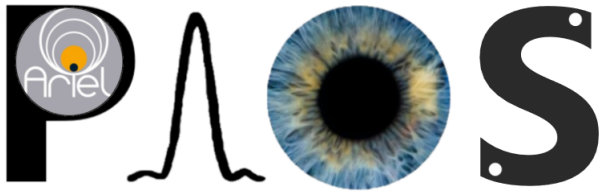
PAOS Manual v1.0.0



Andrea Bocchieri, Lorenzo V. Mugnai, Enzo Pascale

Created on: December, 2021

Last updated: 11th March, 2024

Table of contents

List of figures

List of tables

PAOS is a reliable, user-friendly, and open-source Physical Optics Propagation code that integrates an implementation of Fourier optics. It employs the Fresnel approximation for efficient and accurate optical system simulations.

By including a flexible configuration file and paraxial ray-tracing, PAOS seamlessly facilitates the study of various optical systems, including non-axial symmetric ones, as long as the Fresnel approximation remains valid.

This guide will walk you through the PAOS code with examples and descriptions of the main algorithms.

Warning: This documentation is not completed yet. If you find any issue or difficulty, please contact the developers for help.

Important: A dedicated paper has been submitted and the related information will be published soon.

Caution: In case of inconsistency between the documentation and the paper, always assume that the paper is correct.

Hint: 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.

Want to install it? Head here: *Installation*

Want to jump into the PAOS program? Head here: *User guide*

Want to know more about the code? Head here: *API guide*

Want to collaborate? Head here: *Developer guide*

Curious about the license? Head here: *License*

Curious about the project history? Head here: *Changelog*

Chapter 1

User guide

This guide covers the general installation, as well as the use of PAOS as a standalone program and as a library.

1.1 Introduction

Accurate assessment of the optical performance of advanced telescopes and imaging systems for astrophysical applications is essential to achieve an optimal balance between optical quality, system complexity, costs, and risks.

PAOS is an open-source code implementing physical optics propagation (POP) in Fresnel approximation and paraxial ray tracing to analyze complex waveform propagation through both generic and off-axes optical systems (see *ABCD description* and *POP description*), enabling the generation of realistic Point Spread Functions across various wavelengths and focal planes.

Developed using a Python 3 stack, PAOS includes an installer, documented examples, and **this** comprehensive guide. It improves upon other POP codes offering extensive customization options and the liberty to access, utilize, and adapt the software library to the user's application.

With a generic input system and a built-in Graphical User Interface (see *Input system*), PAOS ensures seamless user interaction and facilitates simulations.

The versatility of PAOS enables its application to a wide array of optical systems, extending beyond its initial use case. PAOS presents a fast, modern, and reliable POP simulation tool for the scientific community, enhancing the assessment of optical performance in various optical systems and making advanced simulations more accessible and user-friendly.

1.2 Installation

1.2.1 Install with pip

The PAOS package is hosted on PyPI repository. You can install it by

```
pip install paos
```

1.2.2 Install from git

You can clone PAOS from our main git repository

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

Move into the PAOS folder

```
cd /your_path/PAOS
```

Then, just do

```
pip install .
```

To test for correct setup you can do

```
python -c "import paos"
```

If no errors appeared then it was successfully installed.

Additionally the PAOS program should now be available in the command line

```
paos
```

and the PAOS GUI (see *GUI editor*) can be accessed calling

```
paosgui
```

1.2.3 Uninstall PAOS

PAOS is installed in your system as a standard python package: you can uninstall it from your Environment as

```
pip uninstall paos
```

1.2.4 Update PAOS

If you have installed PAOS using Pip, now you can update the package simply as

```
pip install paos --upgrade
```

If you have installed PAOS from GitHub, you can download or pull a newer version of PAOS over the old one, replacing all modified data.

Then you have to place yourself inside the installation directory with the console

```
cd /your_path/PAOS
```

Now you can update PAOS simply as

```
pip install . --upgrade
```

or simply

```
pip install .
```

1.2.5 Modify PAOS

You can modify PAOS main code, editing as you prefer, but in order to make the changes effective

```
pip install . --upgrade
```

or simply

```
pip install .
```

To produce new PAOS functionalities and contribute to the code, please see *Developer guide*.

1.3 Quick start

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

1.3.1 Running PAOS from terminal

The quickest way to run PAOS is from terminal.

