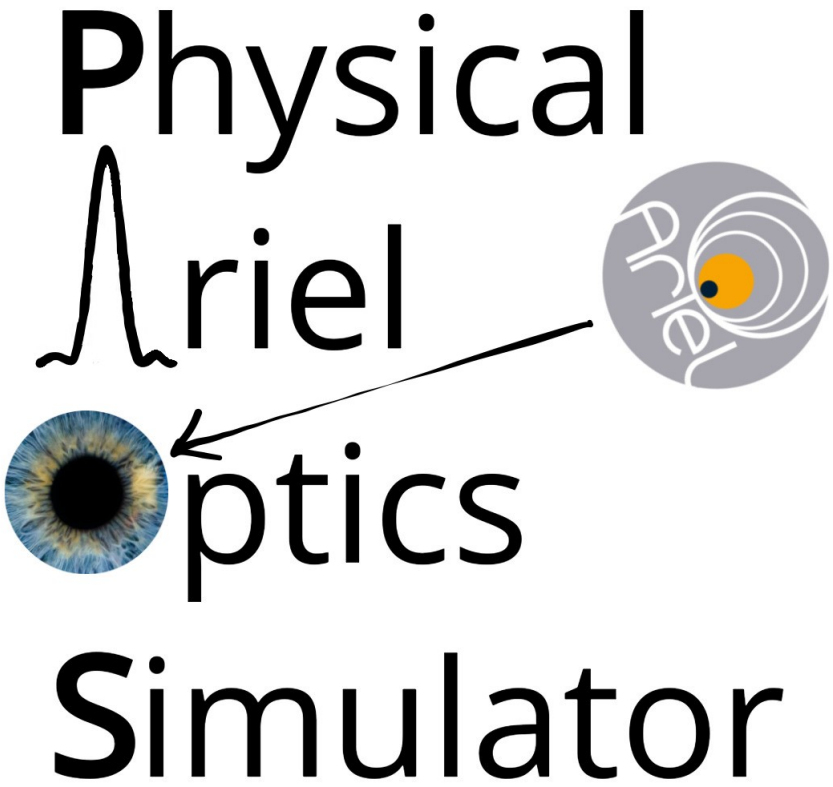
PAOS Manual v0.0.4



Andrea Bocchieri, Enzo Pascale

Created on: December, 2021

Last updated: 6th February, 2022

Table of contents

Python Module Index **97**

Index **99**

List of figures

List of tables

Chapter 1

Introduction

To model the propagation of an electromagnetic field through an optical system, one of two methods is usually adopted:

1. Ray-tracing, i.e. to estimate the path of individual, imaginary lines which represent normals to the wavefront (the surfaces of constant phase);
2. Physical optics propagation (POP), i.e. to estimate the changes in the wavefront as it travels through the optical components

These two methods yield different representations of the beam propagation and are used to different scopes. Ray-tracing is typically used during the design phase as it is fast, flexible and extremely useful to determine the basic properties of an optical system (magnification, aberrations, vignetting, etc.). However, rays propagate along straight lines without interfering with one another and thus are not suitable to predict the effects of diffraction. Conversely, POP models the propagation of wavefronts that coherently interfere with themselves, but it cannot determine aberration changes (which must be input separately).

Several optical propagation codes exist, that implement ray-tracing or POP or combine the two. Widely used examples include commercial ray-tracing programs like Code V [1] and Zemax OpticStudio [1] that also offer POP calculations. However, these programs are costly, require intense training and are not easily customizable to several needs (e.g. Monte Carlo simulations to test wavefront aberration control). Free, publicly available propagation codes, with access to source code are rare. Notable examples include:

- LightPipes
  a set of individual programs (originally in C, now in Python) that can be chained together to model the propagation through an optical system;
- POPPY
  a Python module originally developed as part of a simulation package for the James Webb Space Telescope, that implements an object-oriented system for modeling POP, particularly for telescopic and coronagraphic imaging;
- PROPER
  a library of optical propagation procedures and functions for the IDL (Interactive Data Language), Python, and Matlab environments, intended for exploring diffraction effects in optical systems.

However, these codes are not suitable to every application. For example, POPPY only supports image and pupil planes; intermediate planes are stated as a future goal. PROPER, for its part "assumes an unfolded, linear layout and is not suitable to propagate light through an optical system with optical surfaces that may be tilted or offset from the optical axis". Moreover, none of these codes supports refractive elements.

1.1 About *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 ($\lambda < 3.0$ micron) *Ariel* still delivers high quality data for scientific analysis.

Having a generic input system which mimics the Zemax OpticStudio [r] interface, *PAOS* allows any user expert in CAD modeling to simulate other optical systems, as well (see later in *Input system*).

PAOS simulates the complex wavefront along the propagation axis, and the Point Spread Function (PSF) at the intermediate and focal planes. To do so, *PAOS* implements the Paraxial theory described in Lawrence et al., Applied Optics and Optical Engineering, Volume XI (1992) (see later in *ABCD description*). *PAOS* implements a paraxial ray-tracing (see *Ray tracing*) to estimate the projections of the physical apertures/obscurations, which is used to perform the propagation for an off-axis optical system like the Ariel Telescope without incurring in phase aberrations that are large enough to cause aliasing in the computational grid.

PAOS automizes the choice of algorithm to propagate the wavefront in near-field and far-field condition by using the properties of the pilot beam, an analitically-traced on-axis Gaussian beam (see *Gaussian beams*). *PAOS* also supports the propagation through refractive media (see later in *Materials description*) and is designed to facilitate Monte Carlo simulations for e.g. performance estimation for an ensemble of wavefront error realizations, compatible with an optical performance requirement (see later in Monte Carlo).

PAOS has been validated using the physical optics propagation library PROPER (John E. Krist, PROPER: an optical propagation library for IDL, SPIE (2007)) (see later in *Validation*).

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 (see later in *Ariel*).

Note: *PAOS* v 0.0.4 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.

1.2 Citing

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

1.3 Changelog

Table 1.1 – Changelog table

Version	Date	Changes
0.0.2	15/09/2021	Setting up new *PAOS* repository
0.0.2.1	20/10/2021	First documented *PAOS* release
0.0.3	23/12/2021	Added support for optical materials
0.0.4	22/01/2022	Changed configuration file to .ini

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.

Chapter 2

Installation

The following notes guide you toward the installation of PAOS.

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
```

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
```

```
code version 0.0.4
```

Chapter 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. **Monte Carlo simulations** How to perform Monte Carlo simulations using *PAOS*
8. ***Plotting results*** How to plot POP results
9. ***Saving results*** How to save POP results
10. ***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.

3.1 Quick start

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

3.1.1 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 in Table 3.1.

Table 3.1 – Main command line flags

flag	description
-h, –help	show this help message and exit
-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 *.ini* 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.

Note: *PAOS* implements the *log* submodule which makes use of the python standard module logging for output information. Top-level details of the calculation are output at level logging.INFO, while details of the propagation through each optical plane and debugging messages are printed at level logging.DEBUG. The latter can be accessed by setting the flag *-d*, as explained above. Set the flag *-l* to redirect the logger output to a .log textfile.

Other option flags may be given to run specific simulations, as detailed in Table 3.2.

Table 3.2 – Other option flags

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 in Table 3.3.

Table 3.3 – Lighter output flags

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.

3.1.2 The output file

PAOS stores its main output product to a HDF5 file (extension is *.h5* or *.hdf5*) such as that shown in Fig. 3.1. 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*.

3.1.3 The default plot

An important part of understanding the *PAOS* output is to look at the default plot, as in Fig. 3.2, which shows the PSF, i.e. the squared amplitude of the complex wavefront, at the *AIRS-CH0* focal plane.

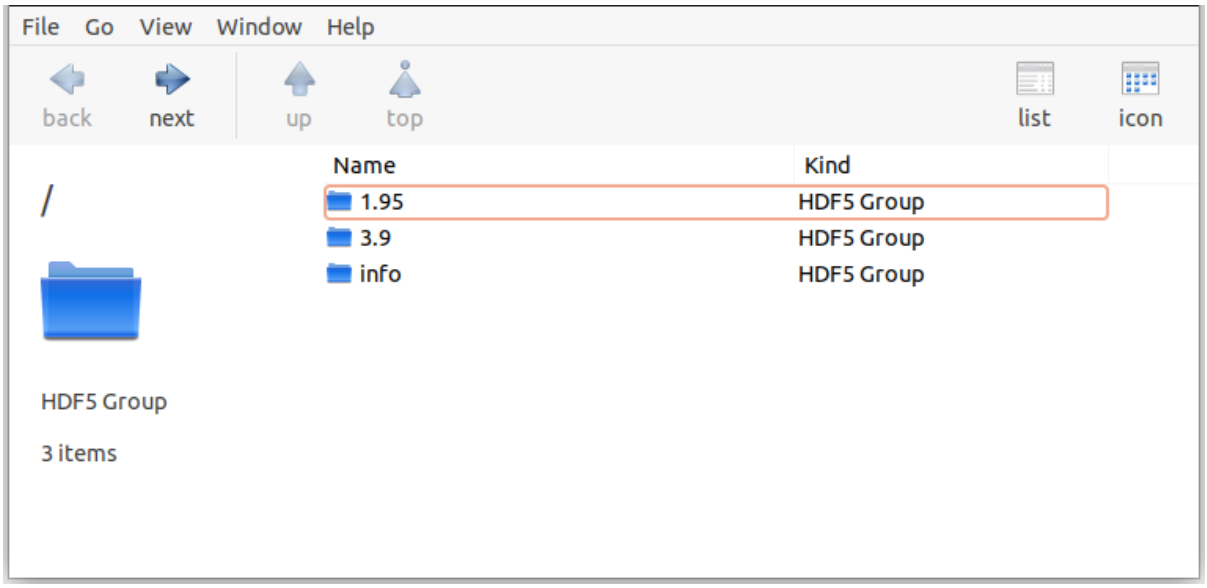


Fig. 3.1 – *Main PAOS output file*

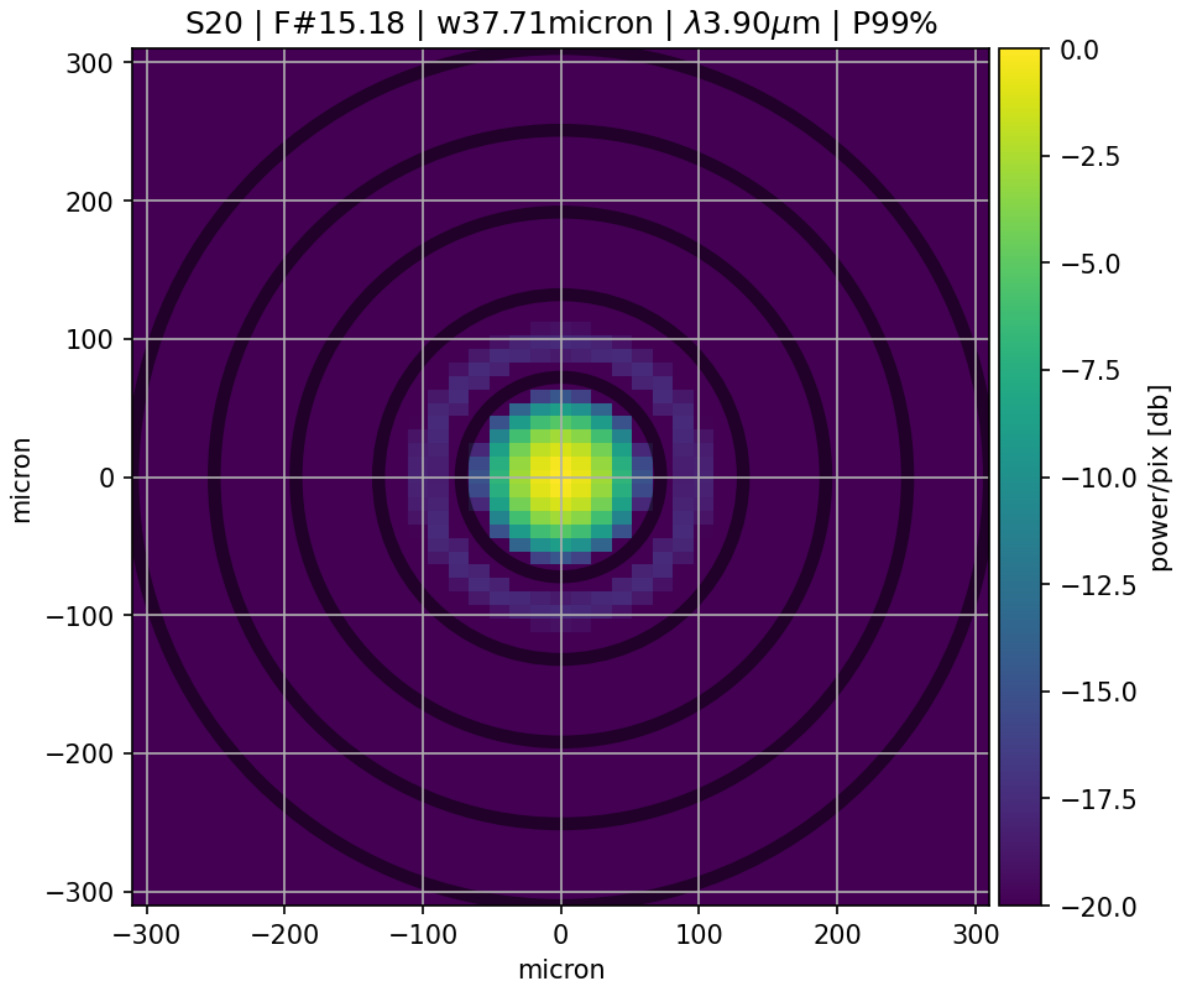


Fig. 3.2 – *Default PAOS plot*

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.

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.

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

3.2 Input system

PAOS has a generic input system to be used by anyone expert in Computer Aided Design (CAD).

Its two pillars are

1. **The *Configuration file*** A .ini configuration file with structure similar to that of Zemax OpticStudio [I];
2. **The *GUI editor*** A GUI to dynamically modify the configuration file and launch instant POP simulations

This structure allows the user to write configuration files from scratch or edit existing ones in a dynamic way, and to launch automized POP simulations that reflect the edits without requiring advanced programming skills.

From a broad perspective, this input system has two advantages:

1. **It can be used to design and test any optical system with relative ease.** Outside the Ariel Consortium, *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.

   ---

2. **It helped in validating the *PAOS* code against existing simulators.**

   Tip: The interested reader may refer to the section *Validation* to see how we validated *PAOS* using the Hubble optical system

   ---

3.2.1 Configuration file

The configuration file is an .ini file structured into four different sections:

1. **DEFAULT** Optional section, not used
2. *General*
3. *Wavelengths*
4. *Fields*
5. *Lens_xx*

Note: *PAOS* defines units as follows:

1. Lens units: meters
2. Angles units: degrees
3. Wavelength units: micron

3.2.1.1 General

Section describing the general simulation parameters and *PAOS* units

Table 3.4 – [general]

keyword	type	description
project	string	A string defining the project name
version	string	Project version (e.g. 1.0)
grid_size	int	Grid size for simulation Must be in [64, 128, 512, 1024]
zoom	int	Zoom size Must be in [1, 2, 4, 8, 16]
lens_unit	string	Unit of lenses Must be 'm'
tambient	float	Ambient temperature in Celsius
pambient	float	Ambient pressure in atmospheres

Below we report a snapshot of this section from the Ariel AIRS CH1 configuration file

```
[general]
  project=Ariel AIRS CH1
  Comment=ARIEL-CEA-PL-ML-002_v3.2
  version=1.0
  grid_size=512
  zoom=4
  lens_unit=m
  # Tambient in C
  Tambient=-223.0
  # Pambient in atm
  Pambient=0.0
```

Fig. 3.3 – *General*

3.2.1.2 Wavelengths

Section listing the wavelengths to simulate (preferably in increasing order)

Table 3.5 – [wavelengths]

keyword	type	description
w1	float	First wavelength
w2	float	Second wavelength
...	...	...

Below we report a snapshot of this section from the Ariel AIRS CH1 configuration file

```
[wavelengths]
  w1=3.90
  w2=5.85
  w3=7.80
```

Fig. 3.4 – *Wavelengths*

3.2.1.3 Fields

Section listing the input fields to simulate

Table 3.6 – [fields]

keyword	type	description
f1	float, float	Field 1: sagittal (x) and tangential (y) angle
f2	float, float	Field 2: sagittal (x) and tangential (y) angle
...	...	...

Below we report a snapshot of this section from the Ariel AIRS CH1 configuration file

```
[fields]
  f1: 0.0,0.0
  f2: 0.0,0.00694
```

Fig. 3.5 – *Fields*

3.2.1.4 Lens_xx

Lens data sections describing how to define the different optical surfaces (INIT, Coordinate Break, Standard, Paraxial Lens, ABCD and Zernike) and their required parameters.

Table 3.7 – [lens_xx]

Surface-Type	Comment	Radius	Thickness	Material	Save	Ignore	Stop	aperture	Par1..N
INIT	string, this surface name	None	None	None	None	None	None	list	None
Coordinate Break	...	None	float	None	Bool	Bool	Bool	list	None

continues on next page

Table 3.7 – continued from previous page

Surface-Type	Comment	Radius	Thickness	Material	Save	Ignore	Stop	aperture	Par1..N
Standard	...	float	float	MIRROR, others	Bool	Bool	Bool	list	None
Paraxial Lens	...	None	float	None	Bool	Bool	Bool	list	Par1 = focal length (float)
ABCD	...	None	float	None	Bool	Bool	Bool	list	Par1..4 = Ax, Bx, Cx, Dx (sagittal) Par5..8 = Ay, By, Cy, Dy (tangential)
Zernike in addition to standard parameters defines: Zindex: polynomial index starting from 0 Z: coefficients in units of wave	...	None	None	None	Bool	Bool	Bool	None	Par1 = wave (in micron) Par2 = ordering, can be standard, ansi, noll, fringe Par3 = Normalisation, can be True or False Par4 = Radius of support aperture of the poly Par5 = origin, can be x (counterclockwise positive from x axis) or y (clockwise positive from y axis)

Note:

1. Set the *Ignore* flag to 1 to skip the surface
2. Set the *Stop* flag to 1 to make the surface a Stop (see *Stops*)
3. Set the *Save* flat to 1 to later save the output for the surface

Note: The *aperture* keyword is a list with the following format:

- aperture = shape type, wx, wy, xc, yc
- shape: either 'elliptical' or 'rectangular'
- type: either 'aperture' or 'obscuration'
- wx, wy: semi-axis of elliptical shapes, or full length of rectangular shape sides
- xc, yc: coordinates of aperture centre

Example: aperture = elliptical aperture, 0.5, 0.3, 0.0, 0.0

Below we report a snapshot of the first lens data section from the Ariel AIRS CH1 configuration file

```
[lens_01]
  SurfaceType=INIT
  Comment=input beam init
  Radius=
  Thickness=
  Material=
  Par1=
  Par2=
  Par3=
  Par4=
  Save=False
  Ignore=False
  aperture=elliptical  aperture, 0.55,0.55,0.0,0.0
```

Fig. 3.6 – *Lens_xx*

3.2.1.5 Parse configuration file

PAOS implements the method *parse_config* that parses the input file, prepares the simulation run and returns the simulation parameters and the optical chain. This method can be called as in the example below.

Example Code example to parse a *PAOS* configuration file.

```
from paos.paos_parseconfig import parse_config
pup_diameter, parameters, wavelengths, fields, opt_chains = parse_config('path/to/ini/
↪file')
```

3.2.2 GUI editor

PAOS implements a GUI editor that allows to dynamically edit and modify the configuration file and to launch POP simulations. This makes it effectively the *PAOS* front-end.

The quickest way to run the GUI is from terminal.

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

```
$ python runGUI.py --help
```

Table 3.8 – GUI command line flags

flag	description
-h, –help	show this help message and exit
-c, –configuration	Input configuration file to pass
-o, –output	Output file path
-d, –debug	Debug mode screen
-l, –log	Store the log output on file

Where the configuration file shall be an *.ini* file (see *Configuration file*). If no configuration file is passed it defaults to the configuration template *template.ini* file. To activate *-d* and *-l* no argument is needed.

The GUI editor then opens and displays a GUI window with a standard Menu (*Open*, *Save*, *Save As*, *Global Settings*, *Exit*) and a series of Tabs:

1. *General Tab*
2. *Fields Tab*
3. ***Lens data Tab*** *Zernike Tab*
4. *Launcher Tab*
5. *Monte Carlo Tab*
6. *Info Tab*

On the bottom of the GUI window, there are five Buttons to perform several actions:

- Submit:
  Submits all values from the GUI window in a flat dictionary
- Show Dict:
  Shows the GUI window values in a nested dictionary, organized into the same sections as the configuration file
- Copy to clipboard:
  Copied the nested dictionary to the local keyboard
- Save:
  Saves the GUI window to the configuration file upon exiting
- Exit:
  Exits the GUI window

The GUI window defines also a right-click Menu with the following options:

- Nothing:
  Does nothing
- Version:
  Displays the current Python, tkinter and PySimpleGUI versions
- Exit:
  Exits the GUI window

3.2.2.1 General Tab

This Tab opens upon starting the GUI. Its purpose is to setup the main simulation parameters.

It contains two Frames:

- **the *General Setup*** Displays the general simulation parameters and *PAOS* units, as defined in *General*. The contents can be altered as necessary, safe if the the cells are disabled.
- **the *Wavelength Setup*** Lists the wavelengths to simulate. This list can be altered by editing the wavelengths, adding more wavelengths, pasting a list of wavelengths from the local clipboard (*comma*-separated or \n-separated) and can also be sorted to increasing order.

Below we report a snapshot of this Tab.

