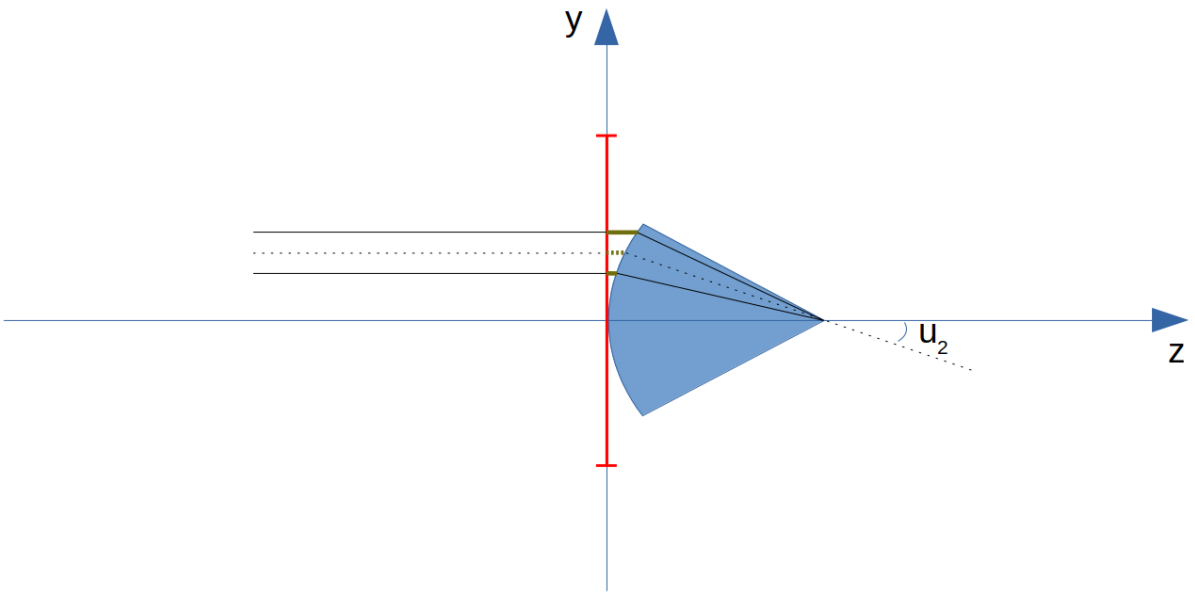
$$\Delta\Phi = -\frac{x^2+y^2}{2f\lambda} + \frac{yu_1}{\lambda} = -\frac{x^2+(y-fu_1)^2}{2f\lambda} + \frac{yu_1^2}{2\lambda} = -\frac{x^2+(y-y_0)^2}{2f\lambda} + \frac{y_0^2}{2f\lambda} \tag{1.35}$$

Apart from the constant phase term, that can be neglected, this is a spherical wavefront centred in $(0, y_0, f)$, with $y_0 = fu_1$.

Note: In this approximation, the focal plane is planar.

Off-axis incoming beam

The case of off-axis optics is described in the diagram below.



In this case, the beam centre is at y_c.

Let δy be a displacement from y_c along y. The lens induced phase change is then

$$\Delta\Phi = -\frac{x^2+y^2}{2f\lambda} = -\frac{x^2+(y_c-\delta y)^2}{2f\lambda} = -\frac{x^2+\delta y^2}{2f\lambda} + \frac{\delta y u_2}{\lambda} - \frac{y_c^2}{2f\lambda} \tag{1.36}$$

If the incoming beam has a slope u_1, then

$$\Delta\Phi = -\frac{x^2+\delta y^2}{2f\lambda} + \frac{\delta y(u_1+u_2)}{\lambda} - \frac{y_c^2}{2f\lambda} + y_c u_1 \tag{1.37}$$

Apart from constant phase terms, that can be neglected, this is equivalent to a beam that is incident on-axis on the lens. The overall slope shifts the focal point in a planar focal plane. No aberrations are introduced.

Paraxial phase correction

For an optical element that can be modeled using its focal length f (that is, mirrors, thin lenses and refractive surfaces), the paraxial phase effect is

$$t(x,y) = e^{jk\left(x^2+y^2\right)/2f}$$

where t(x, y) is the complex transmission function. In other words, the element imposes a quadratic phase shift. The phase shift depends on initial and final position with respect to the Rayleigh range (see *Wavefront propagation*).

As usual, in *PAOS* this is informed by the Gaussian beam parameters. The code implementation consists of four steps:

1. estimate the Gaussian beam curvature after the element (object space) using Eq. (1.24)
2. check the initial position using Eq. (1.31)
3. estimate the Gaussian beam curvature after the element (image space)
4. check the final position

By combining the result of the second and the fourth step, *PAOS* selects the propagator (see *Wavefront propagation*). and the phase shift is imposed accordingly by defining a phase bias (see Lawrence et al., Applied Optics and Optical Engineering, Volume XI (1992)):

Propagator	Phase bias	Description
II	$1/f \rightarrow 1/f$	No phase bias
IO	$1/f \rightarrow 1/f + 1/R'$	Phase bias after lens
OI	$1/f \rightarrow 1/f - 1/R$	Phase bias before lens
OO	$1/f \rightarrow 1/f - 1/R + 1/R'$	Phase bias before and after lens

where R is the radius of curvature in object space and R' in image space.

Apertures

The actual wavefront propagated through an optical system intersects real optical elements (e.g. mirrors, lenses, slits) and can be obstructed by an object causing an obscuration.

For each one of these cases, *PAOS* implements an appropriate aperture mask. The aperture must be projected on the plane orthogonal to the beam. If the aperture is (y_c, ϕ_x, ϕ_y), the aperture should be set as

$$\left(y_a - y_c, \phi_x, \frac{1}{\sqrt{u^2+1}}\phi_y\right)$$

Supported aperture shapes are elliptical, circular or rectangular.

```
xrad *= np.sqrt(1 / (vs[1] ** 2 + 1))
yrad *= np.sqrt(1 / (vt[1] ** 2 + 1))
xaper = xdec - vs[0]
yaper = ydec - vt[0]

aperture_shape = 'elliptical'  # or 'rectangular'
obscuration = False  # if True, applies obscuration

aperture = wfo.aperture(xaper, yaper, hx=xrad, hy=yrad,
                        shape=aperture_shape, obscuration=obscuration)
```

Stops

An aperture stop is an element of an optical system that determines how much light reaches the image plane. It is often the boundary of the primary mirror. An aperture stop has an important effect on the sizes of system aberrations.

The field stop limits the field of view of an optical instrument.

PAOS implements a generic stop normalizing the wavefront at the current position to unit energy.

```
wfo.make_stop()
```

POP propagation loop

PAOS implements the POP simulation through all elements of an optical system. The simulation run is implemented in a single loop.

At first, *PAOS* initializes the beam at the centre of the aperture. Then, it initializes the ABCD matrix.

Once the initialization is completed, *PAOS* repeats these actions in a loop:

1. Apply coordinate break
2. Apply aperture
3. Apply stop
4. Apply aberration (see *Aberration description*)
5. Apply ABCD matrix and update
6. Apply magnification
7. Apply lens
8. Apply propagation thickness
9. Update ABCD matrix
10. Repeat over all optical elements

Note: Each action is performed according to the configuration file, see *Input system*.

Aberration description

Zernike polynomials

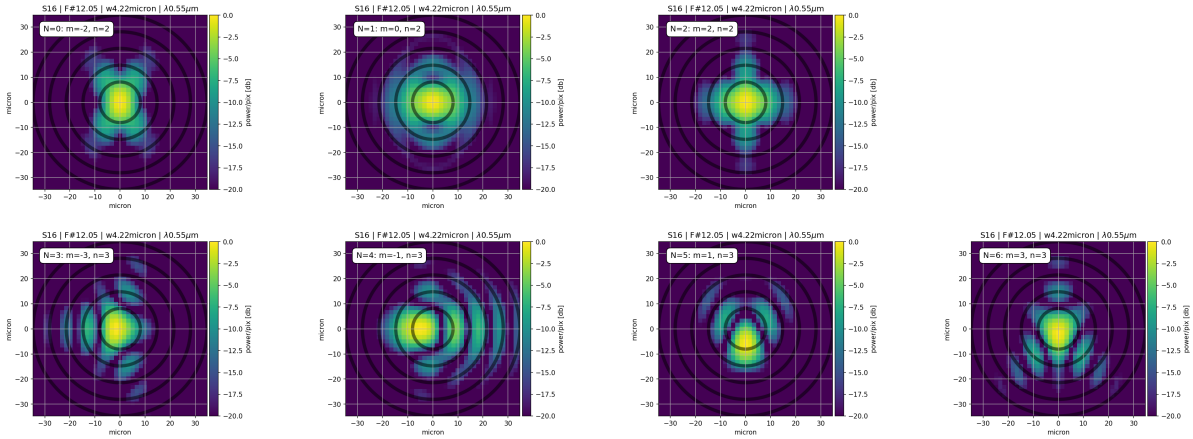
PAOS models an aberration using a series of Zernike polynomials, up to a specified radial order.