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

```
$ paos --help
```

The main command line flags are listed in Table 1.1.

Table 1.1 – 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, --logger	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 1.2.

Table 1.2 – Other option flags

flag	description
`-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 1.3.

Table 1.3 – 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.

1.3.2 The output file

`PAOS` stores its main output product to a HDF5 file (extension is *.h5* or *.hdf5*) such as that shown in Fig. 1.1. To open it, please choose your favourite viewer (e.g. HDFView, HDFCompass) or API (e.g. Cpp, FORTRAN and Python).

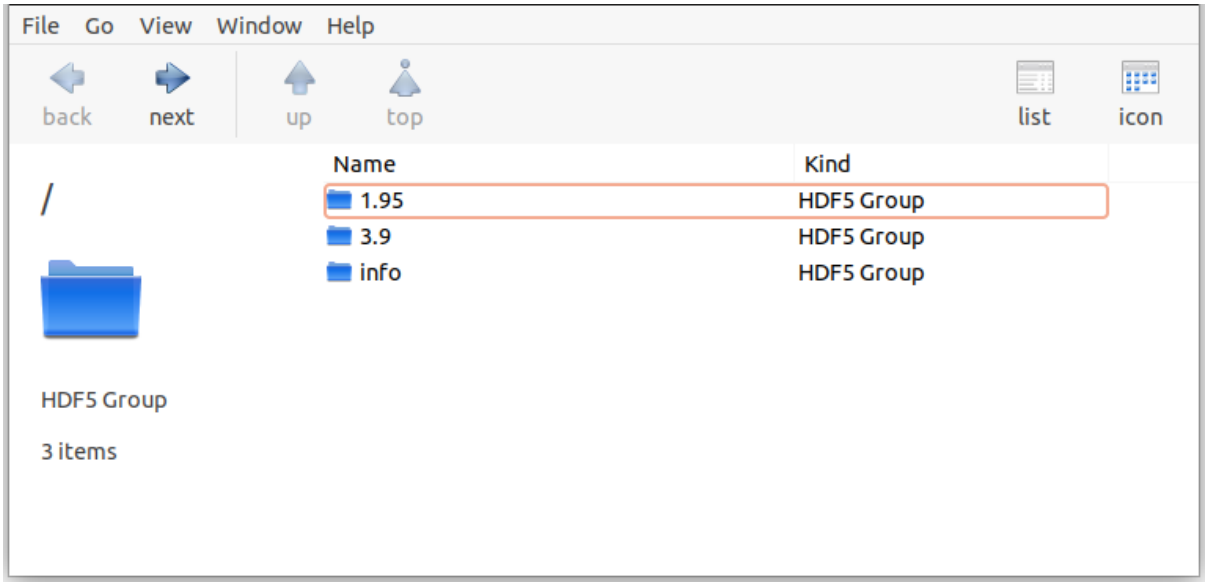


Fig. 1.1 – *Main PAOS output file*

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

1.3.3 The baseline plot

As part of the output, `PAOS` can plot the squared amplitude of the complex wavefront at a given point along the optical path (the focal plane in the case shown in Fig. 1.2).

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.