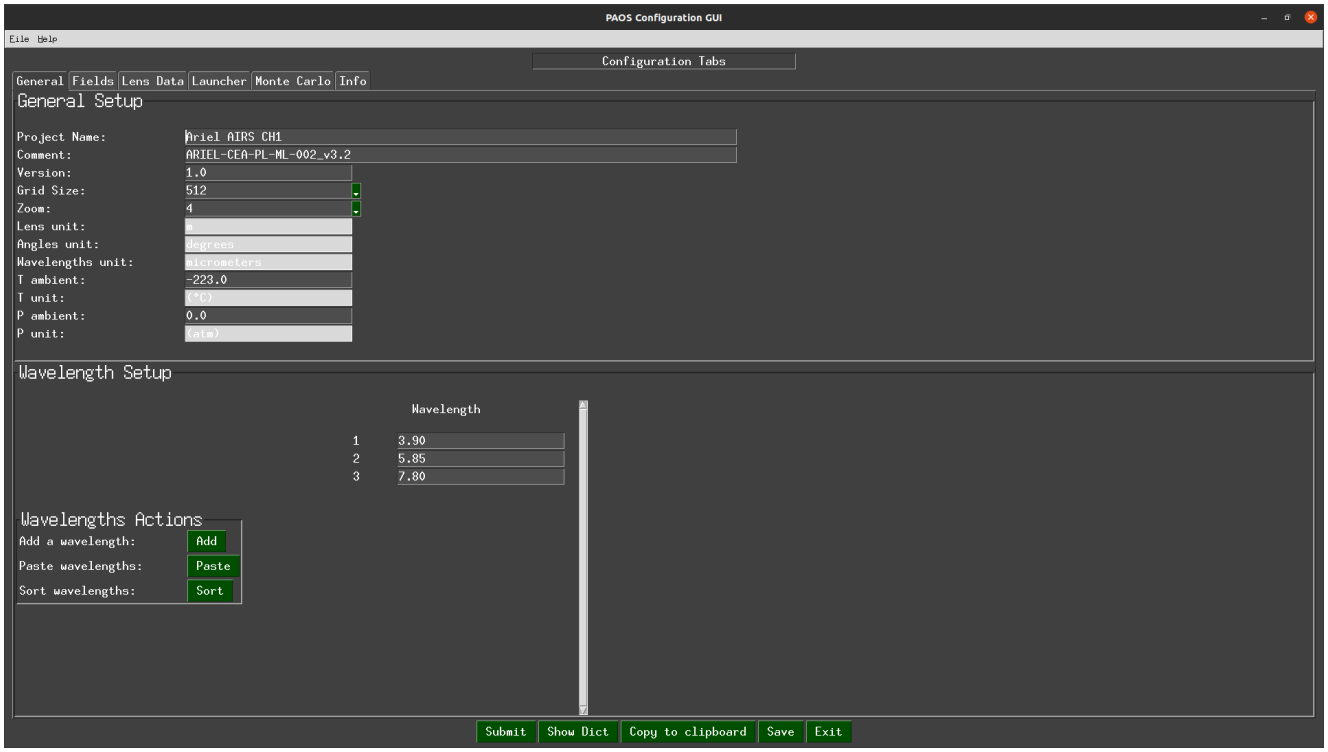


Fig. 3.7 – *General Tab*

3.2.2.2 Fields Tab

This GUI Tab describes the input fields to simulate.

It lists the input fields, as defined in *Fields*. The fields contents can be edited as necessary and new fields can be added.

Note: While more than one field can be listed in this Tab, the current version of *PAOS* only supports simulating one field at a time

---

Below we report a snapshot of this Tab.

3.2.2.3 Lens data Tab

This GUI Tab contains a Table that lists the optical surfaces describing the optical chain to simulate, as defined in *Lens_xx*. The structure is 1:1 with that of Zemax OpticStudio [r] and the columns are the same as in *[lens_xx]*. The contents of each row can be edited as necessary and new surfaces can be added. The Table has horizontal and vertical scrollbars to allow any movement.

For each row, columns are automatically enabled/disabled according to the surface type.

Tip: The column headers for Par1..N change according to the cursor position in the Table.

Tip: It is possible to move the cursor with arrow keys.

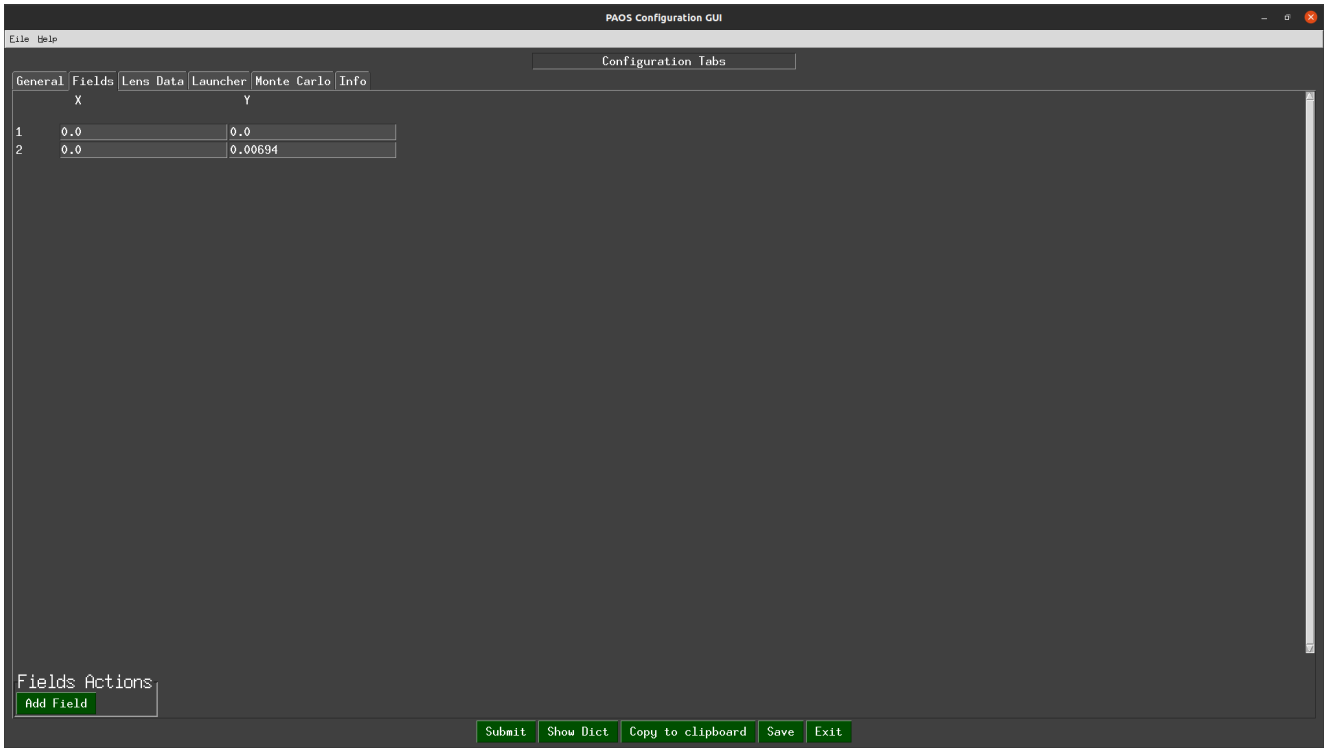


Fig. 3.8 – *Fields Tab*

Tip: To see/edit the contents of the *aperture* column, click on the Button with the yellow triangle.

Below we report a snapshot of this Tab.

Zernike Tab

3.2.2.4 Launcher Tab

3.2.2.5 Monte Carlo Tab

3.2.2.6 Info Tab

3.3 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 class *ABCD*.

3.3.1 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.

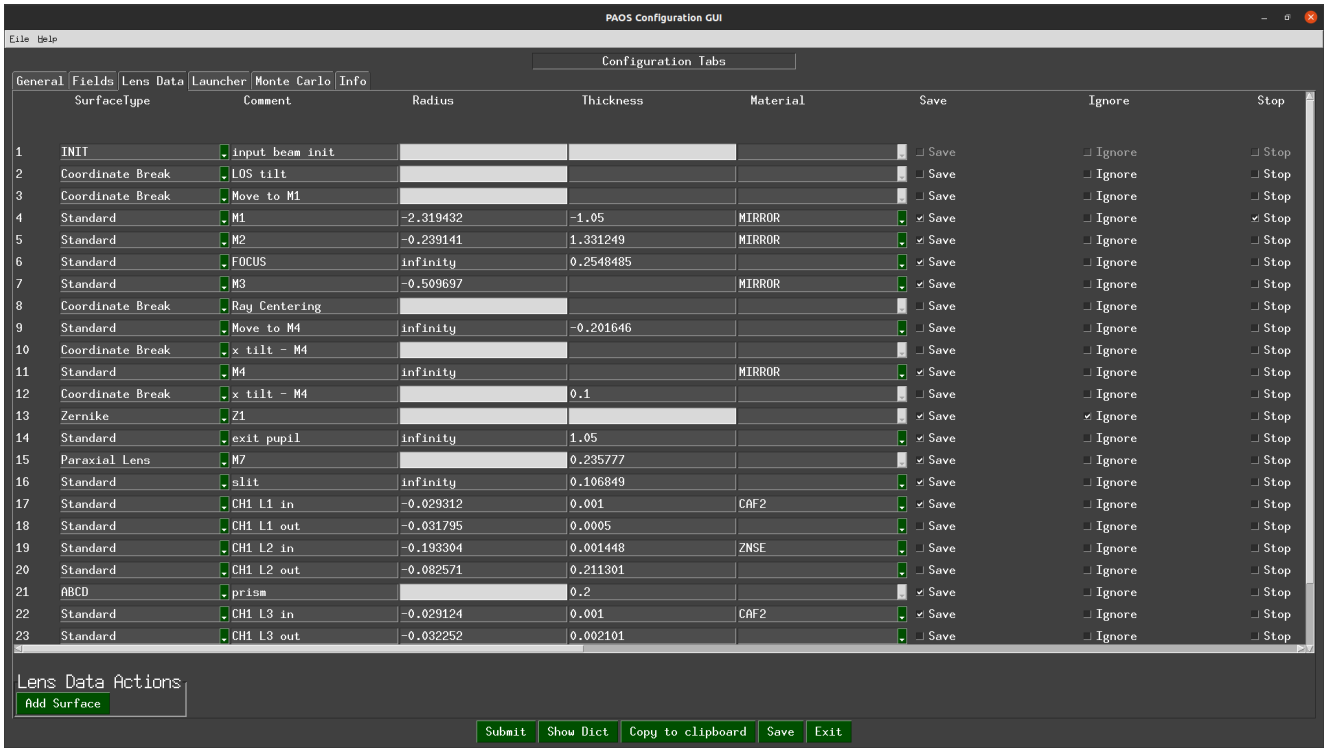


Fig. 3.9 – *Lens data Tab*

3.3.2 Optical coordinates

The *PAOS* code implementation assumes optical coordinates as defined in Fig. 3.15.

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

$$-\frac{1}{z_1} + \frac{1}{z_2} = \frac{1}{f} \tag{3.1}$$

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.

3.3.3 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*).