PAOS can generate both ortho-normal polynomials and orthogonal polynomials as described in Laksminarayan & Fleck, Journal of Modern Optics (2011)

The ordering can be either ANSI (default), or Noll, or Fringe, or Standard (Born&Wolf).

Example of an aberrated pupil

An example of aberrated PSFs at the *Ariel* Telescope exit pupil is shown below.



In this figure, the same Surface Form Error (SFE) of 50 nm root mean square (r.m.s.) is allocated to different aberrations, modeled as Zernike Polynomials. Starting from the top left panel (oblique Astigmatism), seven such simulations are shown, in ascending Ansi order. Each aberration differs in the impact on the Telescope optical quality: some (e.g. Coma) require a more stringent allocation to be compatible with the mission scientific requirements.

Strehl ratio

Encircled energy

Materials description

Supported materials

```
from paos.util.material import Material

wl = 1.95  # micron
mat = Material(wl)
print('Supported materials: ')
print(*mat.materials.keys(), sep = "\n")
```

Sellmeier equation

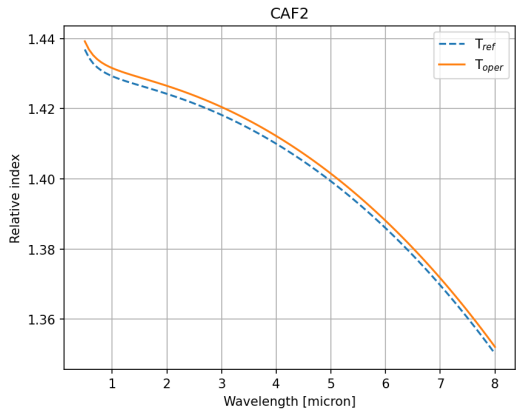
Index of refraction

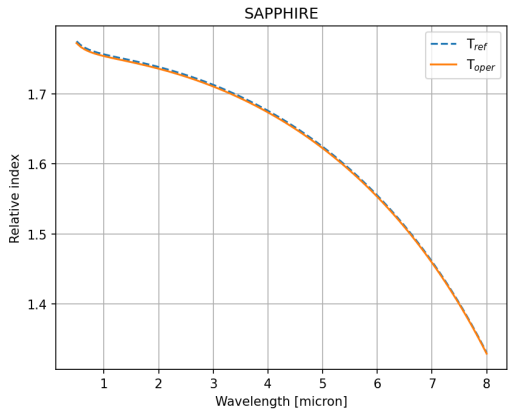
Example

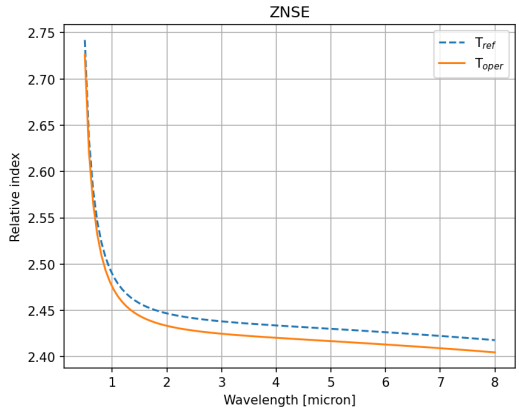
```
from paos.util.material import Material

wl = 1.95  # micron
mat = Material(wl)
glass = 'bk7'
print('absolute index of refraction {:.4f} \nindex relative to air {:.4f}'.format(
    *mat.nmat(glass)), sep = "\n")
```

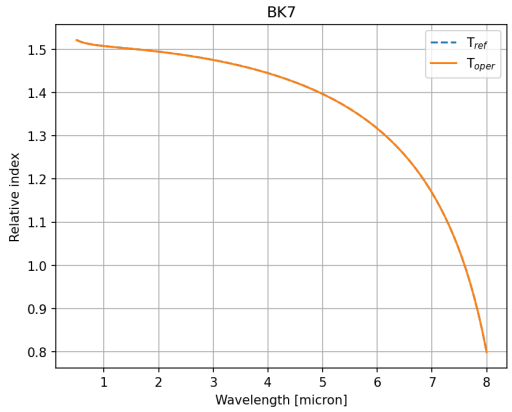
Plotting

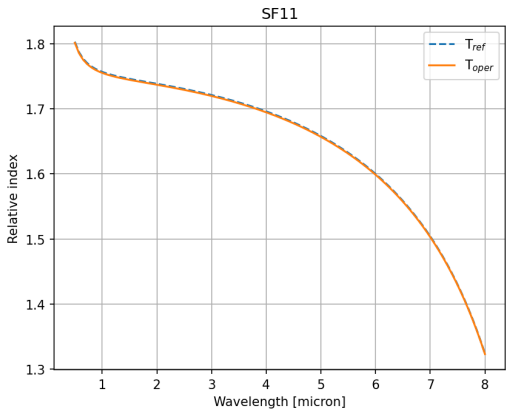
```
import numpy as np
from paos.util.material import Material

wl = np.linspace(0.5, 8.0, 100)  # micron
mat = Material(wl)
mat.plot_relative_index(material_list=mat.materials.keys())
```

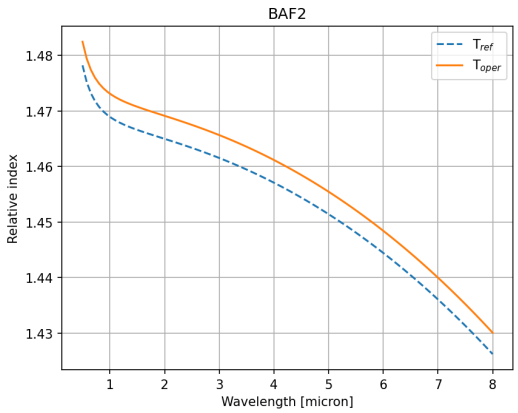








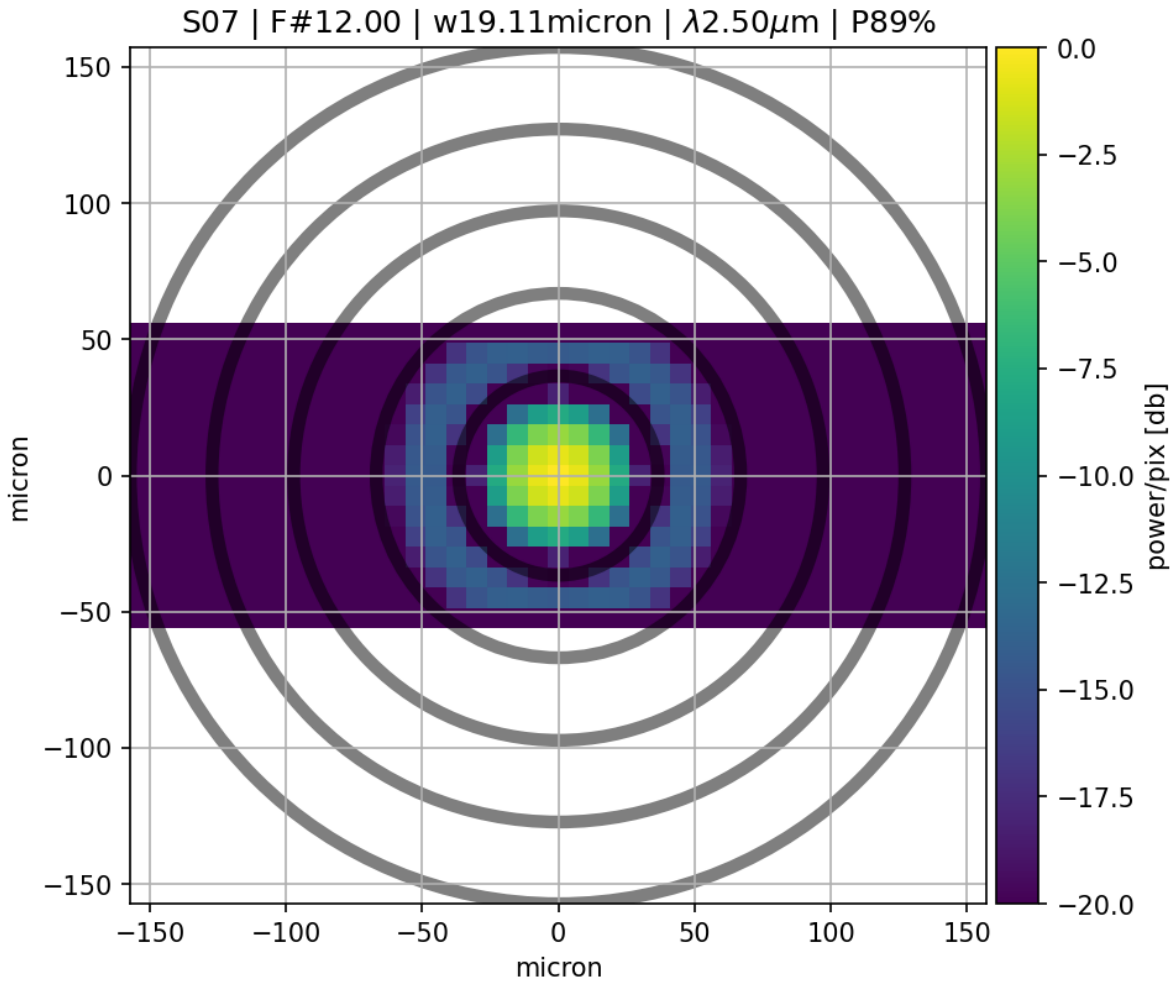




Plotting results

Base plot

Given the POP simulation output dict, plots the squared amplitude of the wavefront at the given optical surface.



Example

```
from matplotlib import pyplot as plt
from paos.paos_parseconfig import ParseConfig
from paos.paos_run import run
from paos.paos_plotpop import simple_plot

pup_diameter, general, fields, opt_chain = ParseConfig('path/to/conf/file')
ret_val = run(pup_diameter, 1.0e-6 * general['wavelength'], general['grid size'],
general['zoom'], fields['0'], opt_chain)
fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(8, 8))
key = list(ret_val.keys())[-1]  # plot at last optical surface
item = ret_val[key]
```

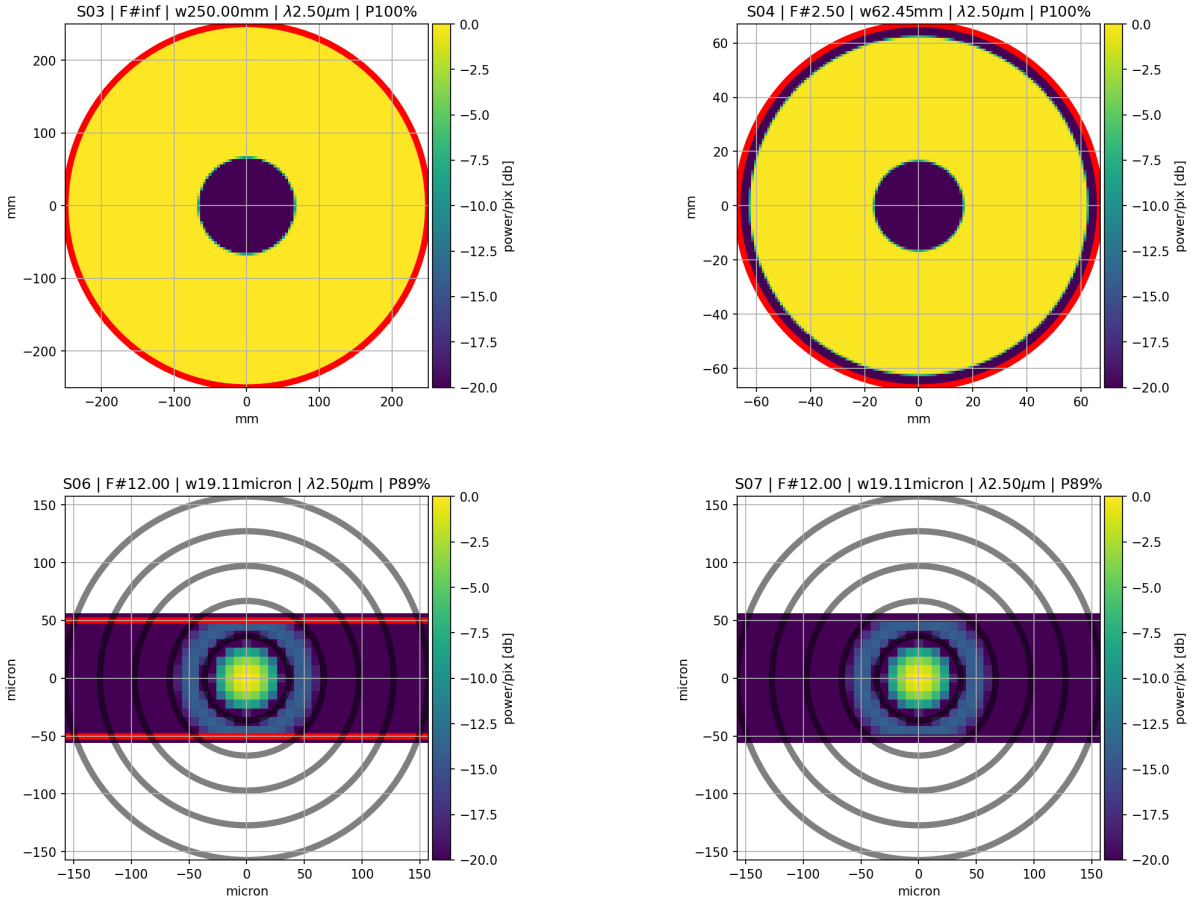
(continues on next page)

(continued from previous page)

```
simple_plot(fig, ax, key=key, item=item, ima_scale='log')
plt.show()
```

POP plot

Given the POP simulation output dict, plots the squared amplitude of the wavefront at all the optical surfaces.



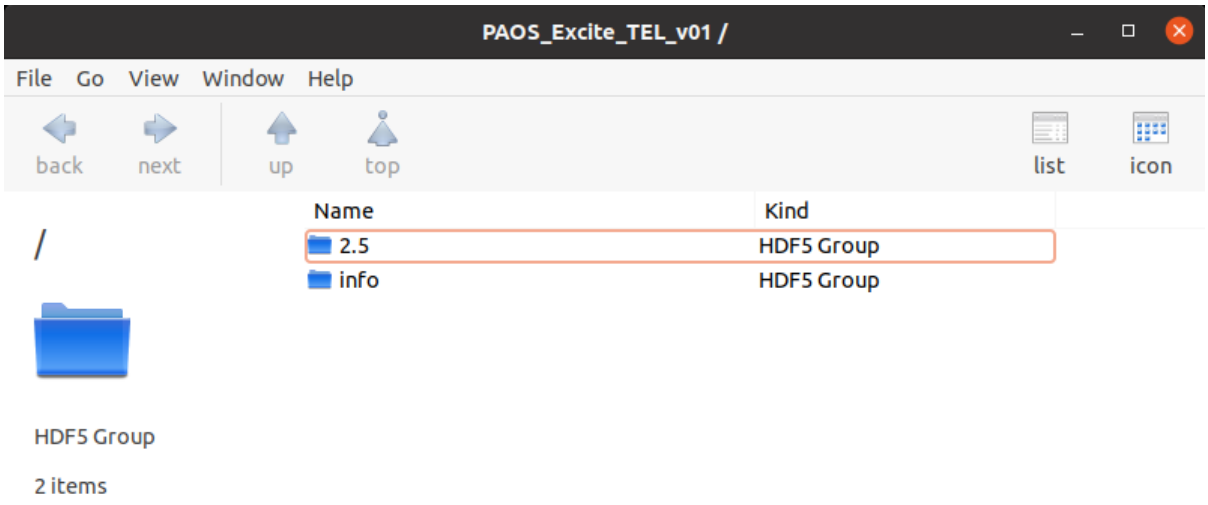
Example

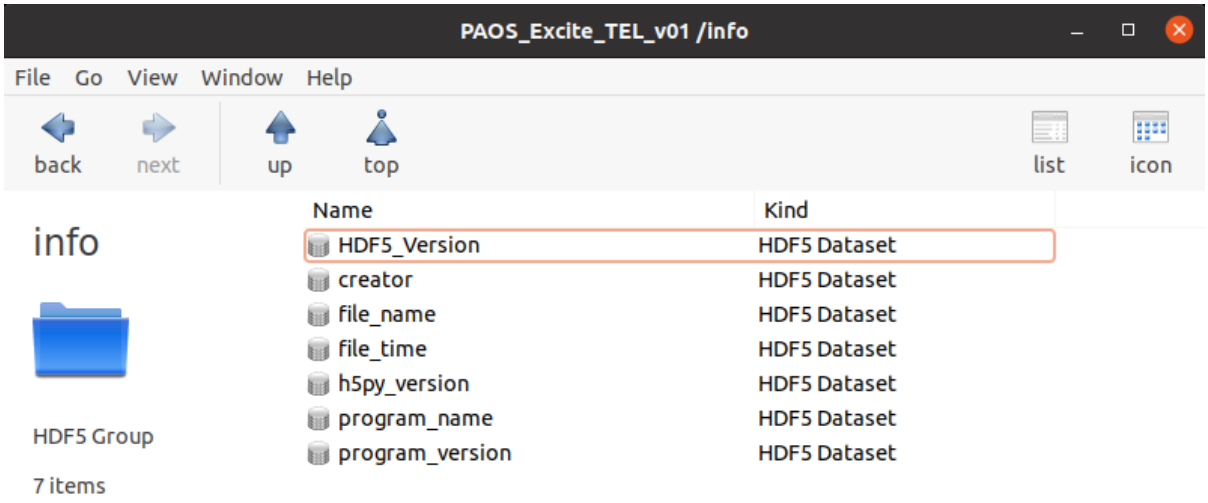
```
from paos.paos_parseconfig import ParseConfig
from paos.paos_run import run
from paos.paos_plotpop import plot_pop

pup_diameter, general, fields, opt_chain = ParseConfig('path/to/conf/file')
ret_val = run(pup_diameter, 1.0e-6 * general['wavelength'], general['grid size'],
general['zoom'], fields['0'], opt_chain)
plot_pop(ret_val, ima_scale='log', ncols=3, figname='path/to/output/plot')
```