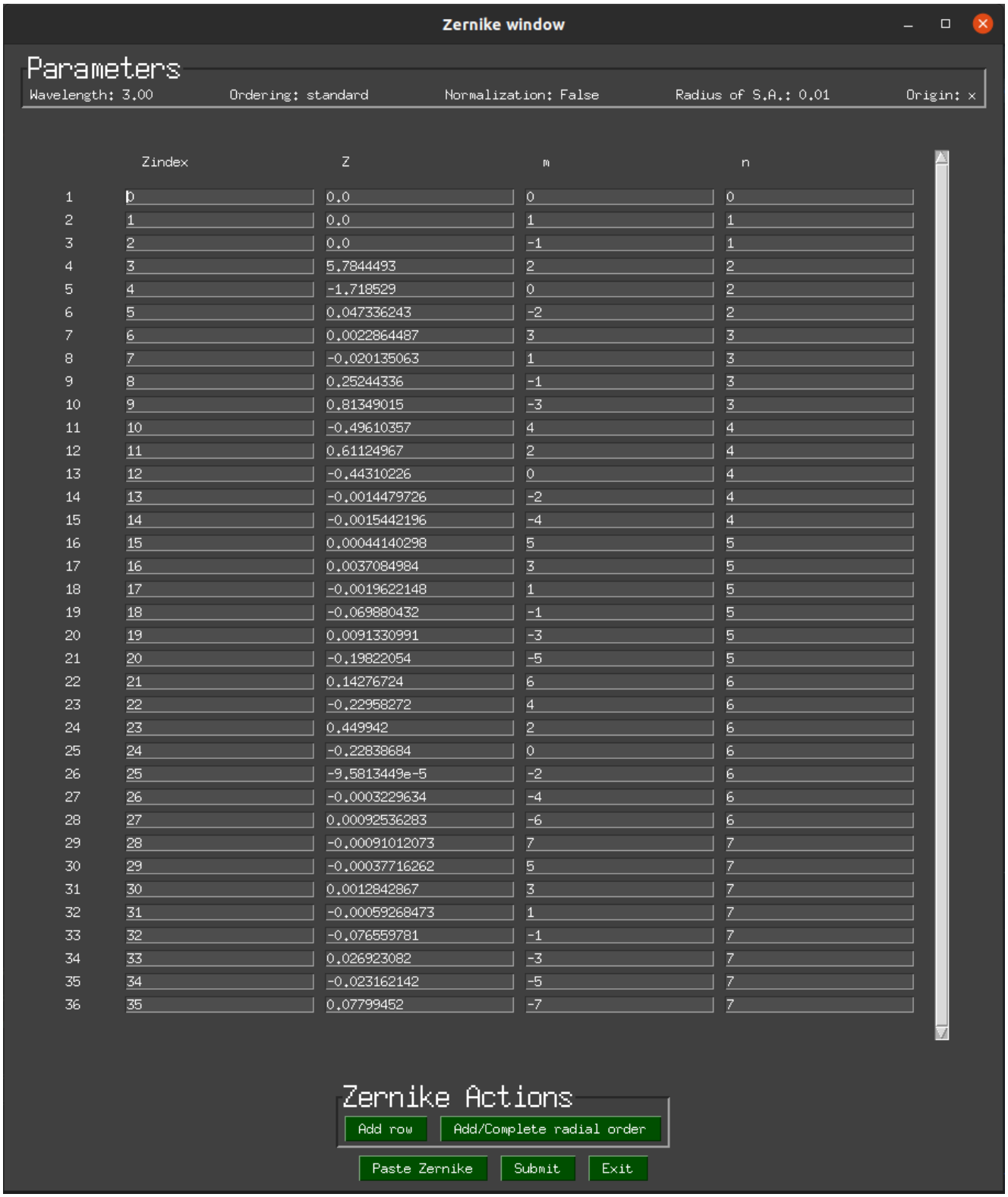


Fig. 3.10 – *Zernike Tab*

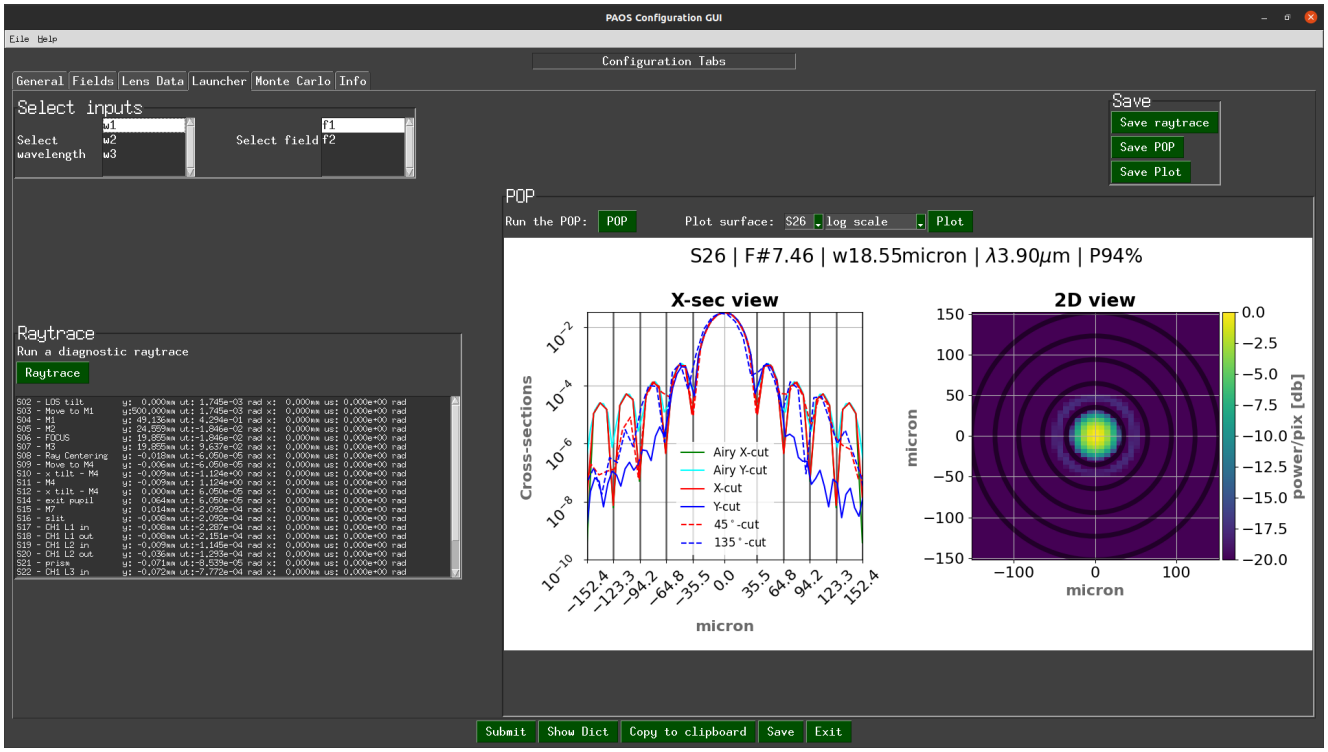


Fig. 3.11 – *Launcher Tab*

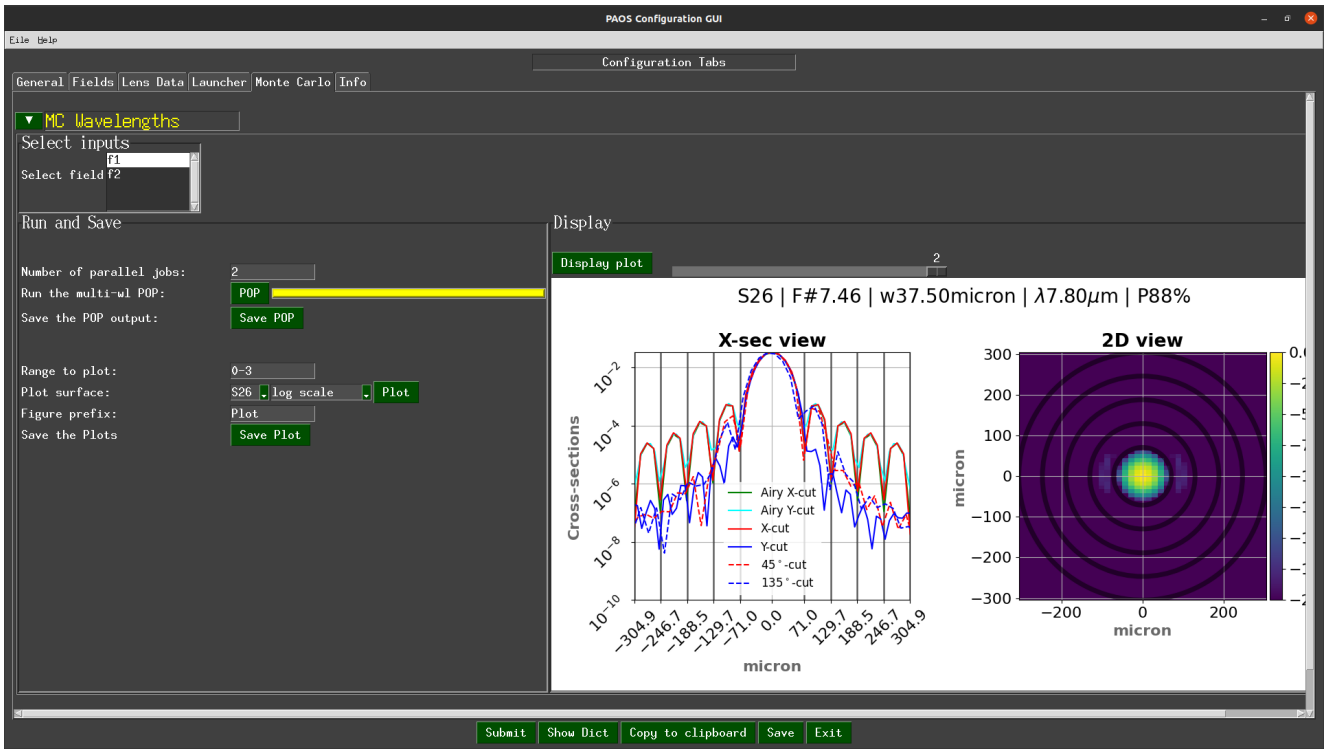


Fig. 3.12 – *Monte Carlo Tab (1)*

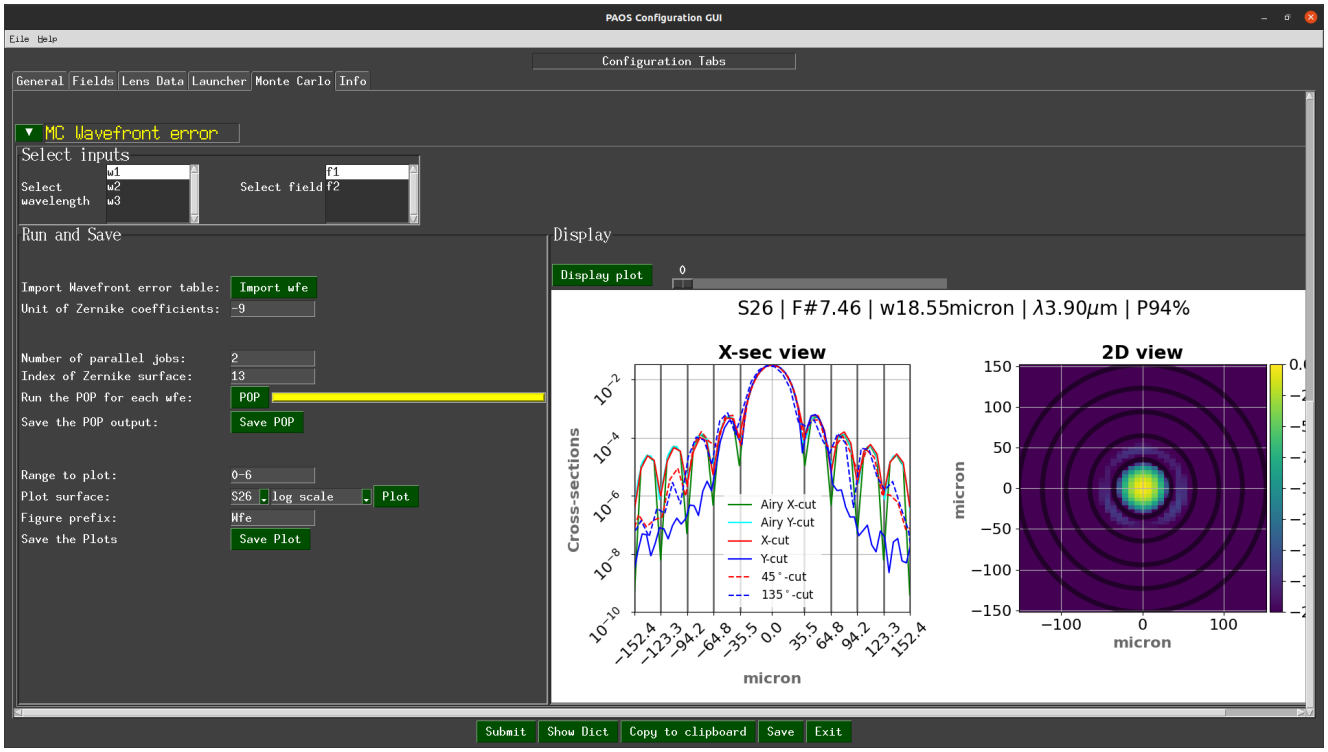


Fig. 3.13 – *Monte Carlo Tab (2)*

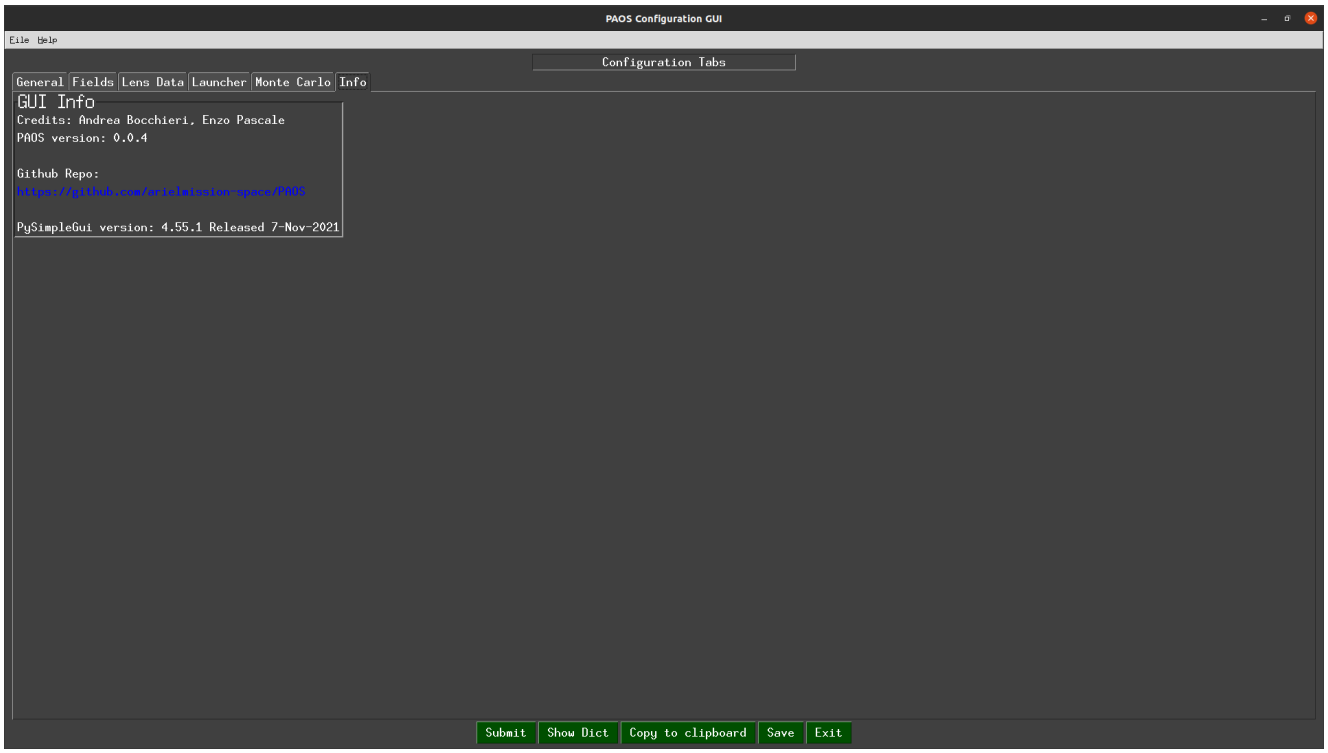


Fig. 3.14 – *Info Tab*

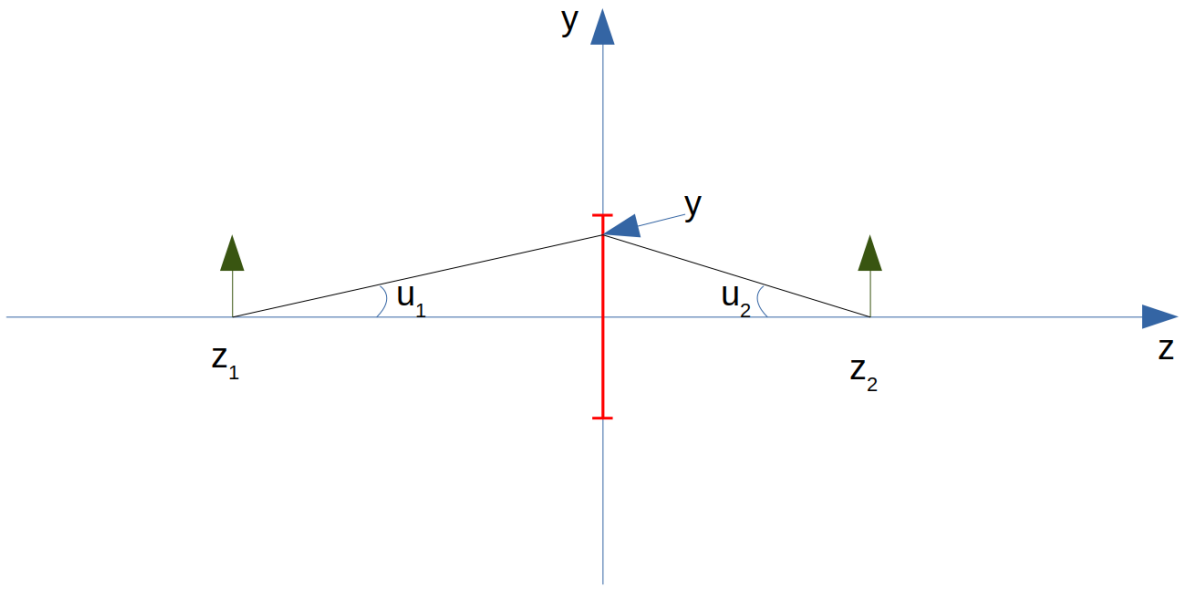


Fig. 3.15 – *Optical coordinates definition*

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 Paraxial 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.

Several Python codes exist that can implement a fully fledged ray-tracing. In a next *PAOS* version we will add support for using one of such codes as an external library to be able to get the expected map of aberrations produced by the realistic elements of the *Ariel* optical chain (e.g. mirrors)

3.3.3.1 Example

Code 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=opt_chains[0])
```

```
['S02 - LOS tilt         y:  0.000mm ut: 1.745e-03 rad x:  0.000mm us: 0.000e+00 rad',
 'S03 - Move to M1       y:500.000mm ut: 1.745e-03 rad x:  0.000mm us: 0.000e+00 rad',
 'S04 - M1               y: 49.136mm ut: 4.294e-01 rad x:  0.000mm us: 0.000e+00 rad',
 'S05 - M2               y: 24.559mm ut:-1.846e-02 rad x:  0.000mm us: 0.000e+00 rad',
 'S06 - FOCUS            y: 19.855mm ut:-1.846e-02 rad x:  0.000mm us: 0.000e+00 rad',
 'S07 - M3               y: 19.855mm ut: 9.637e-02 rad x:  0.000mm us: 0.000e+00 rad',
 'S08 - Ray Centering    y: -0.018mm ut:-6.050e-05 rad x:  0.000mm us: 0.000e+00 rad',
 'S09 - Move to M4       y: -0.006mm ut:-6.050e-05 rad x:  0.000mm us: 0.000e+00 rad',
 'S10 - x tilt - M4      y: -0.009mm ut:-1.124e+00 rad x:  0.000mm us: 0.000e+00 rad',
 'S11 - M4               y: -0.009mm ut: 1.124e+00 rad x:  0.000mm us: 0.000e+00 rad',
```

(continues on next page)

(continued from previous page)

```
 'S12 - x tilt - M4     y:  0.000mm ut: 6.050e-05 rad x:  0.000mm us: 0.000e+00 rad',
 'S13 - exit pupil      y:  0.000mm ut: 6.050e-05 rad x:  0.000mm us: 0.000e+00 rad',
 'S14 - Z1              y:  0.000mm ut: 6.050e-05 rad x:  0.000mm us: 0.000e+00 rad',
 'S15 - L1              y:  0.015mm ut: 6.022e-05 rad x:  0.000mm us: 0.000e+00 rad',
 'S16 - IMAGE_PLANE     y:  0.015mm ut: 6.022e-05 rad x:  0.000mm us: 0.000e+00 rad']
```

3.3.4 Propagation

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

$$\begin{pmatrix} y_2 \\ u_2 \end{pmatrix} = \begin{pmatrix} 1 & t \\ 0 & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ u_1 \end{pmatrix} = \hat{T} \begin{pmatrix} y_1 \\ u_1 \end{pmatrix} \tag{3.2}$$

3.3.4.1 Example

Code example to initialize *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)
print(abcd.ABCD)
```

```
[[ 1. 50.]
 [ 0.  1.]]
```

3.3.5 Thin lenses

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

$$\begin{pmatrix} y_2 \\ u_2 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ -\Phi & 1 \end{pmatrix} \begin{pmatrix} y_1 \\ u_1 \end{pmatrix} = \hat{L} \begin{pmatrix} y_1 \\ u_1 \end{pmatrix} \tag{3.3}$$

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

3.3.5.1 Example

Code example to initialize *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)
print(abcd.ABCD)
```

```
[[ 1.    0.  ]
 [-0.05  1.  ]]
```

3.3.6 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

$$\begin{pmatrix} y_2 \\ u_2 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ -\frac{\Phi}{n_2} & \frac{n_1}{n_2} \end{pmatrix} \begin{pmatrix} y_1 \\ u_1 \end{pmatrix} = \hat{D} \begin{pmatrix} y_1 \\ u_1 \end{pmatrix} \tag{3.4}$$

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.

3.3.6.1 Example

Code example to initialize *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 $n_2 = 1.25$ on a light ray.

```
from paos.paos_abcd import ABCD
n1, n2 = 1.0, 1.25
radius = 20.0  # mm
abcd = ABCD(curvature = 1.0/radius, n1 = n1, n2 = n2)
print(abcd.ABCD)
```

```
[[ 1.    0.  ]
 [-0.01  0.8 ]]
```

3.3.7 Medium change

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

$$\begin{pmatrix} y_2 \\ u_2 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & \frac{n_1}{n_2} \end{pmatrix} \begin{pmatrix} y_1 \\ u_1 \end{pmatrix} = \hat{N} \begin{pmatrix} y_1 \\ u_1 \end{pmatrix} \tag{3.5}$$

3.3.7.1 Example

Code example to initialize *ABCD* to simulate the effect of a change of medium from $n_1 = 1.0$ to $n_2 = 1.25$ on a light ray.

```
from paos.paos_abcd import ABCD
n1, n2 = 1.0, 1.25
abcd = ABCD(n1 = n1, n2 = n2)
print(abcd.ABCD)
```

```
[[1.  0. ]
 [0.  0.8]]
```

3.3.8 Thick lenses

A real (thick) lens is modelled as

$$\begin{pmatrix} y_2 \\ u_2 \end{pmatrix} = \hat{D}_b \hat{T} \hat{D}_a \begin{pmatrix} y_1 \\ u_1 \end{pmatrix} \tag{3.6}$$

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.

3.3.8.1 Example

Code example to initialize *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 $n_l = 1.25$.

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

radius1, radius2 = np.inf, -20.0  # mm
n_os, n_l, n_is = 1.0, 1.25, 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
print(abcd.ABCD)
```

```
[[ 1.      4.    ]
 [-0.0125  0.95  ]]
```

You can now print the thick lens effective focal length as

```
print(abcd.f_eff)
```

```
80.0
```

Notice how in this case the resulting f_{eff} does not depend on t_c.

3.3.9 Magnification

A magnification is modelled as

$$\begin{pmatrix} y_2 \\ u_2 \end{pmatrix} = \begin{pmatrix} M & 0 \\ 0 & 1/M \end{pmatrix} = \hat{M} \begin{pmatrix} y_1 \\ u_1 \end{pmatrix} \tag{3.7}$$

3.3.9.1 Example

Code example to initialize *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)
print(abcd.ABCD)
```

```
[[2.  0. ]
 [0.  0.5]]
```

3.3.10 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:

$$P_t = \begin{pmatrix} \frac{cos(\theta_4)}{cos(\theta_3)} & 0 \\ 0 & \frac{ncos(\theta_3)}{cos(\theta_4)} \end{pmatrix} \begin{pmatrix} 1 & L \\ 0 & 1 \end{pmatrix} \begin{pmatrix} \frac{cos(\theta_2)}{cos(\theta_1)} & 0 \\ 0 & \frac{cos(\theta_1)}{ncos(\theta_2)} \end{pmatrix} \tag{3.8}$$

$$P_s = \begin{pmatrix} 1 & \frac{L}{n} \\ 0 & 1 \end{pmatrix} \tag{3.9}$$

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 Fig. 3.16.

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

$$P_t = \begin{pmatrix} A & B \\ C & D \end{pmatrix} \tag{3.10}$$

where

$$\begin{aligned} A &= \frac{cos(\theta_2)cos(\theta_4)}{cos(\theta_1)cos(\theta_3)} \\ B &= \frac{L}{n}\frac{cos(\theta_1)cos(\theta_4)}{cos(\theta_2)cos(\theta_3)} \\ C &= 0.0 \\ D &= 1.0/A \end{aligned} \tag{3.11}$$

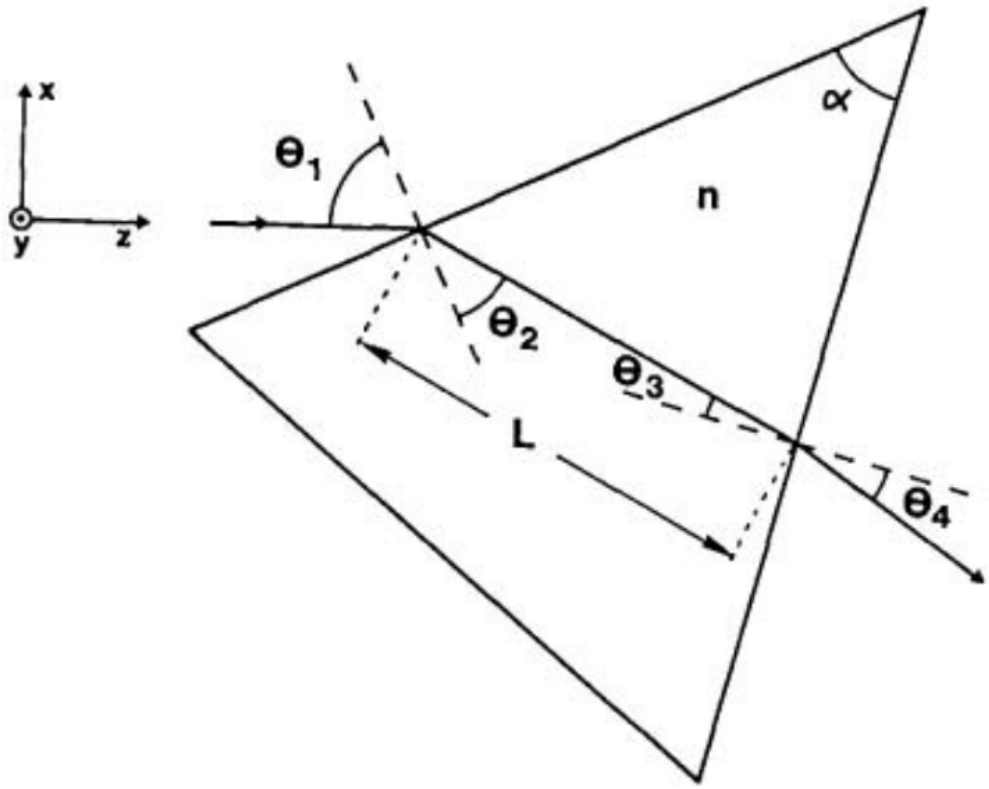


Fig. 3.16 – *Ray propagation through a prism*

3.3.10.1 Example

Code example to initialize *ABCD* to simulate the effect of a prism on a collimated light ray. The prism is $t = 2.0$ mm thick and has a refractive index of $n_p = 1.5$. The prism angles θ_i are selected in accordance with the ray propagation in Fig. 3.16.

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

thickness = 2.0e-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()
```

(continues on next page)

(continued from previous page)

```
abcds.ABCD = np.array([[1, thickness / n], [0, 1]])

print(abcdt.ABCD)
print(abcds.ABCD)
```

```
[[1.59626666 0.35472592]
 [0.         0.62646175]]
[[1.         0.00133333]
 [0.         1.        ]]
```

3.3.11 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:

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} = \begin{pmatrix} 1 & t \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ -\Phi & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & n_1/n_2 \end{pmatrix} \begin{pmatrix} M & 0 \\ 0 & 1/M \end{pmatrix} = \hat{T}\hat{L}\hat{N}\hat{M} \tag{3.12}$$

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:

$$\begin{aligned} M &= \frac{AD - BC}{D} \\ n_1/n_2 &= MD \\ t &= \frac{B}{D} \\ \Phi &= -\frac{C}{M} \end{aligned} \tag{3.13}$$

With these definitions, the effective focal length is

$$f_{eff} = \frac{1}{\Phi M} \tag{3.14}$$

3.3.11.1 Example

Code example to initialize *ABCD* to simulate an optical system equivalent for a magnification $M = 2.0$, a change of medium from $n_1 = 1.0$ to $n_2 = 1.25$, 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, 1.25
thickness = 5.0  # mm
magnification = 2.0

abcd = ABCD(thickness = thickness, curvature = 1.0/radius, n1 = n1, n2 = n2, M =␣
↪magnification)
print(abcd.ABCD)
```

```
[[ 1.9   2.  ]
 [-0.02  0.4 ]]
```

3.4 POP description

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

In *PAOS*, this is handled by the class *WFO*.

3.4.1 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.

3.4.2 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.

3.4.3 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

$$\vec{R_0} = (x - x_{dec}, y - y_{dec}, 0) \tag{3.15}$$

and the unit vector of the light ray

$$\vec{n_0} = (zu_x, zu_y, z) \tag{3.16}$$

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

$$U^T \vec{R_0} + \rho U^T \vec{n_0} = \vec{R_0}' \tag{3.17}$$

that can be solved as

$$\begin{aligned} U^T \vec{n_0} = \vec{n_0}' = z'(u'_x, u'_y, 1) \\ \rho = -\frac{z'_0}{z'} \end{aligned} \tag{3.18}$$

3.4.3.1 Example

Code example to use *coordinate_break* to simulate 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 $x_{rot} = 0.1°$.

```
import numpy as np
from paos.paos_coordinatebreak import coordinate_break

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 = coordinate_break(vt, vs, xdec, ydec, xrot, yrot, zrot, order=0.0)

print(vs, vt)
```

```
[0. 0.] [-0.01000002  0.00174533]
```

3.4.4 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))

$$I(r) = I_0 e^{-\frac{2r^2}{w(z)^2}} = \frac{2P}{\pi w(z)^2} e^{-\frac{2r^2}{w(z)^2}} \tag{3.19}$$

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

$$w(z)^2 = w_0^2 \left[1 + \left(\frac{\lambda z}{\pi w_0^2}\right)^2\right] = w_0^2 \left[1 + \left(\frac{z}{z_R}\right)^2\right] \tag{3.20}$$

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 Fig. 3.17 (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))

$$a(r, 0) = e^{-\frac{r^2}{w_0^2}} e^{-\frac{jkr^2}{R}} \tag{3.21}$$

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

$$a(r, 0) = e^{-\frac{r^2}{w_0^2}} \tag{3.22}$$

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

3.4.4.1 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:

$$z_R = \frac{\pi w_0^2}{\lambda} \tag{3.23}$$

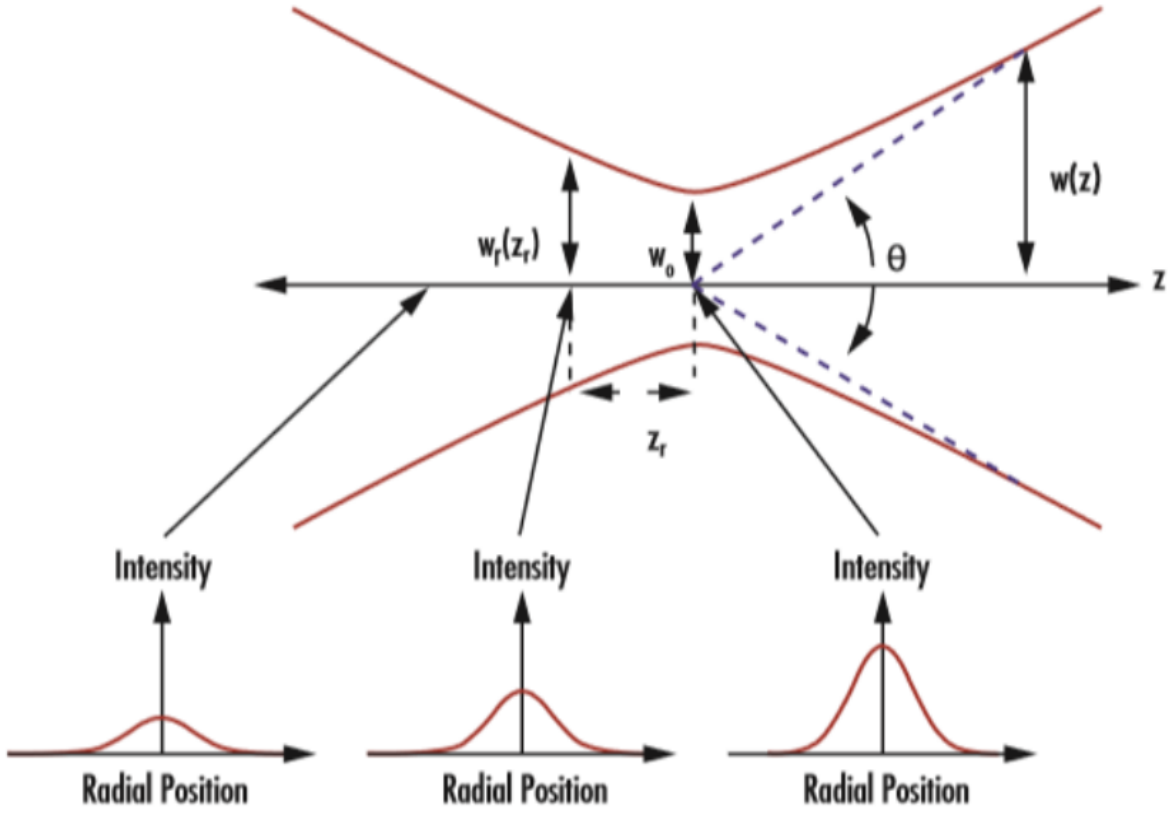


Fig. 3.17 – *Gaussian beam diagram*

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

$$R(z) = z + \frac{z_R^2}{z} \tag{3.24}$$

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*).

3.4.4.2 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))

$$a(r,z) = e^{-j[kz-\theta(z)]} e^{-\frac{r^2}{w(z)^2}} e^{-\frac{jkr^2}{R(z)}} \tag{3.25}$$

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

$$\theta(z) = tan^{-1}\left(\frac{z_R}{z}\right) \tag{3.26}$$

$\theta(z)$ 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:

$$\frac{1}{q(z)} = \frac{1}{R(z)} - \frac{j\lambda}{\pi n w(z)^2} \tag{3.27}$$

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

$$\frac{1}{q_2} = \frac{C + D/q_1}{A + B/q_1} \tag{3.28}$$

3.4.4.3 Gaussian beam magnification

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

$$\begin{aligned} A &= M \\ D &= 1/M \\ B &= C = 0 \end{aligned} \tag{3.29}$$

Therefore, for the complex radius of curvature we have that

$$q_2 = M^2 q_1. \tag{3.30}$$

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.

3.4.4.4 Example

Code example to use *WFO* to simulate a magnification of the beam for the tangential direction $M_t = 3.0$, while keeping the sagittal direction unchanged ($M_s = 1.0$).