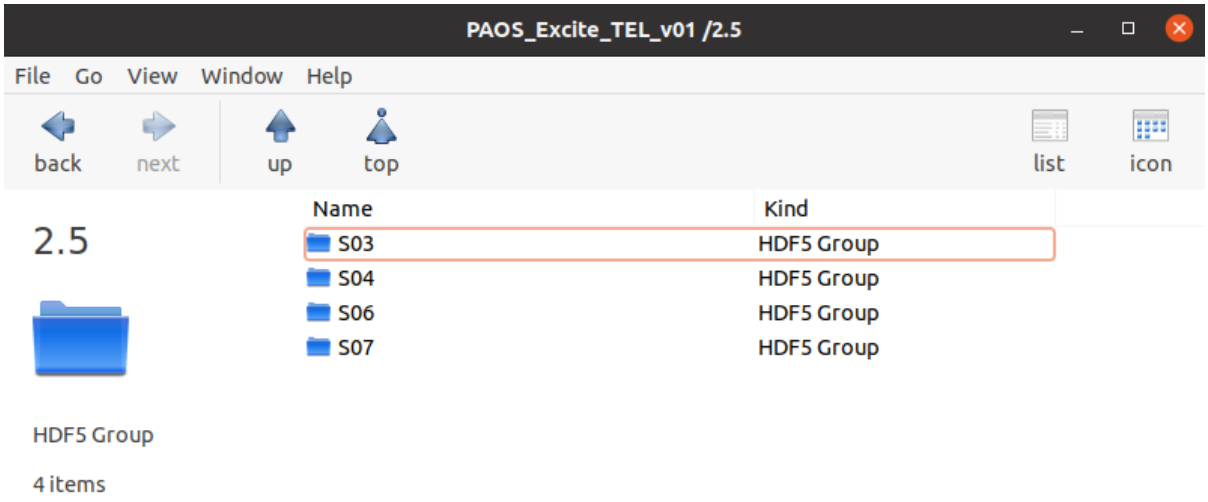
Saving results

Save output

Given the POP simulation output dictionary, a hdf5 file name and the keys to store at each surface, it saves the output dictionary along with the *PAOS* package information to the hdf5 output file. If indicated, overwrites past output file.





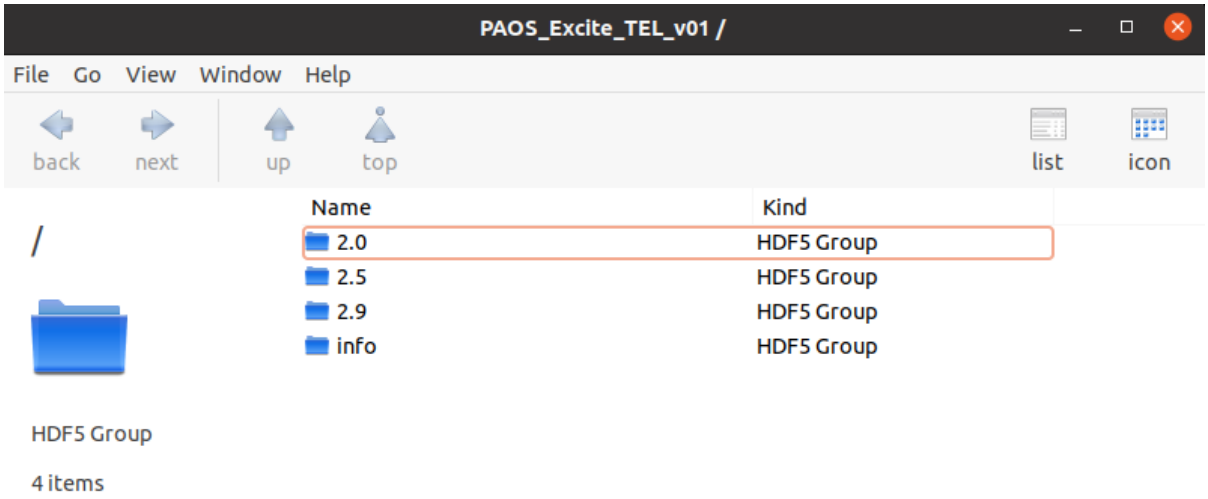


Example

```
from paos.paos_parseconfig import ParseConfig
from paos.paos_run import run
from paos.paos_saveoutput import save_output

pup_diameter, general, fields, opt_chain = ParseConfig('path/to/conf/file')
ret_val = run(pup_diameter, 1.0e-6 * general['wavelength'], general['grid size'],
          general['zoom'], fields['0'], opt_chain)
save_output(ret_val, 'path/to/hdf5/file', keys_to_keep=['wfo', 'dx', 'dy'],
         overwrite=True)
```

Save datacube

Given a list of dictionaries with POP simulation output, a hdf5 file name, a list of identifiers to tag each simulation and the keys to store at each surface, it saves the outputs to a data cube along with the *PAOS* package information to the hdf5 output file. If indicated, overwrites past output file.



Example

```
from paos.paos_parseconfig import ParseConfig
from paos.paos_run import run
from paos.paos_saveoutput import save_datacube
from joblib import Parallel, delayed
from tqdm import tqdm
pup_diameter, general, fields, opt_chain = ParseConfig('path/to/conf/file')
wavelengths = [1.95, 3.9]
ret_val_list = Parallel(n_jobs=2)(delayed(run)(pup_diameter, 1.0e-6 * wl, general['grid␣
↪size'],
              general['zoom'], fields['0'], opt_chain) for wl in tqdm(wavelengths))
group_tags = list(map(str, wavelengths))
save_datacube(ret_val_list, 'path/to/hdf5/file', group_tags,
              keys_to_keep=['amplitude', 'dx', 'dy'], overwrite=True)
```

Automatic pipeline

Pipeline to run a POP simulation and save the results, given the input dictionary.

Base pipeline

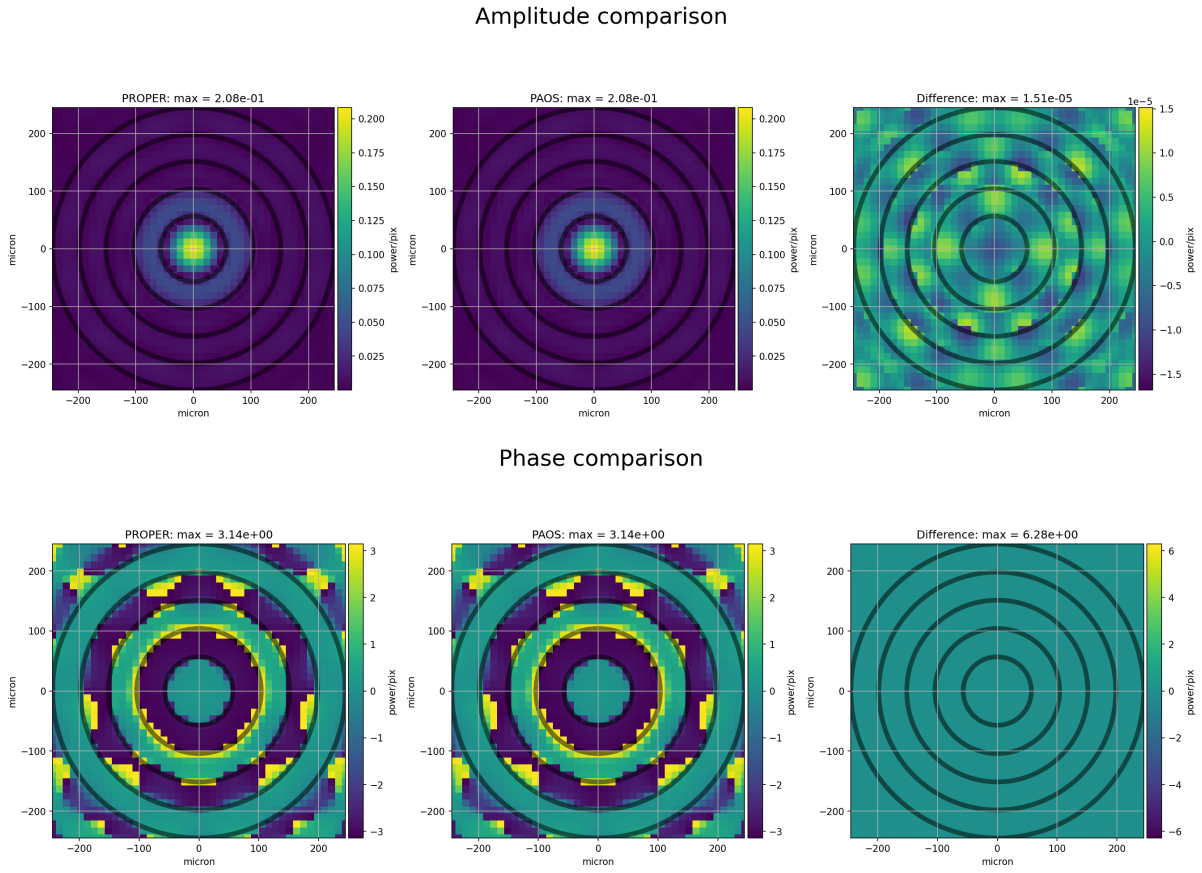
This pipeline parses the lens file, performs a diagnostic ray tracing (optional), sets up the simulation wavelength or produces a user defined wavelength grid, sets up the optical chain for the POP run automatizing the input of an aberration (optional), runs the POP in parallel or using a single thread and produces an output where all (or a subset) of the products are stored. If indicated, the output includes plots.