```
from paos.paos_wfo import WFO

beam_diameter = 1.0  # m
wavelength = 3.0e-6
```

(continues on next page)

(continued from previous page)

```
grid_size = 512
zoom = 4

wfo = WFO(beam_diameter, wavelength, grid_size, zoom)
Ms, Mt = 1.0, 3.0
wfo.Magnification(Ms, Mt)

print(wfo.wz)
```

```
paos - WARNING - Gaussian beam magnification is implemented, but has not been tested.
```

```
1.5
```

As a result, the semi-diameter of the beam increases three-fold.

3.4.5 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:

$$\begin{aligned} \text{inside} &\leftrightarrow |z - z(w)| \leq z_R \\ \text{outside} &\leftrightarrow |z - z(w)| > z_R \end{aligned} \tag{3.31}$$

The situation is described in Fig. 3.18, taken 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

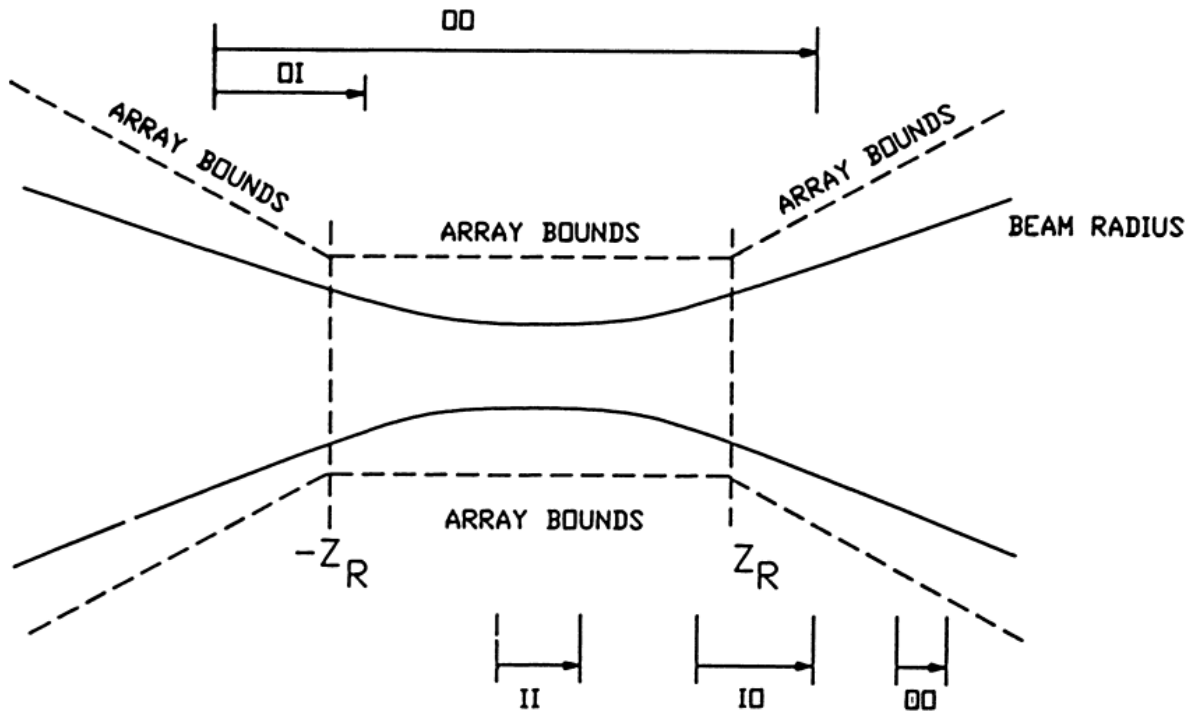


Fig. 3.18 – *Wavefront propagators*

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

3.4.5.1 Example

Code example to use *WFO* to propagate the beam over a thickness of 10.0 mm.

```
from paos.paos_wfo import WFO

wfo = WFO(beam_diameter, wavelength, grid_size, zoom)
print(f'Initial beam position, beam semi diameter: {wfo.z, wfo.wz}')

thickness = 10.0e-3  # m
wfo.propagate(dz = thickness)
print(f'Final beam position, beam semi diameter: ({wfo.z}, {wfo.wz:.6f})')
```

```
Initial beam position, beam semi diameter: (0.0, 0.5)
Final beam position, beam semi diameter: (0.01, 0.500000)
```

The current beam position along the z-axis is now updated.

3.4.6 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 in Fig. 3.19.

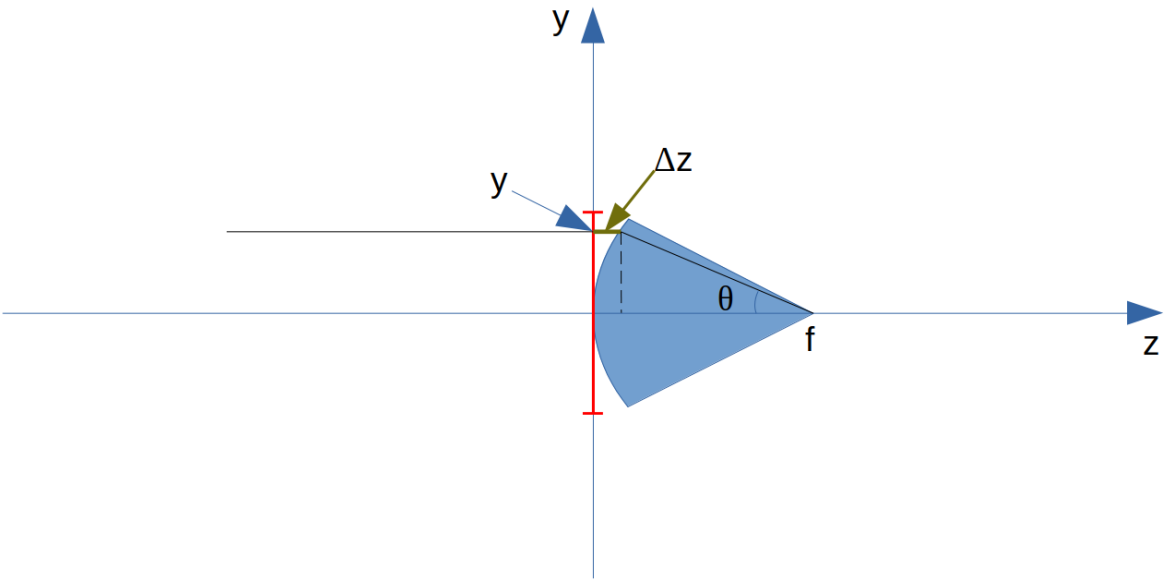


Fig. 3.19 – *Diagram for convergent beam*

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:

$$a_2(x, y, z) = a_1(x, y, z)e^{2\pi j \Delta z/\lambda} \tag{3.32}$$

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

$$\Delta z = f - \sqrt{f^2 - y^2} \simeq \frac{y^2}{2f} \tag{3.33}$$

The phase delay over the whole lens aperture is then

$$\Delta\Phi = -\Delta z/\lambda = -\frac{x^2+y^2}{2f\lambda} \tag{3.34}$$

3.4.6.1 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 in Fig. 3.20.

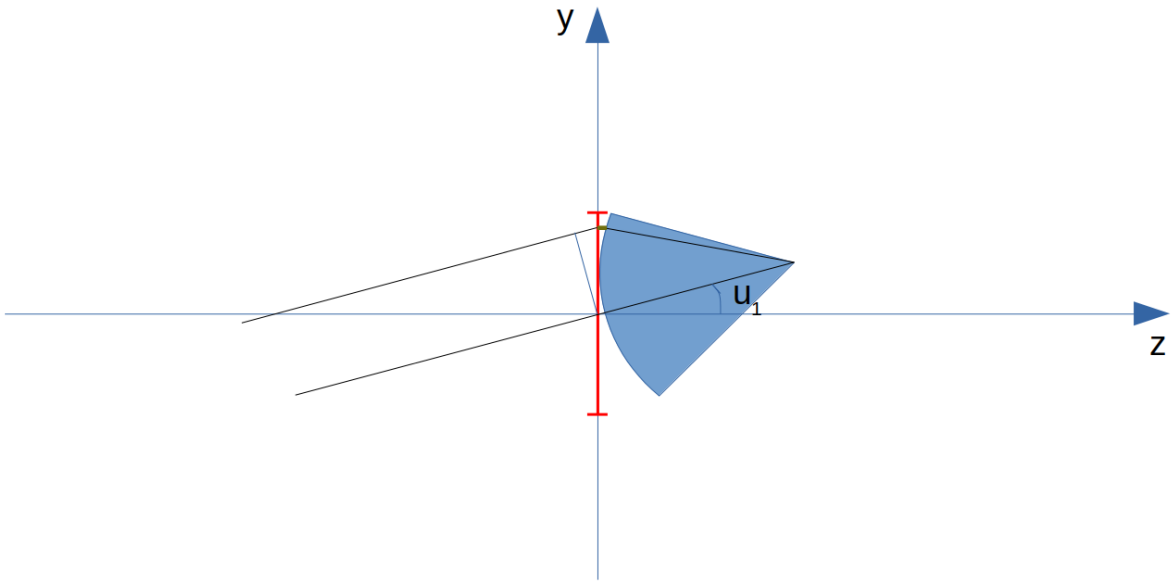


Fig. 3.20 – *Diagram for convergent sloped beam*

The total phase delay is then

$$\Delta\Phi = -\frac{x^2+y^2}{2f\lambda} + \frac{yu_1}{\lambda} = -\frac{x^2+(y-fu_1)^2}{2f\lambda} + \frac{yu_1^2}{2\lambda} = -\frac{x^2+(y-y_0)^2}{2f\lambda} + \frac{y_0^2}{2f\lambda} \tag{3.35}$$

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.

3.4.6.2 Off-axis incoming beam

The case of off-axis optics is described in Fig. 3.21.

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

$$\Delta\Phi = -\frac{x^2+y^2}{2f\lambda} = -\frac{x^2+(y_c-\delta y)^2}{2f\lambda} = -\frac{x^2+\delta y^2}{2f\lambda} + \frac{\delta y u_2}{\lambda} - \frac{y_c^2}{2f\lambda} \tag{3.36}$$

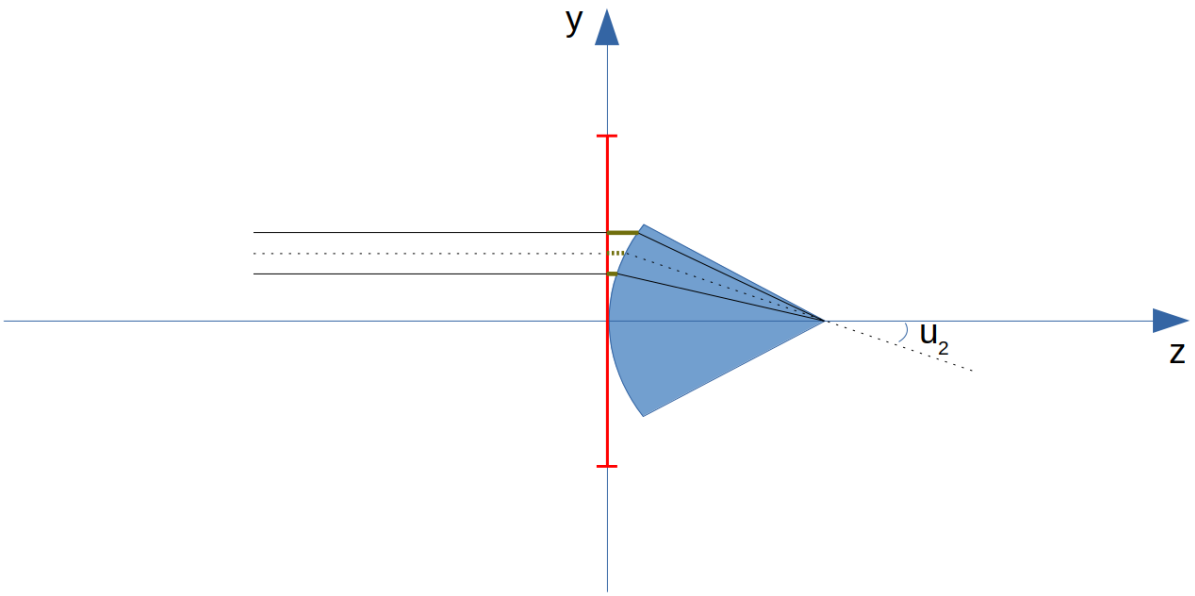


Fig. 3.21 – *Diagram for off-axis beam*

If the incoming beam has a slope u_1, then

$$\Delta\Phi = -\frac{x^2+\delta y^2}{2f\lambda} + \frac{\delta y(u_1+u_2)}{\lambda} - \frac{y_c^2}{2f\lambda} + y_c u_1 \tag{3.37}$$

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.

3.4.6.3 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

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

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. (3.24)
2. check the initial position using Eq. (3.31)
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.

3.4.7 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.

3.4.7.1 Example

Code example to use *WFO* to simulate the beam propagation through an elliptical aperture with semi-major axes $x_{rad} = 0.55$ and $y_{rad} = 0.365$, positioned at $x_{dec} = 0.0$, $y_{dec} = 0.0$.

```
from paos.paos_wfo import WFO

xrad = 0.55  # m
yrad = 0.365
xdec = ydec = 0.0

field = {'us': 0.0, 'ut': 0.1}
vt = np.array([0.0, field['ut']])
vs = np.array([0.0, field['us']])

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

wfo = WFO(beam_diameter, wavelength, grid_size, zoom)

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)

print(aperture)
```

```
Aperture: EllipticalAperture
positions: [256., 256.]
a: 70.4
b: 46.48813752661069
theta: 0.0
```

3.4.8 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.

3.4.8.1 Example

Code example to use *WFO* to simulate an aperture stop.

```
import numpy as np
from paos.paos_wfo import WFO

wfo = WFO(beam_diameter, wavelength, grid_size, zoom)
print(f'Total throughput: {np.sum(wfo.amplitude**2)}')

wfo.make_stop()
print(f'Total throughput: {np.sum(wfo.amplitude**2)}')
```

```
Total throughput: 262144.0
Total throughput: 1.0
```

3.4.9 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:

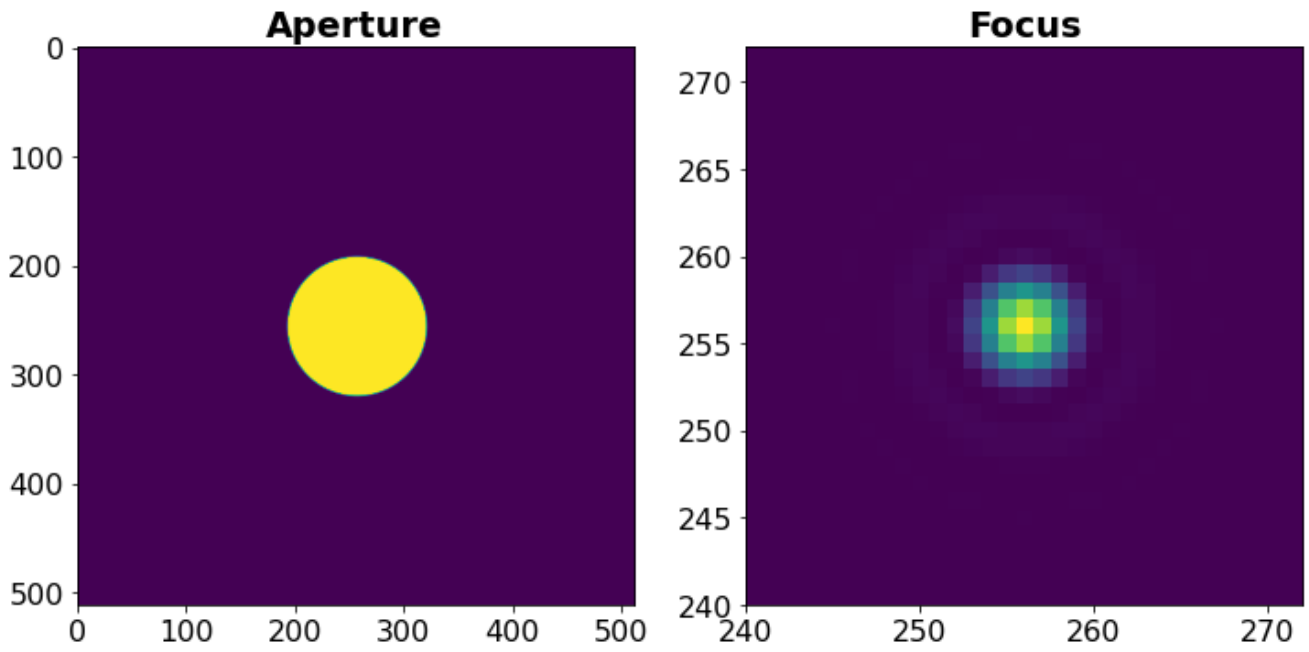
1. Apply coordinate break
2. Apply aperture
3. Apply stop
4. Apply aberration (see *Aberration description*)
5. Apply ABCD matrix and update
6. Apply magnification
7. Apply lens
8. Apply propagation thickness
9. Update ABCD matrix
10. Repeat over all optical elements

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

3.4.9.1 Example

Code example to use *WFO* to simulate a simple propagation loop that involves key actions such as applying a circular aperture, the throughput normalization, applying a Paraxial lens with focal length $f = 1.0$ m, and propagating to the lens focus.

```
import matplotlib.pyplot as plt

fig, (ax0, ax1) = plt.subplots(nrows=1, ncols=2, figsize=(12, 6))
wfo = paos.WFO(beam_diameter, wavelength, grid_size, zoom)

wfo.aperture(xc=xdec, yc=ydec, r=beam_diameter/2, shape='circular')
wfo.make_stop()
ax0.imshow(wfo.amplitude**2)
ax0.set_title('Aperture')

fl = 1.0  # m
thickness = 1.0

wfo.lens(lens_fl=fl)
wfo.propagate(dz=thickness)
ax1.imshow(wfo.amplitude**2)
ax1.set_title('Focus')

zoomin = 16
shapex, shapey = wfo.amplitude.shape
ax1.set_xlim(shapex // 2 - shapex // 2 // zoomin, shapex // 2 + shapex // 2 // zoomin)
ax1.set_ylim(shapey // 2 - shapey // 2 // zoomin, shapey // 2 + shapey // 2 // zoomin)

plt.show()
```



3.5 Aberration description

Brief description of wavefront error (WFE) modelling and how it is implemented in *PAOS*.

In *PAOS*, this is handled by the class `Zernike`.

3.5.1 Introduction

In optics, aberration is a property of optical systems, that causes light to be spread out over some region of space rather than focused to a point. An aberration causes an image-forming optical system to depart from the prediction of paraxial optics (see *Paraxial region*), producing an image which is not sharp. The WFE and the resulting image distortion depend on the type of aberration.

The WFE can be modelled as a superposition of Zernike polynomials that describe *Optical aberrations* and a random Gaussian field that describes *Surface roughness*. That is, the WFE can be decomposed into low frequency and medium-to-high frequency contributors.

Useful concepts to estimate image quality such as

1. *Strehl ratio*
2. *Encircled energy*

are discussed in the following sections for reference, although they are not implemented in the main *PAOS* code.

3.5.1.1 Strehl ratio

For large aberrations, the image size is larger than the Airy disk. From the conservation of energy, the irradiance at the center of the image has to decrease when the image size increases.

A useful definition for image quality is the Strehl ratio (see e.g. Malacara-Hernández, Daniel & Malacaea-Hernandez, Zacarias & Malacara, Zacarias. (2005). Handbook of Optical Design Second Edition.), i.e. the ratio of the irradiance at the center of the aberrated PSF to that of an ideal Airy function, which is

approximated as

$$\text{Strehl ratio} \simeq 1 - k^2\sigma_W^2 \tag{3.38}$$

where k is the wavenumber and σ_W is the wavefront variance, i.e. the square of the rms wavefront deviation. This expression is adequate to estimate the image quality for Strehl ratios as low as 0.5.

3.5.1.2 Encircled energy

Another useful way to estimate how much an optical system deviates from optimal is to compute the radial (or encircled) energy distribution of the PSF.

The encircled energy can be obtained from the spot diagram by counting the number of points in the diagram, inside of circles with different diameters. Alternatively, the fraction of the encircled energy f in function of the encircled energy aperture radius R_f in image-space-normalized units can be computed as

$$f = \int_0^{2\pi} \int_0^{R_f} PSF(r, \phi)\ r\ dr\ d\phi \tag{3.39}$$

For an ideal diffraction limited system $R_{0.838} = 1.22$, and $R_{0.910} = 2.26$ are the energy encircled in the first and second Airy null, respectively. For an optical system that can be described using a single $F_\#$ the normalised radii can be expressed in units of wavelengths using

$$r = R_f F_\# \lambda \tag{3.40}$$

Fig. 3.22 reports the encircled energy in function of R_f for an aberrated PSF and a diffraction limited PSF. The 83.8% and 91.0% levels are marked in red for the aberrated PSF, and in black for the diffraction limited PSF.

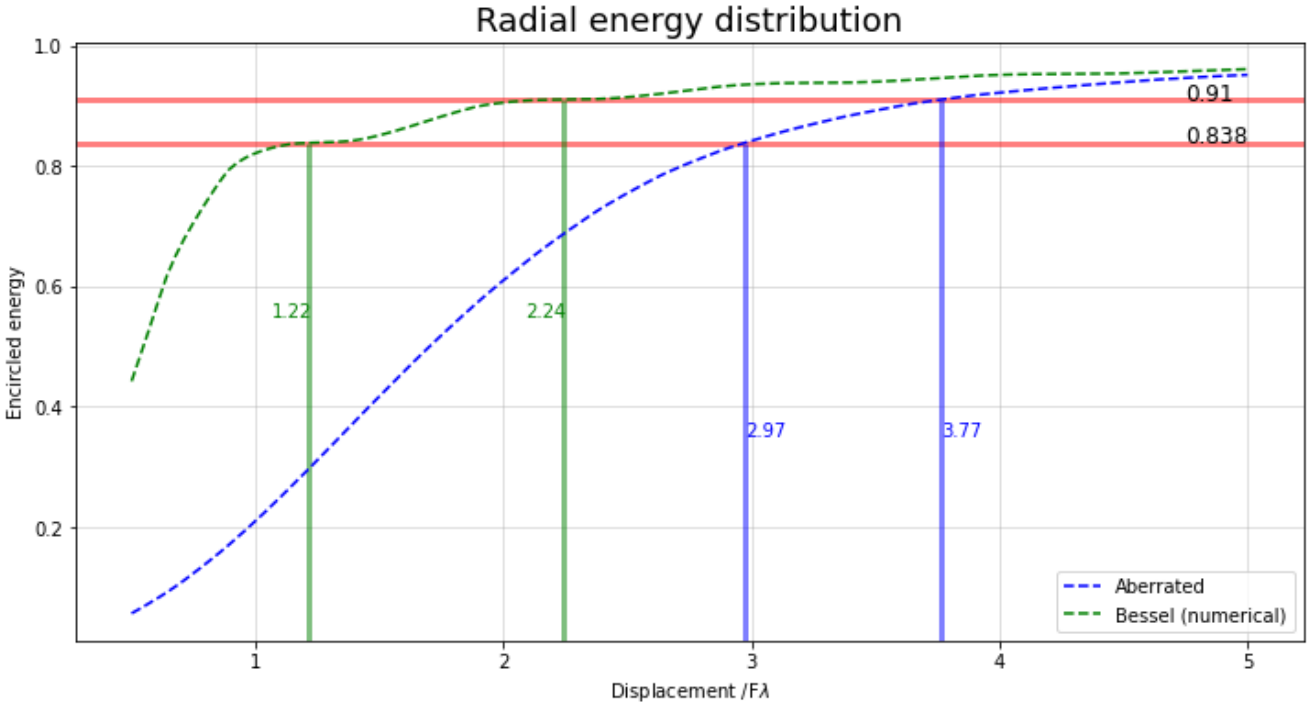


Fig. 3.22 – *Encircled energy*

3.5.2 Optical aberrations

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

Following Laksminarayan & Fleck, Journal of Modern Optics (2011), the function describing an arbitrary wavefront wavefront in polar coordinates W(r, θ) can be expanded in terms of a sequence of Zernike polynomials as

$$W(\rho, \theta) = \sum_{n,m} C_n^m Z_n^m(\rho, \theta) \tag{3.41}$$

where C_n^m and coefficient of the Zernike polynomial $Z_n^m(\rho, \theta)$.

The first three terms in (3.41) describe Piston and Tilt aberrations and can be neglected. Non-normalised Zernike polynomials are defined in *PAOS* as:

$$Z_n^m = \begin{cases} R_n^m(\rho)\ cos(m\phi) & m \geq 0 \\ R_n^{-m}(\rho)\ cos(m\phi) & m < 0 \\ 0 & n - m \text{ is odd} \end{cases} \tag{3.42}$$

where the radial polynomial is normalized such that $R_n^m(\rho = 1) = 1$, or

$$\left\langle [Z_n^m(\rho, \phi)]^2 \right\rangle = 2\frac{n+1}{1+\delta_{m0}} \tag{3.43}$$

with δ_{mn} the Kroneker delta function, and the average operator $\langle\rangle$ is intended over the pupil.

Using polar elliptical coordinates allows *PAOS* to describe pupils that are elliptical in shape as well as circular:

$$\rho^2 = \frac{x_{pup}^2}{a^2} + \frac{y_{pup}^2}{b^2} \tag{3.44}$$

where x_{pup} and y_{pup} are the pupil physical coordinates and a and b are the pupil semi-major and semi-minor axes, respectively.

Fig. 3.23 reports surface plots of the Zernike polynomial sequence up to radial order $n = 10$. The name of the classical aberration associated with some of them is also provided (figure taken from Laksminarayan & Fleck, Journal of Modern Optics (2011)).

PAOS can generate both ortho-normal polynomials and orthogonal polynomials and the ordering can be either ANSI (default), or Noll, or Fringe, or Standard (see e.g. Born and Wolf, Principles of Optics, (1999)).

3.5.2.1 Example of an aberrated pupil

An example of aberrated PSFs at the *Ariel* Telescope exit pupil is shown in Fig. 3.24.

In this figure, the same Surface Form Error (SFE) of 50 nm root mean square (rms) is allocated to different optical aberrations. Starting from the top left panel (oblique Astigmatism), seven such simulations are shown, in ascending Ansi order.

Each aberration has a different impact on optical quality, requiring a detailed analysis to translate e.g. a scientific requirement on optical quality into a WFE allocation.

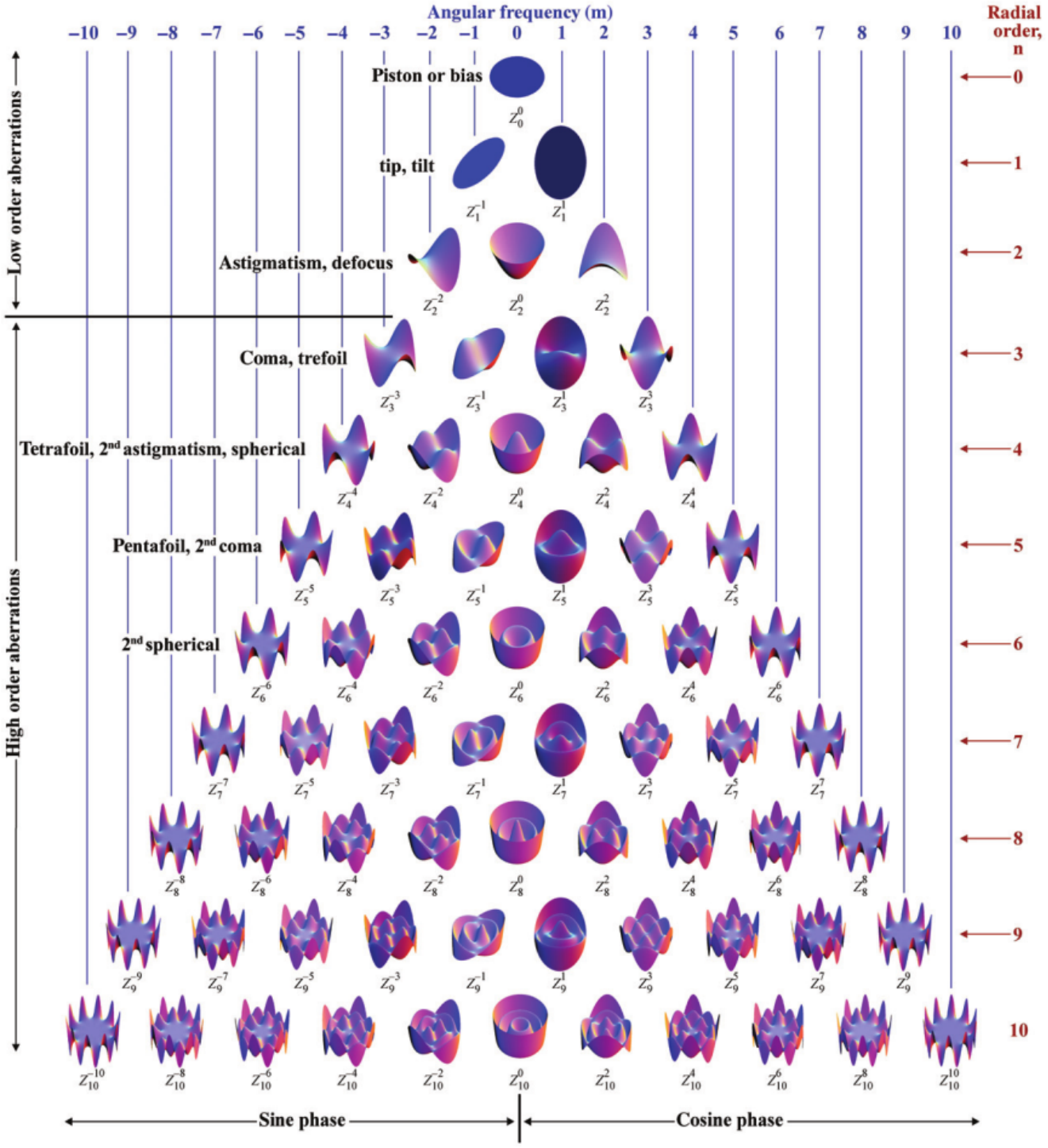


Fig. 3.23 – *Zernike polynomials surface plots*

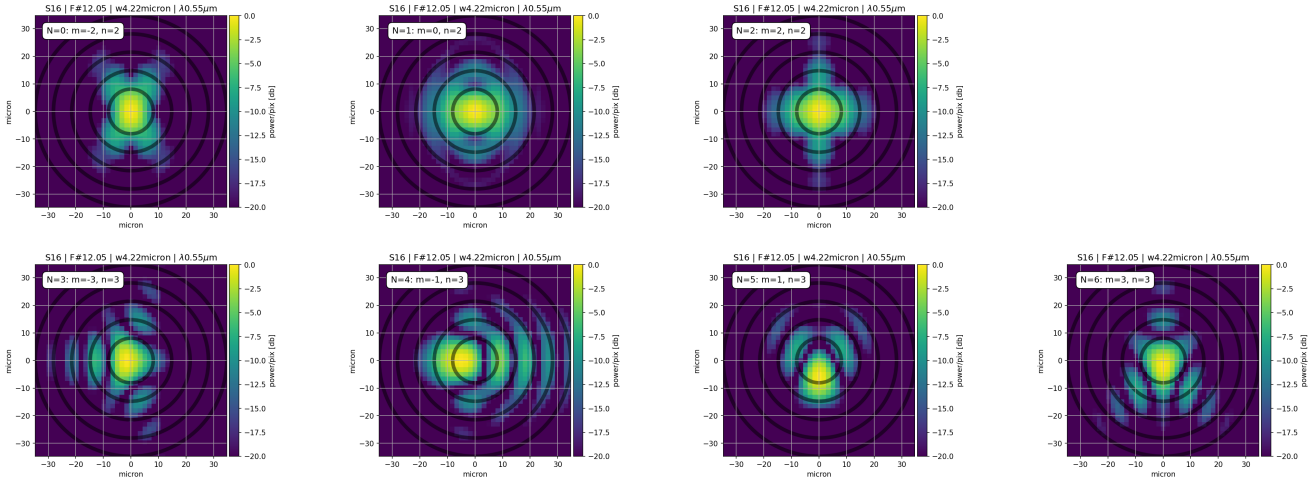


Fig. 3.24 – *Ariel Telescope exit pupil PSFs for different aberrations and same SFE*

3.5.3 Surface roughness

Optical elements exhibit surface roughness, i.e. medium to high frequency defects produced during manufacturing (e.g. using diamond turning machines). The resulting aberrations can be described as a zero-mean random Gaussian field with variance σ_G.

Surface roughness is not yet developed in the main *PAOS* code.

3.6 Materials description

Brief description of dispersion of light by optical materials and how it is implemented in *PAOS*.

In *PAOS*, this is handled by the class `Material`.

3.6.1 Light dispersion

In optics, dispersion is the phenomenon in which the phase velocity of a wave depends on its frequency:

$$v = \frac{c}{n}$$

where c is the speed of light in a vacuum and n is the refractive index of the dispersive medium. Physically, dispersion translates in a loss of kinetic energy through absorption. The absorption by the dispersive medium is different at different wavelengths, changing the angle of refraction of different colors of light as seen in the spectrum produced by a dispersive *Prism* and in chromatic aberration of *Thick lenses*.

This can be seen in geometric optics from Snell's law:

$$\frac{sin(\theta_2)}{sin(\theta_1)} = \frac{n_1}{n_2}$$

that describes the relationship between the angle of incidence θ_1 and refraction θ_2 of light passing through a boundary between an isotropic medium with refractive index n_1 and another with n_2.

For air and optical glasses, for visible and infra-red light refraction indices n decrease with increasing λ (*normal dispersion*), i.e.

$$\frac{dn}{d\lambda} < 0$$

while for ultraviolet the opposite behaviour is typically the case (anomalous dispersion).

See later in *Supported materials* for the dispersion behaviour of supported optical materials in *PAOS*.

3.6.2 Sellmeier equation

The Sellmeier equation is an empirical relationship for the dispersion of light in a particular transparent medium such as an optical glass in function of wavelength. In its original form (Sellmeier, 1872) it is given as

$$n^2(\lambda) = 1 + \sum_i \frac{K_i \lambda^2}{\lambda^2 - L_i} \tag{3.45}$$

where n is the refractive index, λ is the wavelength and K_i and $\sqrt{L_i}$ are the Sellmeier coefficients, determined from experiments.

Physically, each term of the sum represents an absorption resonance of strength K_i at wavelength $\sqrt{L_i}$. Close to each absorption peak, a more precise model of dispersion is required to avoid non-physical values.

PAOS implements the Sellmeier 1 equation (Zemax OpticStudio [I] notation) to estimate the index of refraction relative to air for a particular optical glass at the glass reference temperature and pressure

$$\begin{aligned} T_{ref} &= 20°C \\ P_{ref} &= 1 \text{ atm} \end{aligned} \tag{3.46}$$

This form of the original equation consists of only three terms and is given as

$$n^2(\lambda) = 1 + \frac{K_1 \lambda^2}{\lambda^2 - L_1} + \frac{K_2 \lambda^2}{\lambda^2 - L_2} + \frac{K_3 \lambda^2}{\lambda^2 - L_3} \tag{3.47}$$

The resulting refracting index should deviate by less than 10^{-6} from the actual refractive index which is order of the homogeneity of a glass sample (see e.g. Optical properties).

3.6.2.1 Example

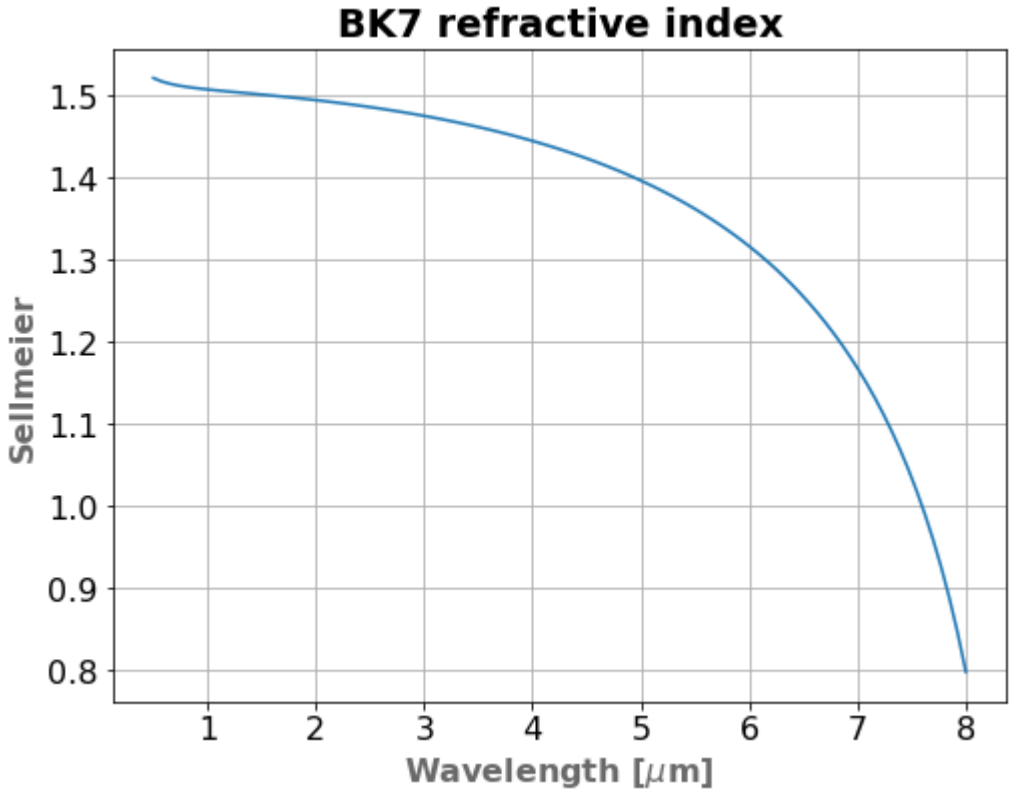
Code example to use *Material* to estimate and plot the index of refraction of borosilicate crown glass (known as *BK7*) for a range of wavelengths from the visible to the infra-red.

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

from paos.util.material import Material

wl = np.linspace(0.5, 8.0, 100)
mat = Material(wl=wl)

glass = 'BK7'
```

(continues on next page)

(continued from previous page)

```
material = mat.materials[glass]
sellmeier = mat.sellmeier(material['sellmeier'])

fig = plt.figure(figsize=(8, 6))
ax = fig.add_subplot(1,1,1)
ax.plot(wl, sellmeier)
ax.set_title(f'{glass} refractive index')
ax.set_ylabel('Sellmeier')
ax.set_xlabel(r'Wavelength [$\mu$m]')
plt.grid()
plt.show()
```



3.6.3 Temperature and refractive index

Changes in the temperature of the dispersive medium affect the refractive index. The temperature coefficient of refractive index is defined as the deviation dn/dT from the curve and depends from both wavelength and temperature.

The temperature coefficient values can be given as absolute (as measured under vacuum) and relative (as measured at ambient air (dry air at standard pressure).

PAOS estimates the air reference index of refraction as

$$n_{ref} = 1.0 + 1.0 \cdot 10^{-8} \left(6432.8 + \frac{2949810\lambda^2}{146\lambda^2 - 1} + 25540 \frac{\lambda^2}{41\lambda^2 - 1} \right) \tag{3.48}$$

where λ is in units of micron, at the reference temperature $T = 15°C$ and standard pressure. Under different temperatures and pressures, *PAOS* rescales this reference index using this formula

$$n_{air} = 1 + \frac{P\,(n_{ref} - 1)}{1.0 + 3.4785 \cdot 10^{-3}(T - 15)} \tag{3.49}$$

The absolute temperature coefficient for a different medium can be calculated from the relative index as (see e.g. Optical properties).

$$\frac{dn}{dT}, \text{absolute} = \frac{dn}{dT}, \text{relative} + n\left(\frac{dn}{dT}, \text{air}\right) \tag{3.50}$$

PAOS calculates the refractive index of an optical material at a given pressure and temperature as

$$n(\Delta T) = \frac{n^2 - 1}{2n} D_0 \Delta T + n \tag{3.51}$$

where ΔT is given by the difference between the material operative temperature T_{oper} and the reference temperature T_{ref}, n is the refractive index as estimated using (3.47) and D_0 is a temperature constant of the material.

3.6.3.1 Example

Code example to use *Material* to estimate the index of refraction of borosilicate crown glass (known as *BK7*) for a given wavelength at reference and operating temperature.

```
from paos.util.material import Material

wl = 1.95  # micron
Tref, Tambient = 20.0, -223.0
mat = Material(wl, Tambient=Tambient)
glass = 'BK7'
nmat0, nmat = mat.nmat(glass)

from IPython.display import display, Latex
display(Latex("\\textrm{Index of refraction at } T_{ref} = %0.1f:\\newline n_{%s, 0}␣
↪= %0.4f " % (Tref, glass, nmat0)))
display(Latex("\\textrm{Index of refraction at } T_{amb} = %0.1f:\\newline n_{%s, 0}␣
↪= %0.4f " % (Tambient, glass, nmat)))
```

Index of refraction at $T_{ref} = 20.0 : n_{BK7,0} = 1.4956$

Index of refraction at $T_{amb} = -223.0 : n_{BK7,0} = 1.4955$

3.6.4 Pressure and refractive index

Note also that *PAOS* can easily model systems used in a vacuum by changing the air pressure to zero.

3.6.4.1 Example

Same code example as before, but ambient pressure is set to zero.

```
mat = Material(wl, Tambient=Tambient, Pambient=0.0)
nmat0, nmat = mat.nmat(glass)

from IPython.display import display, Latex
display(Latex("\\textrm{Index of refraction at } T_{ref} = %0.1f:\\newline n_{%s, 0}␣
↪= %0.4f " % (Tref, glass, nmat0)))
display(Latex("\\textrm{Index of refraction at } T_{amb} = %0.1f:\\newline n_{%s, 0}␣
↪= %0.4f " % (Tambient, glass, nmat)))
```

Index of refraction at $T_{ref} = 20.0 : n_{BK7,0} = 1.4952$

Index of refraction at $T_{amb} = -223.0 : n_{BK7,0} = 1.4951$

Note the non-negligible difference in the resulting refractive indexes.

3.6.5 Supported materials

PAOS supports a variety of optical materials (list is still updating), among which:

1. CAF2 (calcium fluoride)
2. SAPPHIRE (mainly aluminium oxide ($\alpha - Al_2O_3$))
3. ZNSE (zinc selenide)
4. BK7 (borosilicate crown glass)
5. SF11 (a dense-flint glass)
6. BAF2 (barium flouride)

The relevant ones for the *Ariel* space mission are all of them except BAF2. A detailed description of the optical properties of these materials is beyond the scope of this documentation. However, for reference, Fig. 3.25 reports their transmission range (from Thorlabs, Optical Substrates).