Example

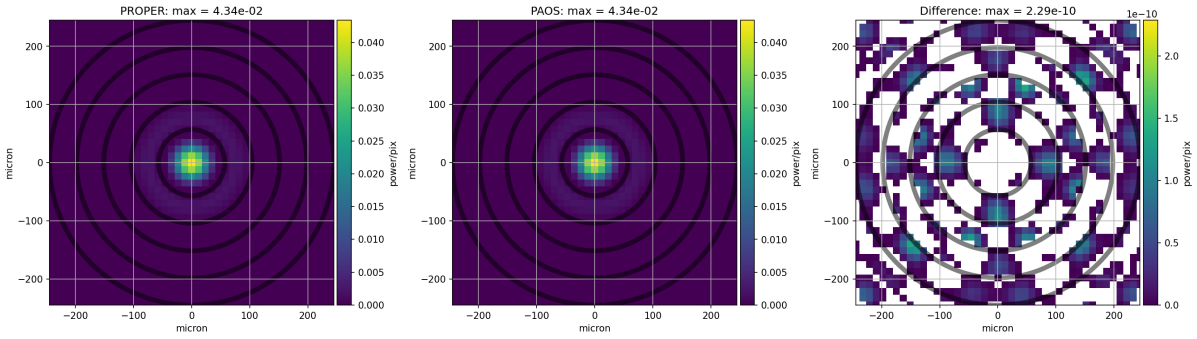
```
from paos.paos_pipeline import pipeline

pipeline(passvalue={'conf':'path/to/conf/file',
                    'output': 'path/to/output/file',
                    'wavelengths': '1.95,3.9',
                    'plot': True,
                    'loglevel': 'debug',
                    'n_jobs': 2,
                    'store_keys': 'amplitude,dx,dy,wl',
                    'return': False})
```

1.4 Validation

PAOS has been validated against the code PROPER (John E. Krist, PROPER: an optical propagation library for IDL, Proc. SPIE, 6675 (2007)).



Psf comparison



1.5 Ariel

This section describes *PAOS* in the context of the *Ariel* space mission.

In particular, we discuss:

1. which are the main use cases of *PAOS* for *Ariel* (*Use of PAOS*)
2. how *PAOS* is compatible with other *Ariel* simulators (*Compatibility of PAOS*)

1.5.1 Table of Contents

Use of PAOS

PAOS was developed to study the effect of diffraction and aberrations impacting the *Ariel* optical performance and related systematics.

The Ariel telescope is required to provide diffraction-limited capabilities at wavelengths longer than $3\ \mu m$. While the Ariel measurements do not require high imaging quality, they still need to have an efficient light bucket, i.e. to collect photons in a sufficiently compact region of the focal plane.

PAOS was developed to demonstrate that even at wavelengths where Ariel is not diffraction-limited it still delivers high quality data for scientific analysis.

PAOS can perform a large number of detailed analyses for any optical system for which the Fresnel approximation holds (see *Fresnel diffraction theory*).

For *Ariel*, it can be used both on the instrument side and on the optimization of the science return from the space mission. In this section, a number of those uses are briefly described.

Aberrations

PAOS can be used to quantitatively analyze the level of aberrations compatible with the scientific requirements of the *Ariel* mission, for example to support the maximum amplitude of the aberrations compatible with the scientific requirement for the manufacturing of the *Ariel* telescope primary mirror (M1). *PAOS* can study the effect of the aberrations to ensure that the optical quality, complexity, costs and risks are not too high.

SNR

placeholder

Gain noise

PAOS can evaluate in a representative way the impact of optical diaphragms on the photometric error in presence of pointing jitter and the impact of thermo-elastic variations on optical efficiency during the flight. This impact has not been completely quantified, especially at small wavelengths where the optical aberrations dominate and the Optical Transfer Function (OTF) is very sensitive to their variation.

Pointing

PAOS can be used to verify that the realistic PSFs of the mission are compatible with the scientific requirements. For example, the FGS instrument (operating well under the diffraction limit) uses the stellar photons to determine the pointing fluctuations by calculating the centroid of the star position. This data is then used to stabilize the spacecraft through the Attitude Orbit Control System (AOCS) of the spacecraft bus. If the PSF is too aberrated, the centroid might have large errors, impacting the pointing stability and the measurement. Therefore, by delivering realistic PSFs of the mission, *PAOS* can be used to estimate whether the pointing stability is compatible with the scientific requirements, before having a system-level measurement that will not be ready for several years.

Calibration

PAOS can be used as a test bed to develop strategies for ground and in-flight calibration. For instance, simulations of the re-focussing mechanism behind the M2 mirror (M2M), which uses actuators on the M2 mirror to allow correction for misalignment generated during telescope assembly or launch and cool-down to operating temperatures. These simulations could inform the best strategy to optimize the telescope focussing ahead of payload delivery, ensuring that the optical system stays focussed and satisfies the requirement on the maximum amplitude of the wavefront aberrations during the measurements.

Compatibility of *PAOS*

PAOS is compatible with existing *Ariel* simulators, such as ArielRad (L. V. Mugnai, et al. ArielRad: the Ariel radiometric model. Exp Astron 50, 303–328 (2020)), and Exosim (Sarkar, S., Pascale, E., Papageorgiou, A. et al. ExoSim: the Exoplanet Observation Simulator. Exp Astron 51, 287–317 (2021)).

Exosim is an end-to-end simulator that models noise and systematics in a dynamical simulation and can capture temporal effects, such as correlated noise and systematics on the light curve. *PAOS* can provide Exosim with representative aberrated PSFs that Exosim can use to create realistic images on the focal planes of the instruments.

1.6 API Guide

1.6.1 *PAOS* package

In the following you find the documentation for the main classes and modules defined in *PAOS*.

Modules

paos.paos_abcd module

class ABCD(*thickness=0.0*, *curvature=0.0*, *n1=1.0*, *n2=1.0*, *M=1.0*)

Bases: `object`

ABCD matrix class for paraxial ray tracing.

Variables

- **thickness** (*scalar*) – optical thickness
- **power** (*scalar*) – optical power
- **M** (*scalar*) – optical magnification
- **n1n2** (*scalar*) – ratio of refractive indices n1/n2 for light propagating from a medium with refractive index n1, into a medium with refractive index b2
- **c** (*scalar*) – speed of light. Can take values +1 for light travelling left-to-right (+Z), and -1 for light travelling right-to-left (-Z)

Note: The class properties can differ from the value of the parameters used at class instantiation. This because the ABCD matrix is decomposed into four primitives, multiplied together as discussed in *Optical system equivalent*.

Examples

```
>>> from paos.paos_abcd import ABCD
>>> thickness = 2.695  # mm
>>> radius = 31.850  # mm
>>> n1, n2 = 1.0, 1.5
>>> abcd = ABCD(thickness=thickness, curvature=1.0/radius, n1=n1, n2=n2)
>>> (A, B), (C, D) = abcd.ABCD
```

Initialize the ABCD matrix.

Parameters

- **thickness** (*scalar*) – optical thickness. It is positive from left to right. Default is 0.0
- **curvature** (*scalar*) – inverse of the radius of curvature: it is positive if the center of curvature lies on the right. If n1=n2, the parameter is assumed describing a thin lens of focal ratio fl=1/curvature. Default is 0.0
- **n1** (*scalar*) – refractive index of the first medium. Default is 1.0
- **n2** (*scalar*) – refractive index of the second medium. Default is 1.0

- **M** (*scalar*) – optical magnification. Default is 1.0

Note: Light is assumed to be propagating from a medium with refractive index n1 into a medium with refractive index n2.

Note: The refractive indices are assumed to be positive when light propagates from left to right (+Z), and negative when light propagates from right to left (-Z)

---

property thickness

property M

property n1n2

property power

property cin

property cout

property f_eff

property ABCD

paos.paos_coordinatebreak module

CoordinateBreak(*vt*, *vs*, *xdec*, *ydec*, *xrot*, *yrot*, *zrot*, *order=0*)

Performs a coordinate break and estimates the new $\vec{v_t} = (y, u_y)$ and $\vec{v_s} = (x, u_x)$.