3.6.5.1 Example

Code example to use *Material* to print all available optical materials.

```
from paos.util.material import Material

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

```
Supported materials:
CAF2
SAPPHIRE
ZNSE
BK7
SF11
BAF2
```

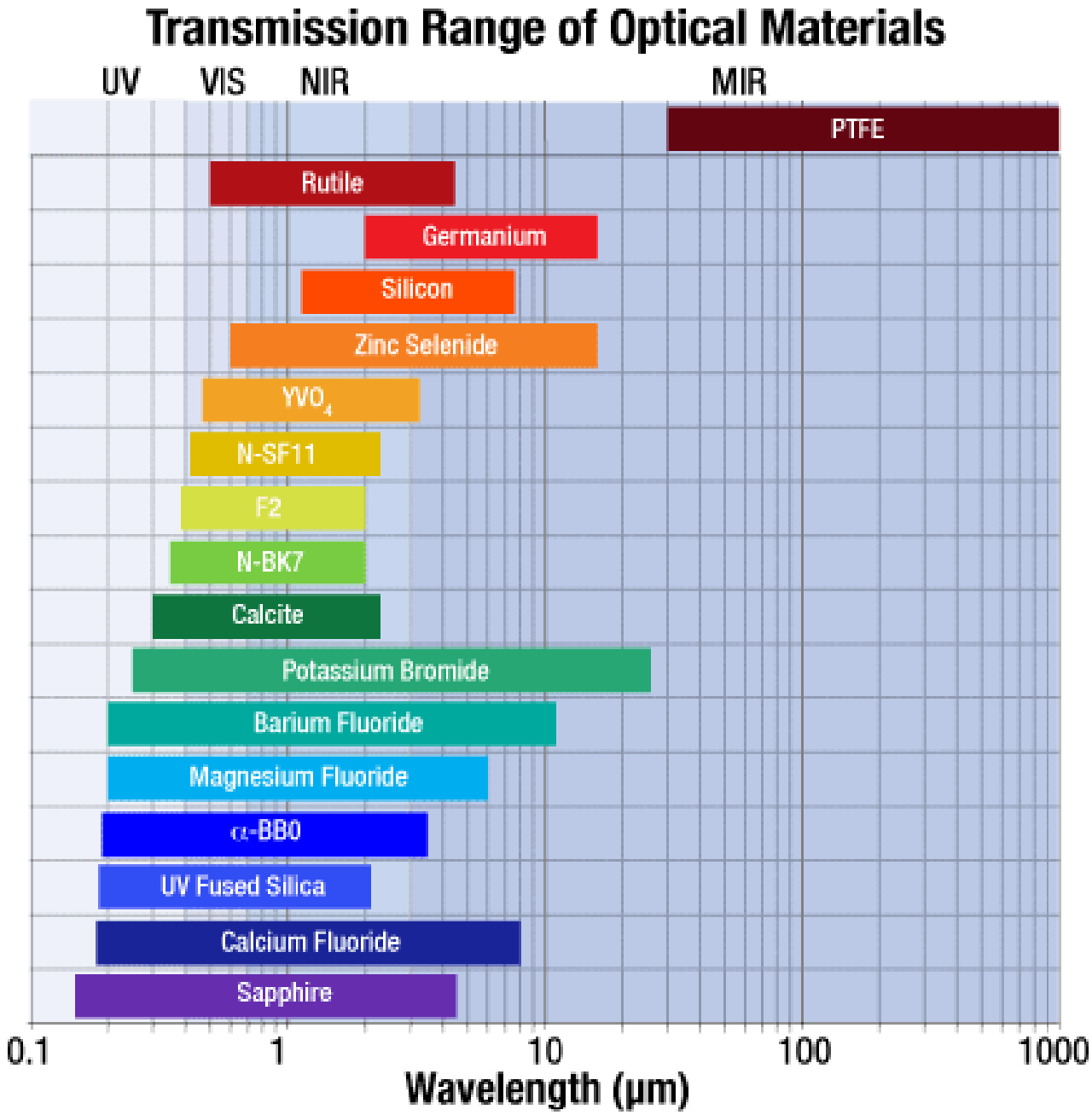
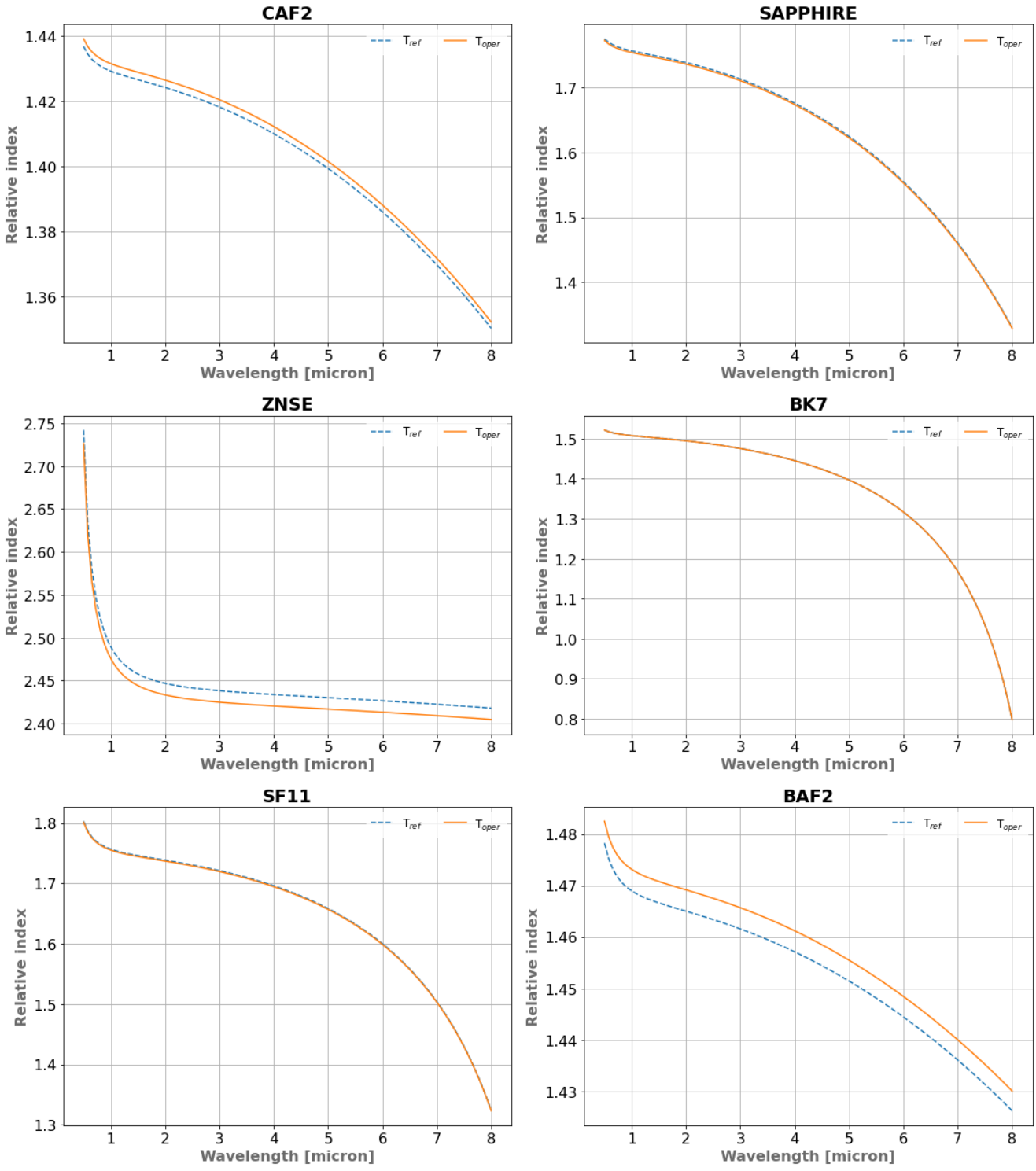


Fig. 3.25 – *Transmission range of optical substrates (Thorlabs)*

3.6.5.2 Example

Code example to use *Material* to plot the refractive index for all available optical materials, at their operating and reference temperature.

```
from paos.util.material import Material

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



3.7 Plotting results

PAOS implements different plotting routines, summarized here, that can be used to give a complementary idea of the main POP simulation results.

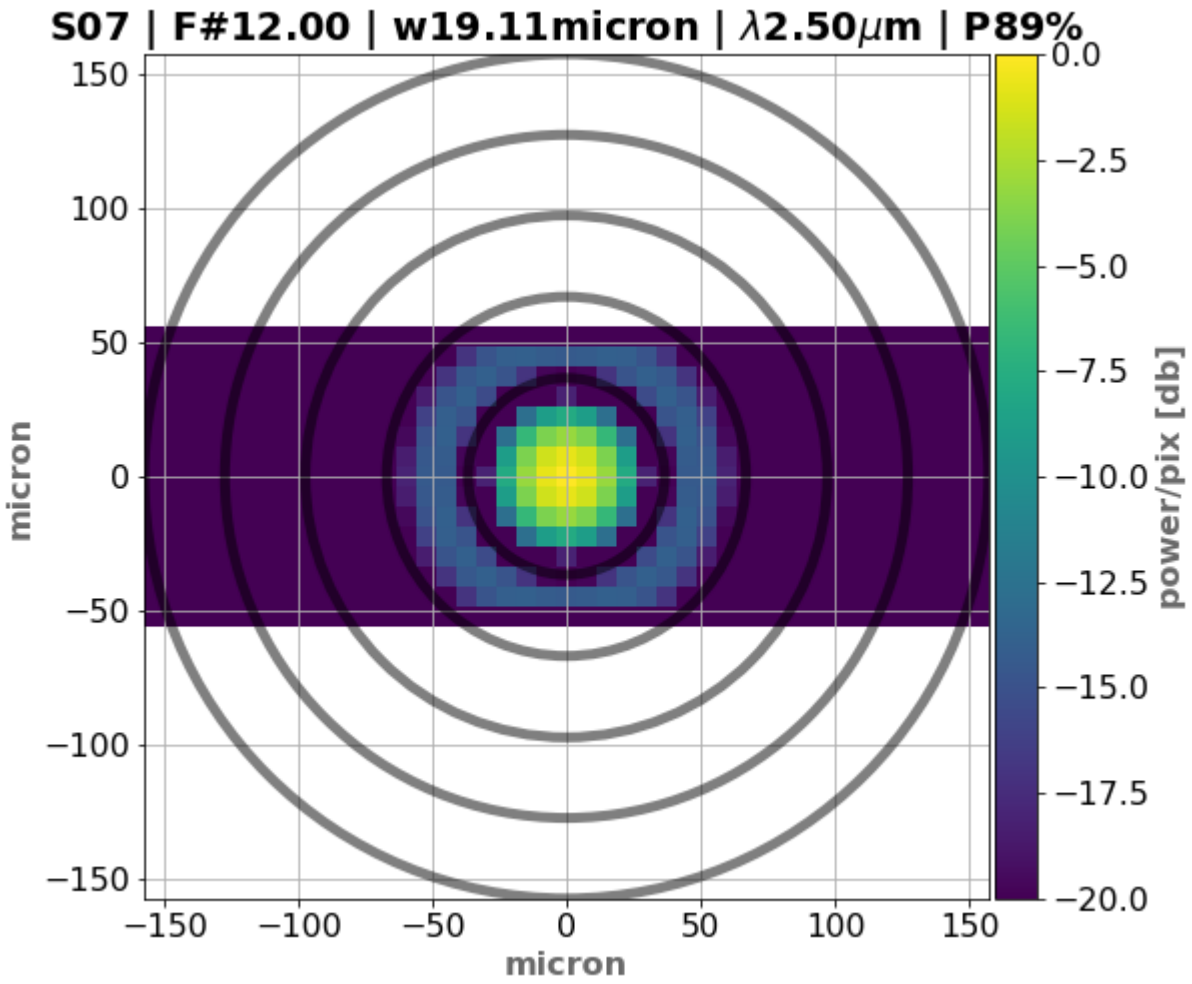
3.7.1 Base plot

The base plot method, *simple_plot*, receives as input the POP output dictionary and the dictionary key of one optical surface and plots the squared amplitude of the wavefront at the given optical surface.

3.7.1.1 Example

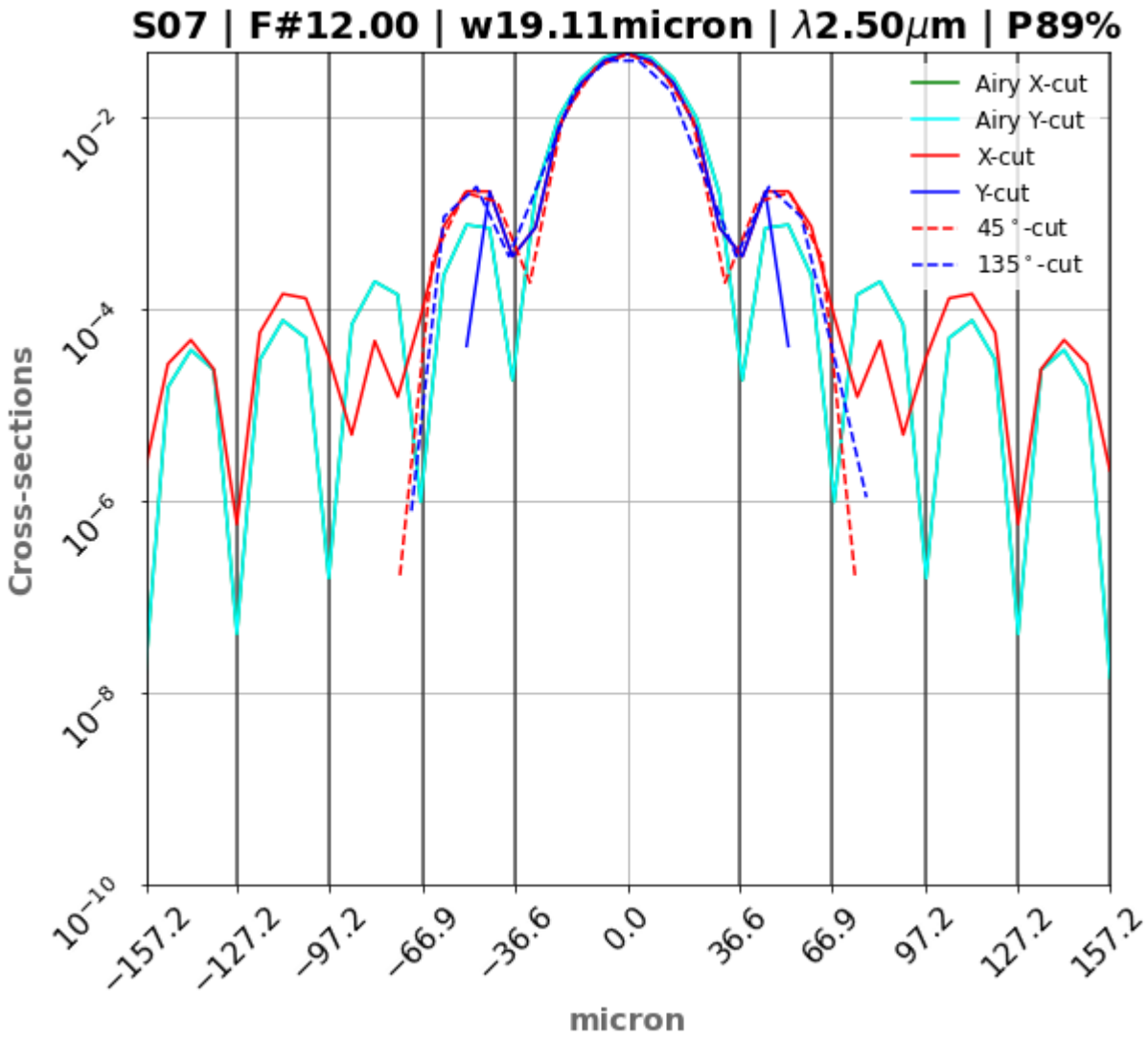
Code example to use *simple_plot* to plot the expected PSF at the image plane of the EXCITE optical chain.

```
import matplotlib.pyplot as plt
from paos.paos_plotpop import simple_plot

fig = plt.figure(figsize=(8, 8))
ax = fig.add_subplot(1,1,1)

key = list(ret_val.keys())[-1]  # plot at last optical surface
simple_plot(fig, ax, key=key, item=ret_val[key], ima_scale='log')

plt.show()
```



The cross-sections for this PSF can be plotted using the method *plot_psf_xsec*, as shown below.

```
from paos.paos_plotpop import plot_psf_xsec

fig = plt.figure(figsize=(9, 8))
ax = fig.add_subplot(1,1,1)

key = list(ret_val.keys())[-1]  # plot at last optical surface
plot_psf_xsec(fig, ax, key=key, item=ret_val[key], ima_scale='log')

plt.show()
```



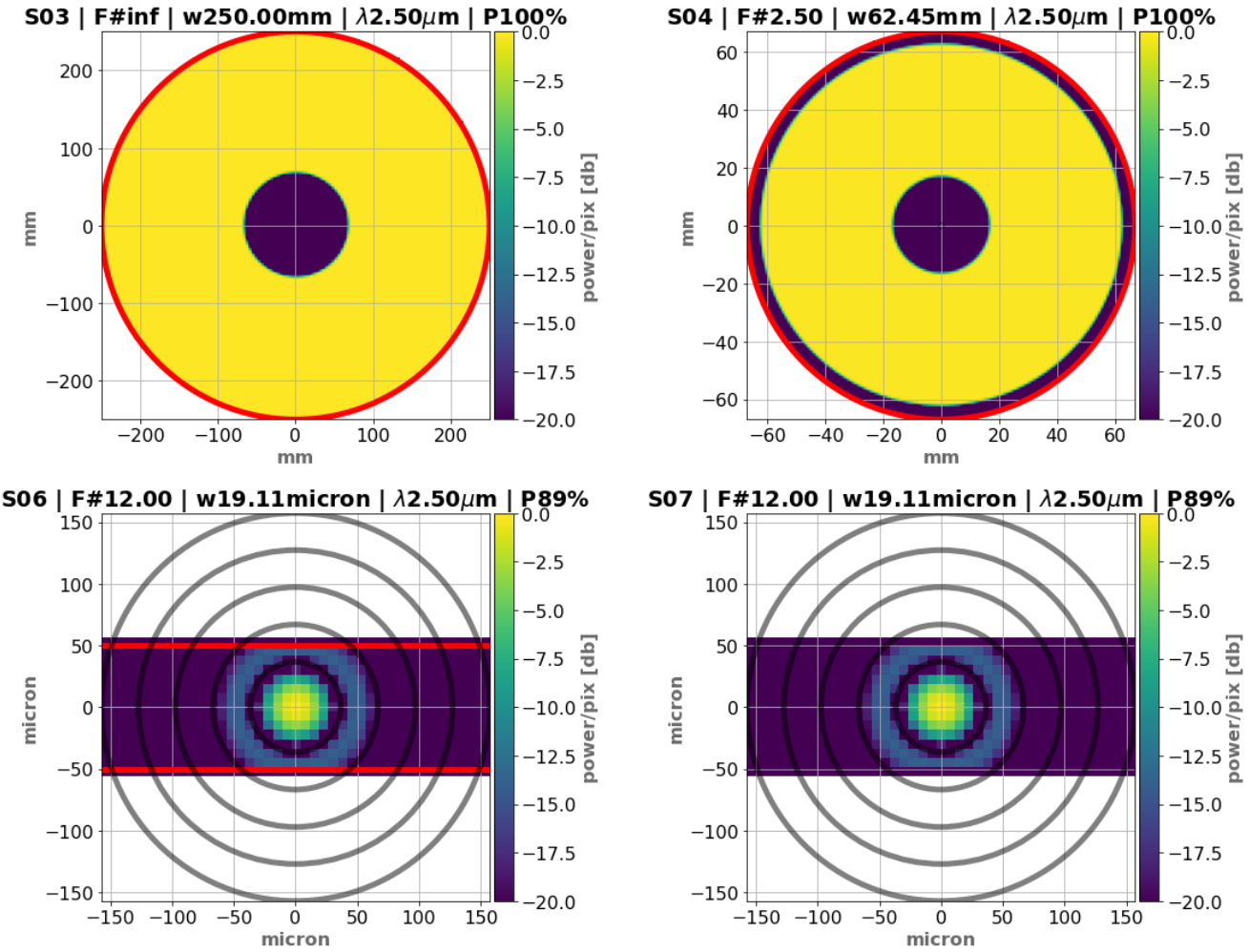
3.7.2 POP plot

The POP plot method, *plot_pop*, receives as input the POP output dictionary plots the squared amplitude of the wavefront at all available optical surfaces.

3.7.2.1 Example

Code example to use *plot_pop* to plot the squared amplitude of the wavefront at all surfaces of the EXCITE optical chain.

```
from paos.paos_plotpop import plot_pop
plot_pop(ret_val, ima_scale='log', ncols=2)
```



3.8 Saving results

PAOS implements different saving routines, summarized here, that can be used to save the main POP simulation results.

3.8.1 Save output

The base saving method, *save_output*, receives as input the POP simulation output dictionary, a hdf5 file name and the keys to store at each surface and saves the dictionary along with the *PAOS* package information to the hdf5 output file. If indicated, this function overwrites a previously saved file.

The hdf5 file is structured in two sub-folders, as shown in Fig. 3.26. The first one is labelled with the wavelength used in the simulation, while the other is labelled 'info'.

The first folder contains a list of sub-folders, in which is stored the data relative to the individual optical surfaces. Each surface is labelled as 'S#' where # is the surface index, as shown in Fig. 3.27.

The 'info' folder contains the data that are needed for traceability and versioning of the results, as shown in Fig. 3.28.

This includes:

1. The HDF5 package version
2. The *PAOS* creator names

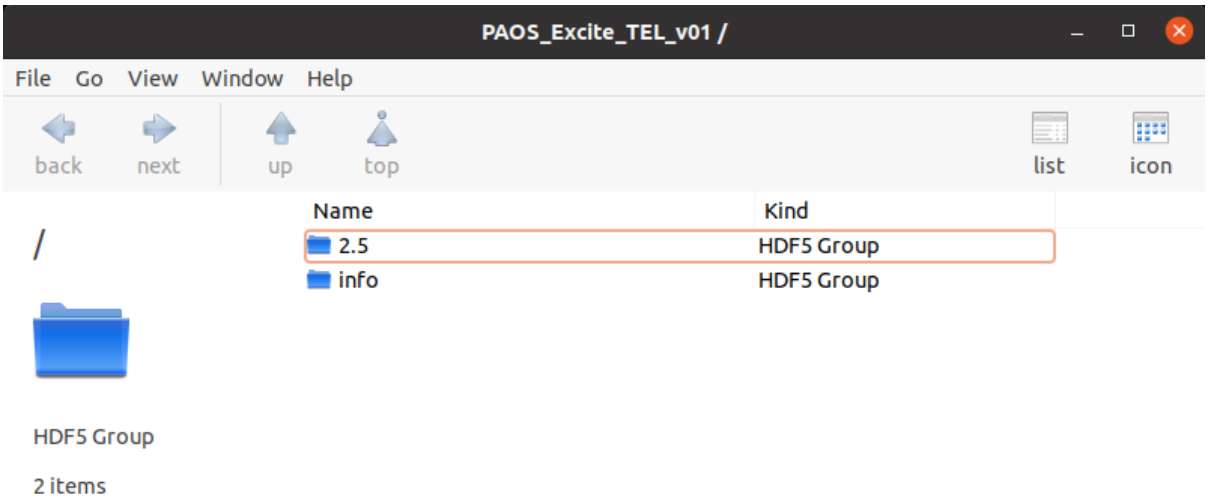


Fig. 3.26 – *Output file general interface*

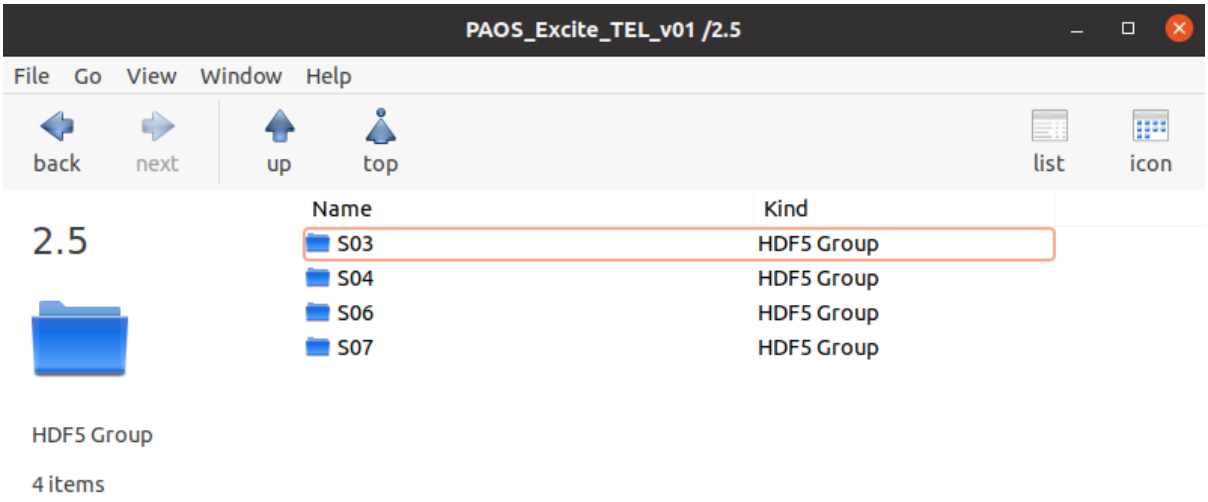


Fig. 3.27 – *Output file surfaces interface*

3. The saving path
4. The saving time in human readable format
5. The h5py version
6. This package's name
7. This package's version

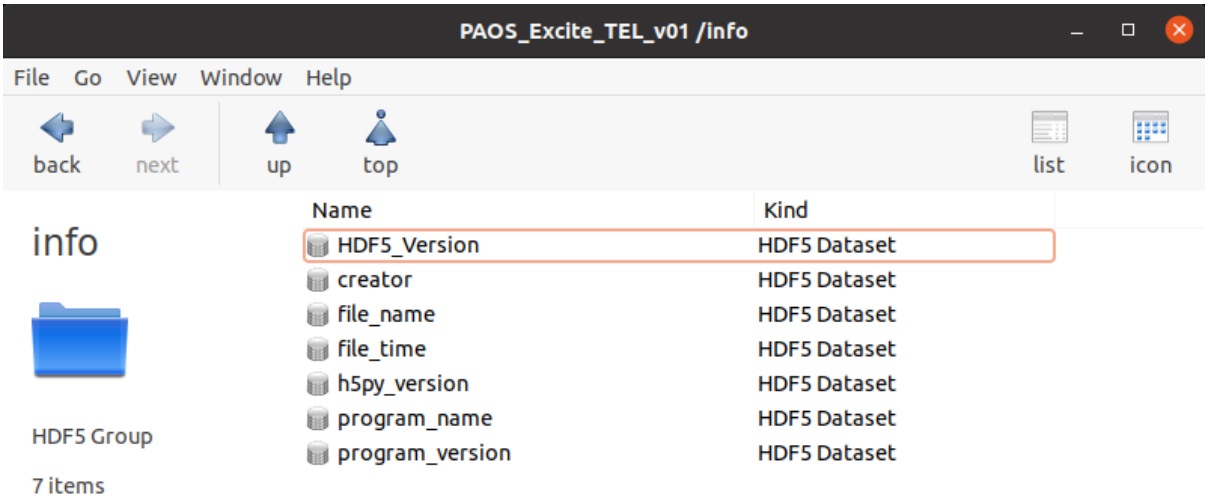


Fig. 3.28 – *Output file info interface*

3.8.1.1 Example

Code example to use *save_output* to save the POP simulation output dictionary.

The user can select to save only the relevant dictionary keys, here 'wfo' (the complex wavefront array), 'dx' (the sampling along the horizontal axis), 'dy' (the sampling along the vertical axis).

```
from paos.paos_saveoutput import save_output
save_output(ret_val, '../output/test.h5', keys_to_keep=['wfo', 'dx', 'dy'],␣
↪overwrite=True)
```

```
paos - INFO - saving ../output/test.h5 started...

paos - INFO - removing old file

paos - INFO - saving ended.
```

3.8.2 Save datacube

The *save_datacube* method receives as input a list of output dictionaries for each POP simulation, a hdf5 file name, a list of identifiers to tag each simulation and the relevant keys to store at each surface, and saves all the outputs to a data cube stored in the hdf5 output file. If indicated, this method overwrites a previously saved file.

Fig. 3.29

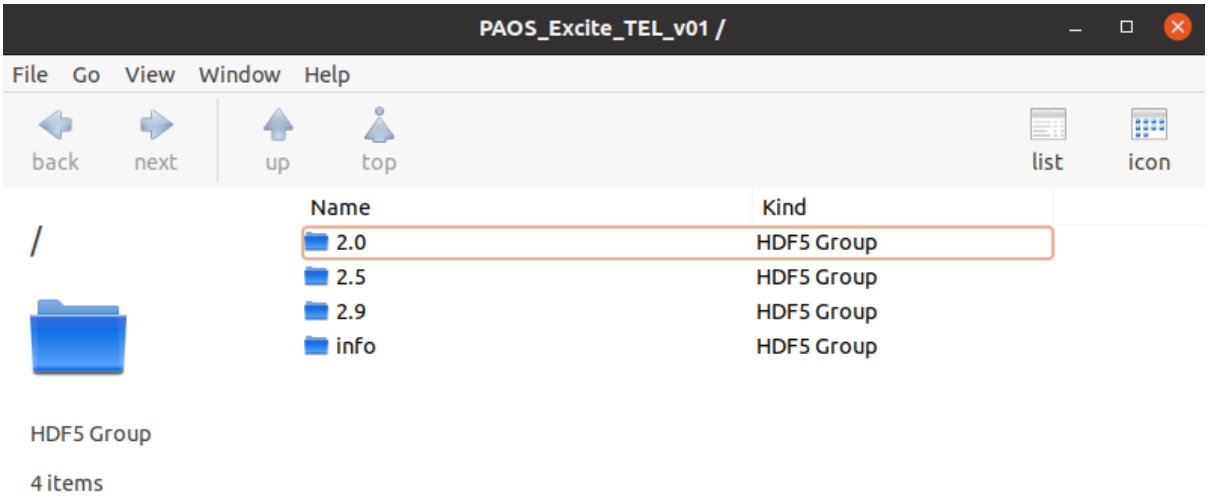


Fig. 3.29 – *Output file cube general interface*

3.8.2.1 Example

Code example to use *save_datacube* to save the output dictionary for multiple POP simulations done at different wavelengths.

The user can select to save only the relevant dictionary keys, here 'amplitude' (the wavefront amplitude), 'dx' (the sampling along the horizontal axis), 'dy' (the sampling along the vertical axis).

```
print(f'Saving wavelengths: {wavelengths}')
group_names = list(map(str, wavelengths))
```

```
Saving wavelengths: [3.9, 5.85, 7.8]
```

```
from paos.paos_saveoutput import save_datacube

save_datacube(retval_list=ret_val_list,
      file_name='../output/test.h5',
      group_names=group_names,
      keys_to_keep=['amplitude', 'dx', 'dy'],
      overwrite=True)
```

```
paos - INFO - Saving ../output/test.h5 started...

paos - INFO - Remove old file

paos - INFO - Saving ended.
```

3.9 Automatic pipeline

Pipeline to run a POP simulation and save the results, given an input dictionary with selected options.

3.9.1 Base pipeline

This pipeline

1. Sets up the logger;
2. Parses the lens file;
3. Performs a diagnostic ray tracing (optional);
4. Sets up the simulation wavelength or produces a user defined wavelength grid;
5. Sets up the optical chain for the POP run automatizing the input of an aberration (optional);
6. Runs the POP in parallel or using a single thread;
7. Produces an output where all (or a subset) of the products are stored;
8. If indicated, the output includes plots.

3.9.1.1 Example

Code example to the method *pipeline* to run a simulation for two wavelengths, $w_1 = 3.9$ and $w_1 = 7.8$ micron, using the configuration file for AIRS-CH1.

Using the option 'wl_grid' instead of 'wavelengths', the user can define the minimum wavelength, the maximum wavelength and the spectral resolution. *PAOS* will then automatically create a wavelength grid to perform the POP.

```
from paos.paos_pipeline import pipeline

pipeline(passvalue={'conf':'../lens data/Ariel_AIRS-CH1.ini',
                    'output': '../output/test.h5',
                    'wavelengths': '3.9,7.8',
                    # or 'wl_grid': '3.9,7.8,5'
                    'plot': True,
                    'loglevel': 'info',
                    'n_jobs': 2,
                    'store_keys': 'amplitude,dx,dy,wl',
                    'return': False})
```

```
code version 0.0.4


paos - INFO - Parse lens file


paos - INFO - Set up the POP


paos - INFO - Run the POP


paos - INFO - Start POP in parallel...


  0%|          | 0/2 [00:00<?, ?it/s]
```

```
100%|| 2/2 [00:00<00:00, 41.07it/s]


paos - INFO - POP completed in    3.1s

paos - INFO - Save POP simulation output .h5 file to ../output/test.h5

paos - INFO - Saving ../output/test.h5 started...

paos - INFO - Remove old file

paos - INFO - Saving ended.

paos - INFO - Save POP simulation output plot


  0%|          | 0/2 [00:00<?, ?it/s]

100%|| 2/2 [00:00<00:00, 987.94it/s]


paos - INFO - Plotting completed in    5.2s
```

Chapter 4

Validation

PAOS has been validated against the code PROPER (John E. Krist, PROPER: an optical propagation library for IDL, Proc. SPIE, 6675 (2007)), a library of optical propagation procedures and functions for the IDL (Interactive Data Language), Python, and Matlab environments.

The validation has been carried out using the Hubble Space Telescope (HST) optical chain as described in one of PROPER example prescriptions. The corresponding configuration file is provided in the *PAOS* package for reproducibility.

The input pupil is uniformly illuminated and is apodized by the primary mirror circular pads (1, 2, and 3), the secondary mirror circular obscuration and rectangular vanes (vertical and horizontal). From M1, the wavefront is propagated to the secondary mirror (M2), then it is brought to the HST focus.

Below, we show the plots for the squared amplitude of the wavefront at all surfaces along the HST optical chain, using *PROPER* (Fig. 4.1) and *PAOS* (Fig. 4.2).

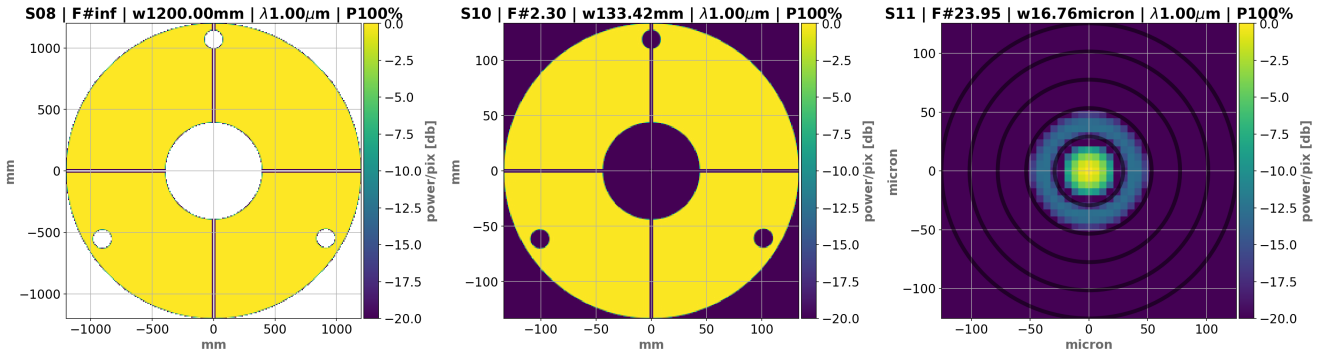


Fig. 4.1 – *PROPER HST*

For the validation, we compared the amplitude (Fig. 4.3), phase (Fig. 4.4) and PSF (Fig. 4.5) at the HST focus as obtained by *PROPER* and *PAOS*, along with their differences. The comparison shows excellent agreement, and small discrepancies (PSF: $< 10^{-10}$) between the two codes are mostly due to aliasing from the aperture masks.

Note: The phase convention used in the two codes differs by a sign, therefore in the comparison the *PAOS* phase is multiplied by -1. Moreover, the maximum value of the difference is due to individual aliased pixels far from the beam axis.

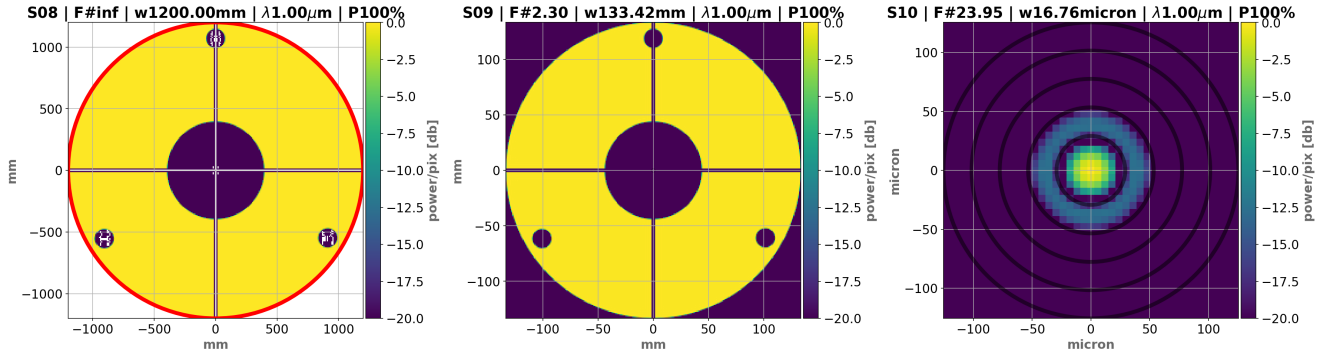


Fig. 4.2 – *PAOS HST*

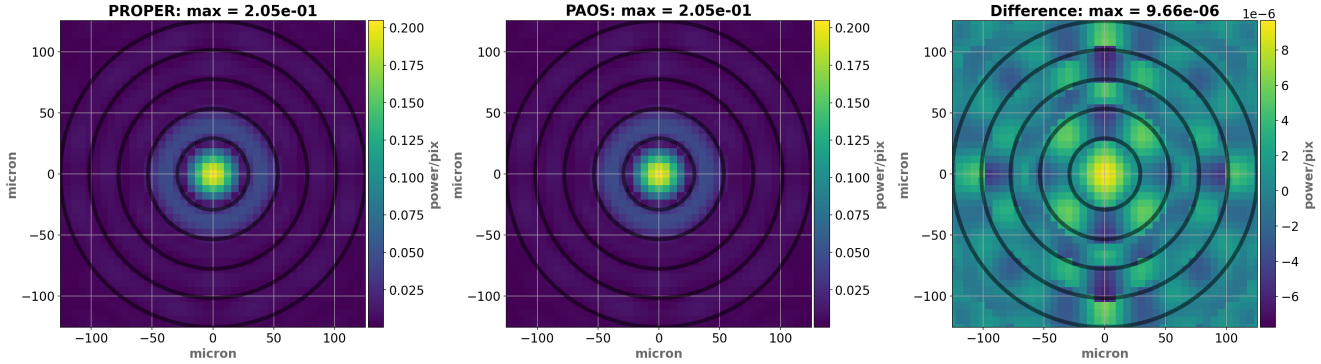


Fig. 4.3 – *Amplitude comparison*

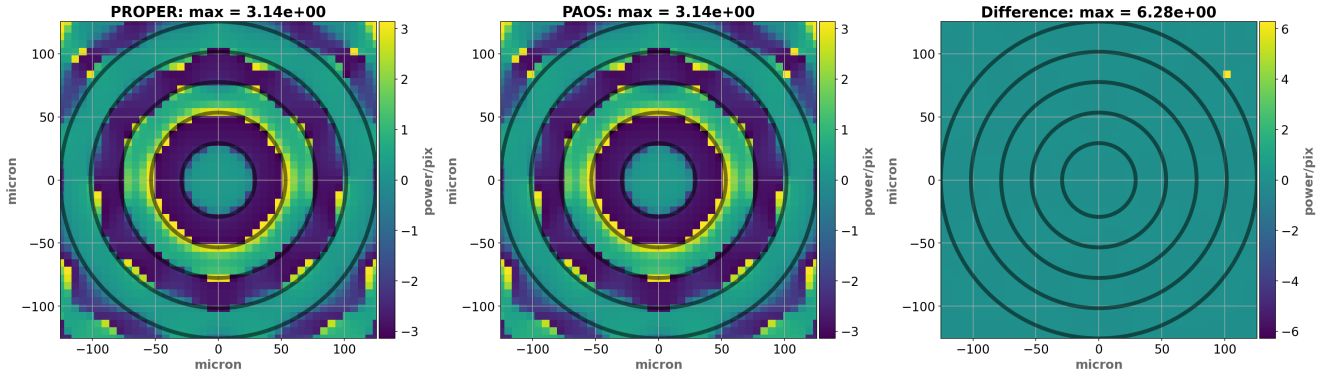


Fig. 4.4 – *Phase comparison*

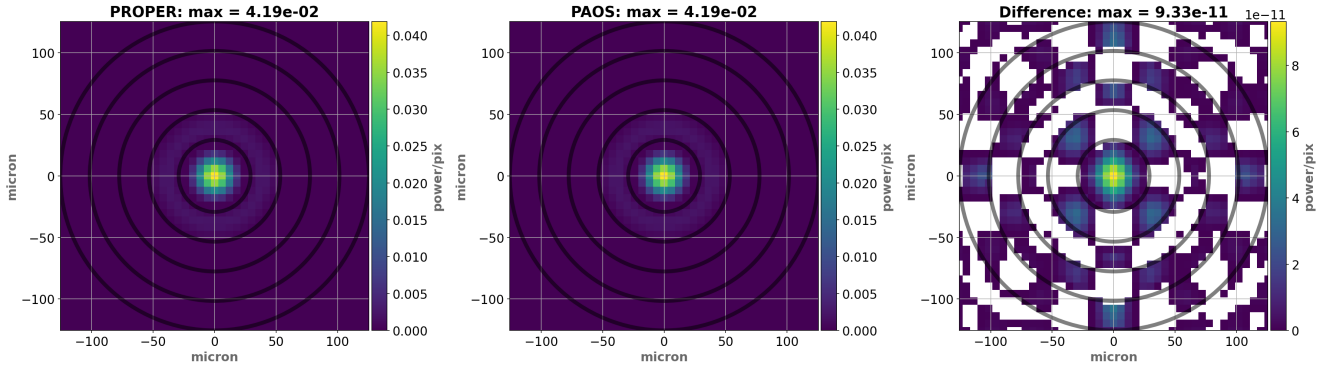


Fig. 4.5 – *PSF comparison*

Chapter 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*)

5.1 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 μm. 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.

5.1.1 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.

5.1.2 SNR

placeholder

5.1.3 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.

5.1.4 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.

5.1.5 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.

5.2 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.

Chapter 6

API Guide

6.1 *PAOS* package

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

6.1.1 Modules

6.1.1.1 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 n2
- **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

6.1.1.2 paos.paos_coordinatebreak module

coordinate_break(*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.

6.1.1.3 paos.paos_parseconfig module

`getfloat`(*value*)

`parse_config`(*filename*)

Parse an ini lens file

Parameters `filename` (*string*) – full path to ini file

Returns

- **pup_diameter** (*float*) – pupil diameter in lens units
- **parameters** (*dict*) – Dictionary with parameters defined in the section 'general' of the ini file
- **field** (*List*) – list of fields
- **wavelengths** (*List*) – list of wavelengths
- **opt_chain_list** (*List*) – Each list entry is a dictionary of the optical surfaces in the ini file, estimated at the give wavelength. (Relevant only for diffractive components)

Examples

```
>>> from paos.paos_parseconfig import parse_config
>>> pup_diameter, parameters, wavelengths, fields, opt_chains = parse_config(
↪'path/to/ini/file')
```

6.1.1.4 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',
```

(continues on next page)

(continued from previous page)

```
>>>                     'plot': True,
>>>                     'loglevel': 'debug',
>>>                     'n_jobs': 2,
>>>                     'store_keys': 'amplitude,dx,dy,wl',
>>>                     'return': False})
```

6.1.1.5 paos.paos_plotpop module

do_legend(*axis*, *ncol=1*)

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

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

Parameters

- **fig** (Figure) – instance of matplotlib figure artist
- **axis** (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'
- **options** (*dict*) – dict containing the options to display the plot: axis scale, option to display physical units, zoom scale and color scale. Examples: 0) options={4: {'ima_scale':'linear'}} 1) options={4: {'surface_scale':60, 'ima_scale':'linear'}} 2) options={4: {'surface_scale':21, 'pixel_units':True, 'ima_scale':'linear'}} 3) options={4: {'surface_zoom':2, 'ima_scale':'log'}}

Returns updates the Figure object

Return type None

Examples

```
>>> from paos.paos_parseconfig import parse_config
>>> from paos.paos_run import run
>>> from paos.paos_plotpop import simple_plot
>>> pup_diameter, parameters, wavelengths, fields, opt_chains = parse_config(
↪'path/to/ini/file')
>>> ret_val = run(pup_diameter, 1.0e-6 * wavelengths[0], parameters['grid size'],
>>>               parameters['zoom'], fields[0], opt_chains[0])
>>> 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*, *options={}*)

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
- **options** (*dict*) – dict containing the options to display the plot: axis scale, axis unit, zoom scale and color scale. Examples: 0) options={4: {'ima_scale':'linear'}} 1) options={4: {'surface_scale':60, 'ima_scale':'linear'}} 2) options={4: {'surface_scale':21, 'pixel_units':True, 'ima_scale':'linear'}} 3) options={4: {'surface_zoom':2, 'ima_scale':'log'}}

Returns out – displays the plot output or stores it to the indicated plot path
Return type None

Examples

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

plot_psf_xsec(*fig*, *axis*, *key*, *item*, *ima_scale='linear'*, *x_units='standard'*)

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

Parameters

- **fig** (Figure) – instance of matplotlib figure artist
- **key** (*int*) – optical surface index
- **item** (*dict*) – optical surface dict
- **ima_scale** (*str*) – y axis scale, can be either 'linear' or 'log'
- **x_units** (*str*) – units for x axis. Default is 'standard', to have units of mm or microns. Can also be 'wave', i.e. Displacement/($F_{num}\lambda$).

Returns out – updates the *~matplotlib.figure.Figure* object
Return type None

Examples

```
>>> import matplotlib.pyplot as plt
>>> from paos.paos_parseconfig import parse_config
>>> from paos.paos_run import run
>>> from paos.paos_plotpop import plot_psf_xsec
>>> pup_diameter, parameters, wavelengths, fields, opt_chains = parse_config('/
↪path/to/config/file')
>>> wl_idx = 0  # choose the first wavelength
>>> wavelength, opt_chain = wavelengths[wl_idx], opt_chains[wl_idx]
>>> ret_val = run(pup_diameter, 1.0e-6 * wavelength, parameters['grid_size'],␣
↪parameters['zoom'],
>>>               fields[0], opt_chain)
>>> key = list(ret_val.keys())[-1]  # plot at last optical surface
```

(continues on next page)

(continued from previous page)

```
>>> fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(16, 8))
>>> plot_psf_xsec(fig=fig, axis=ax, key=key, item=ret_val[key], ima_scale='log',␣
↪x_units='wave')
```

6.1.1.6 paos.paos_raytrace module

raytrace(*field*, *opt_chain*, *x=0.0*, *y=0.0*)

Diagnostic function that implements the Paraxial 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** (*dict*) – contains the slopes in the tangential and sagittal planes as field={'vt': slopey, 'vs': slopex}
- **opt_chain** (*dict*) – the dict of the optical elements returned by paos.parse_config
- **x** (*float*) – X-coordinate of the initial ray position
- **y** (*float*) – Y-coordinate of the initial ray position

Returns out – A list of strings where each list item is the raytrace at a given surface.