Parameters

- **vt** (*array*) – vector $\vec{v_t} = (y, u_y)$ describing a ray propagating in the tangential plane
- **vs** (*array*) – vector $\vec{v_s} = (x, u_x)$ describing a ray propagating in the sagittal plane
- **xdec** (*float*) – x coordinate of the decenter to be applied
- **ydec** (*float*) – y coordinate of the decenter to be applied
- **xrot** (*float*) – tilt angle around the X axis to be applied
- **yrot** (*float*) – tilt angle around the Y axis to be applied
- **zrot** (*float*) – tilt angle around the Z axis to be applied
- **order** (*int*) – order of the coordinate break, defaults to 0.

Returns two arrays representing the new $\vec{v_t} = (y, u_y)$ and $\vec{v_s} = (x, u_x)$.

Return type tuple

Note: When order=0, first a coordinate decenter is applied, followed by a XYZ rotation. Coordinate break orders other than 0 not implemented yet.

---

paos.paos_parseconfig module

ReadConfig(*filename*)

Given the input file name, it parses the simulation parameters and returns a dictionary. The input file is an Excel spreadsheet which contains three data sheets named 'general', 'LD' and 'field'. 'general' contains the simulation wavelength, grid size and zoom, defined as the ratio of grid size to initial beam size in unit of pixel. 'LD' is the lens data and contains the sequence of surfaces for the simulation mimicking a Zemax lens data editor: supported surfaces include coordinate break, standard surface, obscuration, zernike, paraxial lens, slit and prism.

Returns dictionary with the parsed input parameters for the simulation.

Return type dict

Examples

```
>>> from paos.paos_parseconfig import ReadConfig
>>> simulation_parameters = ReadConfig('path/to/conf/file')
```

ParseConfig(*filename*)

It parses the input file name and returns the input pupil diameter and three dictionaries for the simulation: one for the general parameters, one for the input fields and one for the optical chain.

Returns the input pupil diameter, the general parameters, the input fields and the optical chain.

Return type dict

Examples

```
>>> from paos.paos_parseconfig import ParseConfig
>>> pupil_diameter, general, fields, optical_chain = ParseConfig('path/to/conf/file
↪')
```

paos.paos_pipeline module

pipeline(*passvalue*)

Pipeline to run a POP simulation and save the results, given the input dictionary. This pipeline parses the lens file, performs a diagnostic ray tracing (optional), sets up the simulation wavelength or produces a user defined wavelength grid, sets up the optical chain for the POP run automatizing the input of an aberration (optional), runs the POP in parallel or using a single thread and produces an output where all (or a subset) of the products are stored. If indicated, the output includes plots.

Parameters **passvalue** (*dict*) – input dictionary for the simulation

Returns If indicated, returns the simulation output dictionary or a list with a dictionary for each simulation. Otherwise, returns None.

Return type None or dict or list of dict

Examples

```
>>> from paos.paos_pipeline import pipeline
>>> pipeline(passvalue={'conf':'path/to/conf/file',
>>>                     'output': 'path/to/output/file',
>>>                     'wavelengths': '1.95,3.9',
>>>                     'plot': True,
>>>                     'loglevel': 'debug',
>>>                     'n_jobs': 2,
>>>                     'store_keys': 'amplitude,dx,dy,wl',
>>>                     'return': False})
```

paos.paos_plotpop module

simple_plot(*fig*, *axis*, *key*, *item*, *ima_scale*, *surface_zoom={}*)

Given the POP simulation output dict, plots the squared amplitude of the wavefront at the given optical surface.

Parameters

- **fig** (*~matplotlib.figure.Figure*) – instance of matplotlib figure artist
- **axis** (*~matplotlib.axes.Axes*) – instance of matplotlib axes artist
- **key** (*int*) – optical surface index
- **item** (*dict*) – optical surface dict
- **ima_scale** (*str*) – plot color map scale, can be either 'linear' or 'log'
- **surface_zoom** (*dict*) – dict containing the zoom scale to display the plot

Returns displays the plot output or stores it to the indicated plot path

Return type None

Examples

```
>>> from paos.paos_parseconfig import ParseConfig
>>> from paos.paos_run import run
>>> from paos.paos_plotpop import simple_plot
>>> pup_diameter, general, fields, opt_chain = ParseConfig('path/to/conf/file')
>>> ret_val = run(pup_diameter, 1.0e-6 * general['wavelength'], general['grid size
↪'],
>>>               general['zoom'], fields['0'], opt_chain)
>>> from matplotlib import pyplot as plt
>>> fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(8, 8))
>>> key = list(ret_val.keys())[-1]  # plot at last optical surface
>>> item = ret_val[key]
>>> simple_plot(fig, ax, key=key, item=item, ima_scale='log')
>>> plt.show()
```

plot_pop(*retval*, *ima_scale='log'*, *ncols=2*, *figname=None*, *surface_zoom={}*)

Given the POP simulation output dict, plots the squared amplitude of the wavefront at all the optical surfaces.

Parameters

- **retval** (*dict*) – simulation output dictionary

- **ima_scale** (*str*) – plot color map scale, can be either 'linear' or 'log'
- **ncols** (*int*) – number of columns for the subplots
- **figname** (*str*) – name of figure to save
- **surface_zoom** (*dict*) – dict containing the zoom scale to display the plot

Returns displays the plot output or stores it to the indicated plot path

Return type None

Examples

```
>>> from paos.paos_parseconfig import ParseConfig
>>> from paos.paos_run import run
>>> from paos.paos_plotpop import plot_pop
>>> pup_diameter, general, fields, opt_chain = ParseConfig('path/to/conf/file')
>>> ret_val = run(pup_diameter, 1.0e-6 * general['wavelength'], general['grid size
↪'],
>>>               general['zoom'], fields['0'], opt_chain)
>>> plot_pop(ret_val, ima_scale='log', ncols=3, figname='path/to/output/plot')
```

paos.paos_raytrace module

raytrace(*field*, *opt_chain*)

Diagnostic function that implements the full ray tracing and prints the output for each surface of the optical chain as the ray positions and slopes in the tangential and sagittal planes.

Parameters

- **field** (*dictionary*) – contains the slopes in the tangential and sagittal planes as field={'vt': slopey, 'vs': slopex}
- **opt_chain** (*list*) – the list of the optical elements returned by paos.parseconfig

Returns prints the output of the full ray tracing using the logger.

Return type None

Examples

```
>>> from paos.paos_parseconfig import ParseConfig
>>> from paos.paos_raytrace import raytrace
>>> pupil_diameter, general, fields, optical_chain = ParseConfig('path/to/conf/file
↪')
>>> raytrace(fields['0'], optical_chain)
```

paos.paos_run module

push_results(*wfo*)

run(*pupil_diameter*, *wavelength*, *gridsize*, *zoom*, *field*, *opt_chain*)
Run the POP.

Parameters

- **pupil_diameter** (*scalar*) – input pupil diameter in meters
- **wavelength** (*scalar*) – wavelength in meters
- **gridsize** (*scalar*) – the size of the simulation grid. It has to be a power of 2
- **zoom** (*scalar*) – zoom factor
- **field** (*dictionary*) – contains the slopes in the tangential and sagittal planes as field={'vt': slopey, 'vs': slopex}
- **opt_chain** (*list*) – the list of the optical elements parsed by paos.paos_parseconfig.ParseConfig

Returns **out** – dictionary containing the results of the POP

Return type dict

Examples

```
>>> from paos.paos_parseconfig import ParseConfig
>>> from paos.paos_run import run
>>> from paos.paos_plotpop import simple_plot
>>> pup_diameter, general, fields, optical_chain = ParseConfig('path/to/conf/file')
>>> ret_val = run(pup_diameter, 1.0e-6 * general['wavelength'], general['grid size
↪'],
>>>               general['zoom'], fields['0'], optical_chain)
```

paos.paos_saveoutput module

remove_keys(*dictionary*, *keys*)
Removes item at specified index from dictionary.

Parameters

- **dictionary** (*dict*) – input dictionary
- **keys** – keys to remove from the input dictionary

Returns Updates the input dictionary by removing specific keys

Return type None

Examples

```
>>> from paos.paos_saveoutput import remove_keys
>>> my_dict = {'a': 1, 'b': 2, 'c': 3, 'd': 4}
>>> print(my_dict)
>>> keys_to_drop = ['a', 'c', 'e']
>>> remove_keys(my_dict, keys_to_drop)
>>> print(my_dict)
```

save_recursively_to_hdf5(*dictionary*, *outgroup*)
Given a dictionary and a hdf5 object, saves the dictionary to the hdf5 object.

Parameters

- **dictionary** (*dict*) – a dictionary instance to be stored in a hdf5 file
- **outgroup** – a hdf5 file object in which to store the dictionary instance

Returns Save the dictionary recursively to the hdf5 output file

Return type None

save_info(*file_name*, *out*)
Inspired by a similar function from ExoRad2. Given a hdf5 file name and a hdf5 file object, saves the information about the paos package to the hdf5 file object. This information includes the file name, the time of creation, the package creator names, the package name, the package version, the hdf5 package version and the h5py version.