Return type list[str]

Examples

```
>>> from paos.paos_parseconfig import parse_config
>>> from paos.paos_raytrace import raytrace
>>> pup_diameter, parameters, wavelengths, fields, opt_chains = parse_config(
↪'path/to/conf/file')
>>> raytrace(fields[0], opt_chains[0])
```

6.1.1.7 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 parse_config
>>> from paos.paos_run import run
>>> from paos.paos_plotpop import simple_plot
>>> pup_diameter, parameters, wavelengths, fields, opt_chains = parse_config(
↪'path/to/conf/file')
>>> ret_val = run(pup_diameter, 1.0e-6 * wavelength, parameters['grid_size'],␣
↪parameters['zoom'], fields[0], opt_chain)
```

6.1.1.8 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 parse_config
>>> from paos.paos_run import run
>>> from paos.paos_saveoutput import save_output
>>> pup_diameter, parameters, wavelengths, fields, opt_chains = parse_config(
↪'path/to/ini/file')
>>> ret_val = run(pup_diameter, 1.0e-6 * wavelengths[0], parameters['grid size'],
>>>               parameters['zoom'], fields[0], opt_chains[0])
>>> 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 parse_config
>>> from paos.paos_run import run
>>> from paos.paos_saveoutput import save_datacube
>>> from joblib import Parallel, delayed
>>> from tqdm import tqdm
>>> pup_diameter, parameters, wavelengths, fields, opt_chains = parse_config(
↪'path/to/ini/file')
>>> ret_val_list = Parallel(n_jobs=2)(delayed(run)(pup_diameter, 1.0e-6 * wl,␣
↪parameters['grid size'],
>>>                parameters['zoom'], fields[0], opt_chains[0]) 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)
```

6.1.1.9 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
```

(continues on next page)

(continued from previous page)

```
>>> 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)
>>> 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’, ‘standard’, ‘noll’, ‘fringe’

Returns **m, n**

Return type array

static 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

6.1.1.10 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 matplotlib.pyplot as plt
>>> beam_diameter = 1.0  # m
>>> wavelength = 3.0  # micron
>>> grid_size = 512
>>> zoom = 4
```

(continues on next page)

(continued from previous page)

```
>>> xdec, ydec = 0.0, 0.0
>>> fig, (ax0, ax1) = plt.subplots(nrows=1, ncols=2, figsize=(12, 6))
>>> wfo = paos.WFO(beam_diameter, 1.0e-6 * wavelength, grid_size, zoom)
>>> wfo.aperture(xc=xdec, yc=ydec, r=beam_diameter/2, shape='circular')
>>> wfo.make_stop()
>>> ax0.imshow(wfo.amplitude)
>>> wfo.lens(lens_fl=1.0)
>>> wfo.propagate(dz=1.0)
>>> ax1.imshow(wfo.amplitude)
>>> plt.show()
```

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(*My*, *Mx=None*)
Given the optical magnification along one or both directions, updates the sampling along both directions, the beam semi-diameter, the Rayleigh distance, the distance to focus, and the beam focal ratio

Parameters
- **My** (*scalar*) – optical magnification along tangential direction
- **Mx** (*scalar*) – optical magnification along sagittal direction

Returns **out** – updates the wfo parameters
Return type None

ChangeMedium(*n1n2*)
Given the ratio of refractive indices n1/n2 for light propagating from a medium with refractive index n1, into a medium with refractive index n2, updates the Rayleigh distance, the wavelength, the distance to focus, and the beam focal ratio

Parameters **n1n2** –
Returns **out** – updates the wfo parameters
Return type None

ptp(*dz*)
Plane-to-plane (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

6.1.2 Subpackages

6.1.2.1 paos.gui package

Submodules

paos.gui.simpleGUI module

class SimpleGUI

Bases: object

Base class for the Graphical User Interface (GUI) for *PAOS*, built using the publicly available library PySimpleGUI

Initializes the GUI. This includes instantiating the global variables, defining the GUI theme and symbols to use, defining a quick message to display when opening the GUI that auto-closes itself and defining the GUI Menu (principal and right-click)

static add_heading(*headings*, *size=(24, 2)*)

Given a list of header names and a tuple for the size, returns a chained list of Text widgets with the specified size where the first element is returned to prettify spacing

Parameters

- **headings** (*list*) – a list of header names
- **size** (*tuple*) – a tuple for the widget sizes defined as (width, height)

Returns **out** – a chained list of Text widgets

Return type List[Text]

static collapse_frame(*title*, *layout*, *key*)

Helper function that creates a Frame that can be later made hidden, thus appearing "collapsed"

Parameters

- **title** (*str*) – the Frame title
- **layout** (*List[List[Element]]*) – the layout for the section
- **key** (*str*) – key used to make this section visible / invisible

Returns out – A pinned Frame that can be placed directly into your layout
Return type Frame

`static collapse_column(layout, key)`
Helper function that creates a Column that can be later made hidden, thus appearing "collapsed"

Parameters
- **layout** (*List[List[Element]]*) – the layout for the section
- **key** (*str*) – key used to make this section visible / invisible

Returns out – A pinned Column that can be placed directly into your layout
Return type Column

`static update_column_scrollbar(window, col_key)`
Given the current GUI window and the current Column key, updates the column scrollbar if the Column has changed

Parameters
- **window** (*Window*) – the current GUI window
- **col_key** (*str*) – the current Column key

Returns out – Updates the column scrollbar
Return type None

`static get_clipboard_text()`
Returns a copy of the local clipboard content (e.g. an Excel column)

Returns out – the local copy of the clipboard's content
Return type List[str]

`static to_configparser(dictionary)`
Given a dictionary, it converts it into a ConfigParser object

Parameters **dictionary** (*dict*) – input dictionary to be converted
Returns out
Return type ConfigParser

`static draw_figure(figure, canvas)`
CURRENTLY NOT USED Given a Canvas and a figure, it draws the figure onto the Canvas

Parameters
- **figure** (Figure) – the figure to be drawn
- **canvas** (*Canvas*) – the canvas onto which to draw the figure

Returns out – the Tkinter widget to draw a Figure onto a Canvas
Return type FigureCanvasTkAgg

`static draw_image(figure, element)`
Draws the previously created "figure" in the supplied Image Element

Parameters
- **figure** (Figure) – a Matplotlib figure
- **element** (*Image*) – an Image element

Returns out – The figure canvas
Return type Canvas

`static clear_image(element)`
Given an Image widget, it clears it

Parameters **element** (*Image*) – the Image widget to clear
Returns out – clears an Image widget
Return type None

static reset_progress_bar(*progress_bar*)
Given a progress bar element, it resets it

Parameters **progress_bar** (*ProgressBar*) – the progress bar element to reset
Returns **out** – resets the progress bar element
Return type ProgressBar

static move_with_arrow_keys(*window*, *event*, *values*, *elem_key*, *max_rows*, *max_cols*)
Given the current GUI window, the latest event, the dictionary containing the window values, the dictionary key for the cell with focus and the maximum sizes for the current table editor, this method sets the focus on the new cell (if the current cell changed).

Parameters
- **window** (*Window*) – the current GUI window
- **event** (*str*) – the latest GUI event
- **values** (*dict*) – the dictionary containing the window values
- **elem_key** (*str*) – the dictionary key for the cell with focus
- **max_rows** (*int*) – the number of rows for the current table editor
- **max_cols** (*int*) – the number of columns for the current table editor

Returns **out** – The row number corresponding to where the current focus is
Return type int

static copy_to_clipboard(*dictionary*)
Saves the relevant data from the GUI configuration tabs into a dictionary, then copies it to the local clipboard

Returns **out** – copies the data to the local clipboard
Return type None

static chain_widgets(*row*, *input_list*, *prefix*)
Given the row in the GUI editor, an input list and the key prefix, returns a list of widgets

Parameters
- **row** (*int*) – the current row in the GUI editor
- **input_list** (*List*) – the item list to insert in the cell widgets
- **prefix** (*str*) – the key prefix

Returns **out** – list of widgets
Return type List[Text, List[Checkbox or Input or Column or Combo]]

make_visible(*event*, *visible*, *key*)
Given the current event, a boolean and the widget key, it makes the widget visible/invisible

Parameters
- **event** (*str*) – the current event
- **visible** (*bool*) – if True, the widget becomes visible. If False, it becomes invisible
- **key** (*str*) – the widget key

Returns **out** – the updated visibile parameter of the widget
Return type bool

close_window()
Deletes the temporary .ini file (if present), any unsaved outputs and then closes the GUI window

Returns **out** – closes the GUI window and cleans up the memory
Return type None

`static sort_column(window, values, col_key)`

Given a GUI window, the window contents (values) and the Column key, it sorts the Column elements in increasing order

Parameters
- **window** (*Window*) – the GUI window
- **values** (*dict*) – the dictionary with the Window contents
- **col_key** (*str*) – the Column key

Returns out – Sorts the Column elements in ascending order
Return type None

paos.gui.zernikeGUI module

`class ZernikeGUI(config, values, row, key)`

Bases: *paos.gui.simpleGUI.SimpleGUI*

Generates the Zernike editor for the main *PAOS* GUI as a secondary GUI window

Parameters
- **config** (ConfigParser object) – the main GUI window parsed configuration file
- **values** (*dict*) – the main GUI window dict of values (returned when the Window element is read)
- **row** (*int*) – the Zernike surface row index in the lens data editor
- **key** (*str*) – the Zernike surface key in the configuration file

Initializes the GUI. This includes instantiating the global variables, defining the GUI theme and symbols to use, defining a quick message to display when opening the GUI that auto-closes itself and defining the GUI Menu (principal and right-click)

`add_row(row, dictionary, ordering)`

Parameters
- **row** (*int*) – the last row index in the Zernike tab
- **dictionary** (*dict*) – the dictionary with the Zindex and Z coefficients to update with the new Zernike table row
- **ordering** (*str*) – the Zernike coefficients ordering

Returns out – Adds a row in the Zernike tab and returns the updated number of rows and the azimuthal number, m, and radial number, n for the Zernike coefficients
Return type tuple(int, tuple(int, int))

`make_window()`

Generates the Zernike GUI window (secondary window to the main *PAOS* GUI window). The layout is composed of only one Tab, with four headers: 'Zindex', 'Z', 'm', 'n'. 'Zindex' contains the index j for each Zernike coefficient (starting from 0), 'Z' contains the Zernike coefficients, 'm' contains the azimuthal number and 'n' the radial number for the Zernike coefficients

Returns out – Generates the Zernike GUI window
Return type None

paos.gui.paosGUI module

`class PaosGUI(passvalue, theme='Dark Blue 3', font=('Courier New', 16))`

Bases: *paos.gui.simpleGUI.SimpleGUI*

Generates the Graphical User Interface (GUI) for *PAOS*, built using the publicly available library PySimpleGUI

Parameters

- **passvalue** (*dict*) – input dictionary for the GUI. It contains the file path to the input configuration .ini file to pass (defaults to using a template if unspecified), the debug and logger keywords and the file path to the output file to write.
- **theme** (*str*) – GUI theme. By default, it is set to the official PySimpleGUI theme
- **font** (*tuple*) – GUI default font. By default, it is set to ("Courier New", 16)

Example

```
>>> from paos.gui.paosGUI import PaosGUI
>>> passvalue = {'conf': '/path/to/ini/file/', 'debug': False}
>>> PaosGUI(passvalue=passvalue, theme='Dark')()
```

Note: The first implementation of PaosGUI is in *PAOS* v0.0.4

Initializes the Paos GUI. This includes instantiating the global variables, setting up the debug logger (optional), defining the content to display in the main GUI Tabs and a fallback configuration file if the user did not specify it

init_window()
Initializes the main GUI window by parsing the configuration file and initializing the input data dimensions

get_widget(*value*, *key*, *item*, *size=(24, 2)*)
Given the cell value, key and item, returns the widget of the desired type

Parameters

- **value** (*str or Iterable or dict*) – value or selection of values to put inside the widget
- **key** (*str*) – key to which the returned widget will be associated
- **item** (*bool or str*) – typically, the default value to show in the widget
- **size** (*tuple*) – a tuple for the widget sizes defined as (width, height)

Returns out – the desired Widget
Return type Checkbox or Input or Column or Combo

fill_aperture_tab(*row*, *col*)
Given the row and column corresponding to the cell in the GUI lens data editor, returns a list of widgets with the aperture parameters names and values

Parameters

- **row** (*int*) – row corresponding to the optical surface in the GUI lens data editor
- **col** (*int*) – column corresponding to the aperture header in the GUI lens data editor

Returns out – list of widgets with the aperture parameters names and values
Return type List[List[Text], List[Checkbox or Input or Column or Combo]]

aperture_tab(*row*, *col*, *key*, *disabled*)
Given the row and column corresponding to the cell in the GUI lens data editor, returns the Column widget with the aperture parameters

Parameters

- **row** (*int*) – row corresponding to the optical surface in the GUI lens data editor

- **col** (*int*) – column corresponding to the aperture header in the GUI lens data editor
- **key** (*str*) – key to which the returned Column widget will be associated
- **disabled** (*bool*) – boolean to disable or enable events for the Button widget

Returns out – the Column widget for the aperture
Return type Column

static par_heading_rules(*surface_type*)
Given the optical surface type, applies pre-defined rules to return the desired list of headers

Parameters surface_type (*str*) – the surface type (e.g. Standard)
Returns out – the desired list of headers
Return type List[str] or None

update_headings(*row*)
Updates the displayed headers according to the rules set in *par_heading_rules*

Parameters row (*int*) – row corresponding to the optical surface in the GUI lens data editor
Returns out – Updates the headers
Return type None

static lens_data_rules(*surface_type*, *header*, *item=None*)
Given the optical surface type, applies pre-defined rules to return the desired item for the widget

Parameters
- **surface_type** (*str*) – the surface type (e.g. Standard)
- **header** (*str*) – the column header from the lens data editor
- **item** (*bool or str*) – the item to put in the cell widget

Returns out – the item to put in the cell widget
Return type bool or str

chain_widgets(*row*, *input_list*, *prefix*)
Given the row in the GUI lens data editor, an input list and a prefix, returns a list of widgets to fill a GUI editor data row

Parameters
- **row** (*int*) – row corresponding to the optical surface in the GUI lens data editor
- **input_list** (*list*) – items list with which to fill the new row
- **prefix** (*str*) – prefix to indicate which kind of widgets list must be returned

Returns out – list of widgets to fill a GUI editor data row
Return type List[Text, List[Checkbox or Input or Column or Combo]]

add_row(*column_key*)
Given the Column key, it updates the current Column by adding a row. For example, can add a row to the wavelength Column.

Parameters column_key (*str*) – key for the Column widget to which we want to add a row
Returns out – the current Column's updated number of rows
Return type int

save_to_dict(*show=True*)
Saves the data content from the GUI input Tabs (General, Fields, Lens Data) to a dictionary

Parameters **show** (*bool*) – If True, displays a popup window with the output dictionary that can be copied to the local clipboard. If False, returns the output dictionary.
Returns **out** – the GUI input content as a dictionary
Return type dict or None

to_ini(*filename=None*, *temporary=False*)
Given the output .ini file path, it writes the data content from the GUI input Tabs to it

Parameters
- **filename** (*str or None*) – if not given, writes to the current .ini configuration file
- **temporary** (*bool*) – if True, the .ini file is temporary and is deleted upon exiting the GUI

Returns **out** – writes the data content from the GUI input Tabs to a .ini file
Return type None

static plot_surface(*key*, *retval*, *ima_scale*)
Given the optical surface key, the POP output dictionary and the image scale, plots the squared amplitude of the wavefront at the given surface (cross-sections and 2D plot)

Parameters
- **key** (*int*) – the key index associated to the optical surface
- **retval** (*dict*) – the POP output dictionary
- **ima_scale** (*str*) – the image scale. Can be either 'linear' or 'log'

Returns **out** – the figure with the squared amplitude of the wavefront at the given surface
Return type Figure

draw_surface(*retval_list*, *groups*, *figure_agg*, *image_key*, *surface_key*, *scale_key*, *range_key=None*)
Given a list of simulation output dictionaries, the names to associate to each simulation, the Figure element to draw on and its key, the key for the surface to plot and the plot scale, plots at the given optical surface and returns the figure(s) produced

Parameters
- **retval_list** (*List of dict*) – list of simulation output dictionaries
- **groups** (*List of str*) – names to associate to each simulation in the output
- **figure_agg** (*Figure*) – element Figure to draw on
- **image_key** (*str*) – Figure element key
- **surface_key** (*str*) – key associated to the surface to plot
- **scale_key** (*str*) – plot scale. Can be either 'linear' or 'log'
- **range_key** (*str*) – range of simulations to plot. Examples: 0) '0,1' means only the first simulation is plotted 1) '0,10' means that the first ten simulations are plotted If the number of simulations to plot is greater than len(retval_list), plots all the simulations and nothing bad happens

Returns **out** – or (List[Figure], List[int]) Plots at the given optical surface and returns the figure(s)
Return type None or Figure or List[Figure]

Notes

Do not plot too many figures (> 20) at a time, otherwise they may occupy too much memory, and you will get a warning from matplotlib for reached maximum number of opened figures

`display_plot_slide`(*figure_list*, *figure_agg*, *image_key*, *slider_key*)

Given a list of figures, a figure canvas, the image key and the slider key, returns the updated Image element

Parameters
- **figure_list** (List of Figure) – a list of matplotlib figures
- **figure_agg** (*Canvas*) – the figure Canvas
- **image_key** (*str*) – the Image key
- **slider_key** – the Slider key

Returns **out** – the updated Image element
Return type None or Canvas

`static save_figure`(*figure*, *filename=None*)

Given a matplotlib figure and a file path (optional), saves the figure to the file path

Parameters
- **figure** (Figure) – the matplotlib figure to save
- **filename** (*str*) – the saving file path

Returns **out** – Saves the matplotlib figure
Return type None

`static to_hdf5`(*retval_list*, *groups*, *keys_to_keep=None*)

Given the POP output dictionary list, the names for each simulation (saving groups) and the keys to store, opens a popup to get the output file name and then dumps the simulation outputs to a hdf5 file.

Parameters
- **retval_list** (*List of dict*) – list of simulation output dictionaries
- **groups** (*List of strings*) – names to associate to each simulation in the output
- **keys_to_keep** (*List of strings*) – dictionary keys to store

Returns **out** – Dumps the simulation output to the indicated hdf5 file
Return type None

`static to_txt`(*text_list*)

Given a list of strings, opens a popup to get the output file name and then dumps the text ordered in rows to the output text file

Parameters **text_list** (*list of strings*) –
Returns **out** – writes the input text list to a text file. Used to save the output of the diagnostic raytrace
Return type None

`make_window`()

Generates the main GUI window. The layout is composed of 5 Tabs, the first one for the general input parameters and the wavelengths, the second one for the input fields, the third one for the optical data surfaces, the fourth one for the simulation launcher (diagnostic raytrace, POP and plots) and the last one for the general info on the GUI and its developers. Defines also some pretty cursors and binds the method to update the headings

Returns **out** – Generates the main GUI window
Return type None

Module contents

6.1.2.2 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.

6.1.2.3 paos.util package

Submodules

paos.util.material module

class Material(*wl*, *Tambient=- 218.0*, *Pambient=1.0*, *materials=None*)
Bases: object

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

Parameters
- **Tambient** (*scalar*) – Ambient temperature during operation ($^{\circ}C$)
- **Pambient** (*scalar*) – Ambient pressure during operation (atm)
- **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 $T_{ref} = 20^{\circ}C$ and pressure $P_{ref} = 1\ atm$.

The Sellmeier 1 equation consists of three terms and is given as $n^2(\lambda) = 1 + \frac{K_1\lambda^2}{\lambda^2 - L_1} + \frac{K_2\lambda^2}{\lambda^2 - L_2} + \frac{K_3\lambda^2}{\lambda^2 - L_3}$

Parameters **par** (*dict*) – dictionary containing the K_1, L_1, K_2, L_2, K_3, L_3 parameters of the Sellmeier 1 model
Returns **out** – the refractive index
Return type scalar or array (same shape as wl)

static nT(*n*, *D0*, *delta_T*)
Estimate the change in the glass absolute index of refraction with temperature as

$n(\Delta T) = \Delta n_{abs} + n$

where

$\Delta n_{abs} = \frac{n^2-1}{2n} D_0 \Delta T$

Parameters
- **n** (*scalar or array*) – relative index at the reference temperature of the glass
- **D0** (*scalar*) – model parameter (constant provided by the glass manufacturer to describe the glass thermal behaviour)

- **delta_T** (*scalar*) – change in temperature relative to the reference temperature of the glass. It is positive if the temperature is greater than the reference temperature of the glass

Returns out – the scaled relative index
Return type scalar or array (same shape as n)

nair(*T*, *P=1.0*)

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

$n_{air} = 1 + \frac{(n_{ref}-1)P}{1.0+(T-15)\cdot(3.4785\times10^{-3})}$

where

$n_{ref} = 1 + \left[6432.8 + \frac{2949810\lambda^2}{146\lambda^2-1} + \frac{25540\lambda^2}{41\lambda^2-1}\right]\cdot 10^{-8}$

This formula for the index of air is from F. Kohlrausch, Praktische Physik, 1968, Vol 1, page 408.

Parameters
- **T** (*scalar*) – temperature in $^{\circ}C$
- **P** (*scalar*) – relative pressure in atmospheres (dimensionless in the formula). Defaults to 1 atm.

Returns out – the air index of refraction
Return type scalar or array (same shape as wl)

Note:

1) Air at the system temperature and pressure is defined to be 1.0, all other indices are relative
2) *PAOS* can easily model systems used in a vacuum by changing the air pressure to zero

---

nmat(*name*)

Given the name of an optical glass, returns the index of refraction in vacuum as a function of wavelength.

Parameters **name** (*str*) – name of the optical glass
Returns out – returns two arrays for the glass index of refraction at the given wavelengths: the index of refraction at $T_{ref} = 20^{\circ}C$ (nmat0) and the index of refraction at T_{amb} (nmat)
Return type tuple(scalar, scalar) or tuple(array, array)

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 out – displays the plot output or stores it to the indicated plot path
Return type None

Examples

```
>>> from paos.util.material import Material
>>> Material(wl = np.linspace(1.8, 8.0, 1024)).plot_relative_index(material_
↪list=['Caf2', 'Sf11', 'Sapphire'])
```

Module contents

Chapter 7

License

BSD 3-Clause License

Copyright (c) 2020, arielmission-space. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Chapter 8

Acknowledgements

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

- Andrea Bocchieri
- Enzo Pascale

Python Module Index

p

Index

N

P

R

S

T

U

W

Z