Parameters

- **file_name** (*str*) – the hdf5 file name for saving the POP simulation
- **out** – the hdf5 file object

Returns Saves the paos package information to the hdf5 output file

Return type None

save_retval(*retval*, *keys_to_keep*, *out*)
Given the POP simulation output dictionary, the keys to store at each surface and the hdf5 file object, it saves the output dictionary to a hdf5 file.

Parameters

- **retval** (*dict*) – POP simulation output dictionary to be saved into hdf5 file
- **keys_to_keep** (*list*) – dictionary keys to store at each surface. example: ['amplitude', 'dx', 'dy']
- **out** (*~h5py.File*) – instance of hdf5 file object

Returns Saves the POP simulation output dictionary to the hdf5 output file

Return type None

save_output(*retval*, *file_name*, *keys_to_keep=None*, *overwrite=True*)
Given the POP simulation output dictionary, a hdf5 file name and the keys to store at each surface, it saves the output dictionary along with the paos package information to the hdf5 output file. If indicated, overwrites past output file.

Parameters

- **retval** (*dict*) – POP simulation output dictionary to be saved into hdf5 file
- **file_name** (*str*) – the hdf5 file name for saving the POP simulation

- **keys_to_keep** (*list*) – dictionary keys to store at each surface. example: ['amplitude', 'dx', 'dy']
- **overwrite** (*bool*) – if True, overwrites past output file

Returns Saves the POP simulation output dictionary along with the paos package information to the hdf5 output file

Return type None

Examples

```
>>> from paos.paos_parseconfig import ParseConfig
>>> from paos.paos_run import run
>>> from paos.paos_saveoutput import save_output
>>> pup_diameter, general, fields, opt_chain = ParseConfig('path/to/conf/file')
>>> ret_val = run(pup_diameter, 1.0e-6 * general['wavelength'], general['grid size
↪'],
>>>               general['zoom'], fields['0'], opt_chain)
>>> save_output(ret_val, 'path/to/hdf5/file', keys_to_keep=['wfo', 'dx', 'dy'],
>>>             overwrite=True)
```

save_datacube(*retval_list*, *file_name*, *group_names*, *keys_to_keep=None*, *overwrite=True*)

Given a list of dictionaries with POP simulation output, a hdf5 file name, a list of identifiers to tag each simulation and the keys to store at each surface, it saves the outputs to a data cube along with the paos package information to the hdf5 output file. If indicated, overwrites past output file.

Parameters

- **retval_list** (*list*) – list of dictionaries with POP simulation outputs to be saved into a single hdf5 file
- **file_name** (*str*) – the hdf5 file name for saving the POP simulation
- **group_names** (*list*) – list of strings with unique identifiers for each POP simulation. example: for one optical chain run at different wavelengths, use each wavelength as identifier.
- **keys_to_keep** (*list*) – dictionary keys to store at each surface. example: ['amplitude', 'dx', 'dy]
- **overwrite** (*bool*) – if True, overwrites past output file

Returns Saves a list of dictionaries with the POP simulation outputs to a single hdf5 file as a datacube with group tags (e.g. the wavelengths) to identify each simulation, along with the paos package information.

Return type None

Examples

```
>>> from paos.paos_parseconfig import ParseConfig
>>> from paos.paos_run import run
>>> from paos.paos_saveoutput import save_datacube
>>> from joblib import Parallel, delayed
>>> from tqdm import tqdm
>>> pup_diameter, general, fields, opt_chain = ParseConfig('path/to/conf/file')
>>> wavelengths = [1.95, 3.9]
>>> ret_val_list = Parallel(n_jobs=2)(delayed(run)(pup_diameter, 1.0e-6 * wl,␣
↪general['grid size'],
>>>                general['zoom'], fields['0'], opt_chain) for wl in␣
↪tqdm(wavelengths))
>>> group_tags = list(map(str, wavelengths))
>>> save_datacube(ret_val_list, 'path/to/hdf5/file', group_tags,
>>>                keys_to_keep=['amplitude', 'dx', 'dy'], overwrite=True)
```

paos.paos_zernike module

class Zernike(*N*, *rho*, *phi*, *ordering='ansi'*, *normalize=False*)

Bases: `object`

Generates Zernike polynomials

Parameters

- **N** (*integer*) – Number of polynomials to generate in a sequence following the defined 'ordering'
- **rho** (*array like*) – the radial coordinate normalised to the interval [0, 1]
- **phi** (*array like*) – Azimuthal coordinate in radians. Has same shape as rho.
- **ordering** (*string*) – Can be either ANSI ordering (ordering='ansi', this is the default), or Noll ordering (ordering='noll'), or Fringe ordering (ordering='fringe'), or Standard (Born&Wolf) ordering (ordering='standard')
- **normalize** (*bool*) – Set to True generates ortho-normal polynomials. Set to False generates orthogonal polynomials as described in Laksminarayan & Fleck, Journal of Modern Optics (2011). The radial polynomial is estimated using the Jacobi polynomial expression as in their Equation in Equation 14.

Returns **out** – An instance of Zernike.

Return type masked array

Example

```
>>> import numpy as np
>>> from matplotlib import pyplot as plt
>>> x = np.linspace(-1.0, 1.0, 1024)
>>> xx, yy = np.meshgrid(x, x)
>>> rho = np.sqrt(xx**2 + yy**2)
>>> phi = np.arctan2(yy, xx)
```

(continues on next page)

(continued from previous page)

```
>>> zernike = Zernike(36, rho, phi, ordering='noll', normalize=True)
>>> zer = zernike() # zer contains a list of polynomials, noll-ordered
```

```
>>> # Plot the defocus zernike polynomial
>>> plt.imshow(zer[3])
>>> plt.show()
```

```
>>> # Plot the defocus zernike polynomial
>>> plt.imshow(zernike(3))
>>> plt.show()
```

Note: In the example, the polar angle is counted counter-clockwise positive from the x axis. To have a polar angle that is clockwise positive from the y axis (as in figure 2 of Laksminarayan & Fleck, Journal of Modern Optics (2011)) use

```
>>> phi = 0.5*np.pi - np.arctan2(yy, xx)
```

---

static j2mn(*N*, *ordering*)

Convert index j into azimuthal number, m, and radial number, n for the first N Zernikes

Parameters

- **N** (*integer*) – Number of polynomials (starting from Piston)
- **ordering** (*string*) – can take values 'ansi', 'noll', 'fringe'

Returns **m, n**

Return type array

mn2j(*m*, *n*, *ordering*)

Convert radial and azimuthal numbers, respectively n and m, into index j

cov()

Computes the covariance matrix M defined as

```
>>> M[i, j] = np.mean(Z[i, ...]*Z[j, ...])
```

When a pupil is defined as $\Phi = \sum c[k]Z[k,...]$, the pupil RMS can be calculated as

```
>>> RMS = np.sqrt( np.dot(c, np.dot(M, c)) )
```

This works also on a non-circular pupil, provided that the polynomials are masked over the pupil.

Returns **M** – the covariance matrix

Return type array

paos.paos_wfo module

class WFO(*beam_diameter*, *wl*, *grid_size*, *zoom*)

Bases: object

Physical optics wavefront propagation. Implements the paraxial theory described in Lawrence et al., Applied Optics and Optical Engineering, Volume XI (1992)

All units are meters.

Parameters

- **beam_diameter** (*scalar*) – the input beam diameter. Note that the input beam is always circular, regardless of whatever non-circular apodization the input pupil might apply.
- **wl** (*scalar*) – the wavelength
- **grid_size** (*scalar*) – grid size must be a power of 2
- **zoom** (*scalar*) – linear scaling factor of input beam.

Variables

- **wl** (*scalar*) – the wavelength
- **z** (*scalar*) – current beam position along the z-axis (propagation axis). Initial value is 0
- **w0** (*scalar*) – pilot Gaussian beam waist. Initial value is beam_diameter/2
- **zw0** (*scalar*) – z-coordinate of the Gaussian beam waist. initial value is 0
- **zr** (*scalar*) – Rayleigh distance: $\pi w_0^2/\lambda$
- **rayleigh_factor** (*scalar*) – Scale factor multiplying zr to determine 'I' and 'O' regions. Built in value is 2
- **dx** (*scalar*) – pixel sampling interval along x-axis
- **dy** (*scalar*) – pixel sampling interval along y-axis
- **C** (*scalar*) – curvature of the reference surface at beam position
- **fratio** (*scalar*) – pilot Gaussian beam f-ratio
- **wfo** (*array [gridsize, gridsize], complex128*) – the wavefront complex array
- **amplitude** (*array [gridsize, gridsize], float64*) – the wavefront amplitude array
- **phase** (*array [gridsize, gridsize], float64*) – the wavefront phase array in radians
- **wz** (*scalar*) – the Gaussian beam waist w(z) at current beam position
- **distancetofocus** (*scalar*) – the distance to focus from current beam position
- **extent** (*tuple*) – the physical coordinates of the wavefront bounding box (xmin, xmax, ymin, ymax). Can be used directly in im.set_extent.

Returns **out**

Return type an instance of _wfo

Example

```
>>> import paos
>>> import matplolib.pyplot as plt
>>> fig, (ax0, ax1) = plt.sublots(nrows=1, ncols=1)
>>> wfo = paos.WFO(1.0, 0.5e-6, 1024, 4)
>>> wfo.aperture(0, 0, r=1.0/2, shape='circular')
>>> wfo.make_stop()
>>> ax0.imshow(wfo.amplitude())
>>> wfo.lens(1.0)
>>> wfo.propagate(1.0)
>>> ax1.imshow(wfo.amplitude())
```

property wl

property z

property w0

property zw0

property zr

property rayleigh_factor

property dx

property dy

property C

property fratio

property wfo

property amplitude

property phase

property wz

property distancetofocus

property extent

make_stop()
Make current surface a stop. Stop here just means that the wf at current position is normalised to unit energy.

aperture(*xc*, *yc*, *hx=None*, *hy=None*, *r=None*, *shape='elliptical'*, *tilt=None*, *obscuration=False*)
Apply aperture mask

Parameters

- **xc** (*scalar*) – x-centre of the aperture
- **yc** (*scalar*) – y-centre of the aperture
- **hx** (*scalars*) – semi-axes of shape 'elliptical' aperture, or full dimension of shape 'rectangular' aperture
- **hy** (*scalars*) – semi-axes of shape 'elliptical' aperture, or full dimension of shape 'rectangular' aperture
- **r** (*scalar*) – radius of shape 'circular' aperture

- **shape** (*string*) – defines aperture shape. Can be 'elliptical', 'circular', 'rectangular'
- **tilt** (*scalar*) – tilt angle in degrees. Applies to shapes 'elliptical' and 'rectangular'.
- **obscuration** (*boolean*) – if True, aperture mask is converted into obscuration mask.

insideout(*z=None*)
Check if z position is within the Rayleigh distance

Parameters **z** (*scalar*) – beam coordinate long propagation axis

Returns **out** – 'I' if $|z - z_{w0}| < z_r$ else 'O'

Return type string

lens(*lens_fl*)
Apply wavefront phase from paraxial lens

Parameters **lens_fl** (*scalar*) – Lens focal length. Positive for converging lenses. Negative for diverging lenses.

Note: A paraxial lens imposes a quadratic phase shift.

Magnification(*Mx*, *My*)

ptp(*dz*)
Point-to-point (far field) wavefront propagator

Parameters **dz** (*scalar*) – propagation distance

stw(*dz*)
Spherical-to-waist (near field to far field) wavefront propagator

Parameters **dz** (*scalar*) – propagation distance

wts(*dz*)
Waist-to-spherical (far field to near field) wavefront propagator

Parameters **dz** (*scalar*) – propagation distance

propagate(*dz*)
Wavefront propagator. Selects the appropriate propagation primitive and applies the wf propagation

Parameters **dz** (*scalar*) – propagation distance

zernikes(*index*, *Z*, *ordering*, *normalize*, *radius*, *offset=0.0*, *origin='x'*)
Add a WFE represented by a Zernike expansion

Parameters

- **index** (*array of integers*) – Sequence of zernikes to use. It should be a continuous sequence.
- **Z** (*array of floats*) – The coefficients of the Zernike polynomials in meters
- **ordering** (*string*) – Can be 'ansi', 'noll', 'fringe'
- **normalize** (*bool*) – Polynomials are normalised to RMS=1 if True, or to unity at radius if false
- **radius** (*float*) – the radius of the circular aperture over which the polynomials are calculated
- **offset** (*float*) – angular offset in degrees

- **origin** (*string*) – angles measured counter-clockwise positive from x axis by default (origin='x'). Set origin='y' for angles measured clockwise-positive from the y-axis.

Returns **out** – the WFE

Return type masked array

Subpackages

paos.log package

Submodules

paos.log.logger module

class Logger

Bases: object

Abstract class

Standard logging using logger library. It's an abstract class to be inherited to load its methods for logging. It define the logger name at the initialization, and then provides the logging methods.

set_log_name()
Produces the logger name and store it inside the class. The logger name is the name of the class that inherits this Logger class.

info(*message*, **args*, ***kwargs*)
Produces INFO level log See logging.Logger

warning(*message*, **args*, ***kwargs*)
Produces WARNING level log See logging.Logger

debug(*message*, **args*, ***kwargs*)
Produces DEBUG level log See logging.Logger

trace(*message*, **args*, ***kwargs*)
Produces TRACE level log See logging.Logger

error(*message*, **args*, ***kwargs*)
Produces ERROR level log See logging.Logger

critical(*message*, **args*, ***kwargs*)
Produces CRITICAL level log See logging.Logger

Module contents

trace(*self*, *message*, **args*, ***kws*)
Log TRACE level. Trace level log should be produced anytime a function or a methods is entered and exited.

setLogLevel(*level*, *log_id=0*)
Simple function to set the logger level

Parameters

- **level** (*logging level*) –
- **log_id** (*int*) – this is the index of the handler to edit. The basic handler index is 0. Every added handler is appended to the list. Default is 0.

disableLogging(*log_id=0*)
It disables the logging setting the log level to ERROR.

Parameters **log_id** (*int*) – this is the index of the handler to edit. The basic handler index is 0. Every added handler is appended to the list. Default is 0.

enableLogging(*level=20*, *log_id=0*)
It disables the logging setting the log level to ERROR.

Parameters

- **level** (*logging level*) – Default is logging.INFO.
- **log_id** (*int*) – this is the index of the handler to edit. The basic handler index is 0. Every added handler is appended to the list. Default is 0.

addHandler(*handler*)
It adds a handler to the logging handlers list.

Parameters **handler** (*logging handler*) –

addLogFile(*fname='paos.log'*, *reset=False*, *level=10*)
It adds a log file to the handlers list.

Parameters

- **fname** (*str*) – name for the log file. Default is exosim.log.
- **reset** (*bool*) – it reset the log file if it exists already. Default is False.
- **level** (*logging level*) – Default is logging.INFO.

paos.util package

Submodules

paos.util.material module

class Material(*wl*, *Tambient=- 218.0*, *materials=None*)
Bases: object

Class for handling different optical materials for use in *PAOS*

Parameters

- **Tambient** (*scalar*) – Ambient temperature during operation
- **wl** (*scalar or array*) – wavelength in microns
- **materials** (*dict*) – library of materials for optical use

sellmeier(*par*)
Implements the Sellmeier 1 equation to estimate the glass index of refraction relative to air at the glass reference temperature, $T_{ref} = 20°$, and pressure, $P_{ref} = 1\ atm$.

Parameters **par** (*dict*) – dictionary containing the 'K1', 'L1', 'K2', 'L2', 'K3', 'L3' parameters of the Sellmeier 1 model

Returns **out**

Return type the refractive index

static nT(*n*, *D0*, *delta_T*)

Estimate the change in the glass absolute index of refraction with temperature as $n(\Delta T) = \frac{n^2-1}{2n} D_0 \Delta T + n$

Parameters

- **n** (*scalar or array*) – relative index at the reference temperature of the glass
- **D0** (*scalar*) – model parameter
- **delta_T** (*scalar*) – change in temperature relative to the reference temperature of the glass. It is positive if the temperature is grater than the reference temperature

Returns **out** – the scaled relative index

Return type scalar or array (same shape as n)

nair(*T*, *P=1*)

Estimate the air index of refraction at wavelength λ, temperature T, and relative pressure P.

Parameters

- **T** (*scalar*) – reference temperature in $^\circ K$
- **P** (*scalar*) – reference pressure in atmospheres. Defaults to 1 atm.

nmat(*name*)

Given the name of an optical glass, returns the absolute and scaled relative index of refraction in function of wavelength.

Parameters **name** (*str*) – name of the optical glass

Returns **out** – returns two arrays for the glass index of refraction in function of wavelength: one for the absolute index and the other for the index relative to air.

Return type tuple

plot_relative_index(*material_list=None*, *ncols=2*, *figname=None*)

Given a list of materials for optical use, plots the relative index in function of wavelength, at the reference and operating temperature.

Parameters

- **material_list** (*list*) – a list of materials, e.g. [‘SF11’, ‘ZNSE’]
- **ncols** (*int*) – number of columns for the subplots
- **figname** (*str*) – name of figure to save

Returns displays the plot output or stores it to the indicated plot path

Return type None

Examples

```
>>> from paos.util.material import Material
>>> Material().plot_relative_index(material_list=['Caf2', 'Sf11', 'Sapphire'])
```

Module contents

1.7 License

1.8 Acknowledgements

The development of `PAOS` has been possible thanks to:

- Andrea Bocchieri
- Enzo Pascale

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX

R

S

T

W

Z