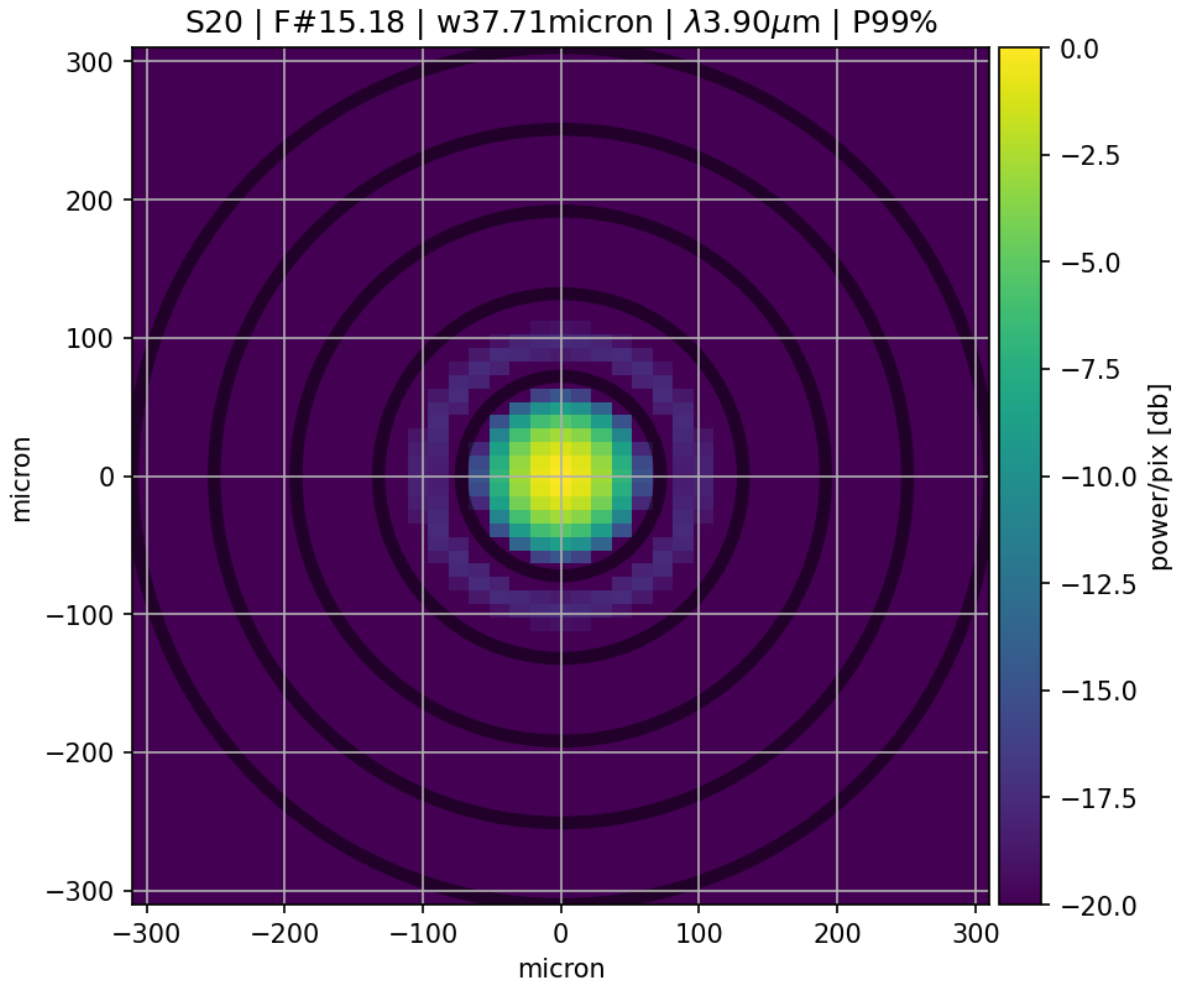


Fig. 1.2 – *Baseline PAOS plot*

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

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

1.4 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 ©;
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 automatized 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 Ariel, PAOS is currently used to simulate the optical performance of the stratospheric balloon-borne experiment EXCITE.

   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.

1.4.1 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

1.4.1.1 General

Section describing the general simulation parameters and PAOS units

Table 1.4 – 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. 1.3 – *General*

1.4.1.2 Wavelengths

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

Table 1.5 – 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

1.4.1.3 Fields

Section listing the input fields to simulate

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

Fig. 1.4 – *Wavelengths*

Table 1.6 – 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. 1.5 – *Fields*

1.4.1.4 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 1.7 – 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
Standard	...	float	float	MIRROR, others	Bool	Bool	Bool	list	None
Paraxial Lens	...	None	float	None	Bool	Bool	Bool	list	Par1 = focal length (float)

continues on next page

Table 1.7 – continued from previous page

Surface-Type	Comment	Radius	Thickness	Material	Save	Ignore	Stop	aperture	Par1..N
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 = wavelength (in micron) Par2 = ordering, can be standard, ansi, noll, fringe Par3 = Normalisation, can be True or False Par4 = Radius of support aperture of the polynomial 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. 1.6 – *Lens_xx*

1.4.1.5 Parse configuration file

PAOS implements the method *parse_config* that parses the .ini configuration 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.core.parseConfig import parse_config
pup_diameter, parameters, wavelengths, fields, opt_chains = parse_config('path/to/ini/
↪file')
```

1.4.2 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. To achieve this, PAOS uses the PySimpleGui package, a Python package that aims at "bridging the GUI gap between software developers and end users".

The quickest way to run the PAOS GUI is from terminal.

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

```
$ paosgui --help
```

Table 1.8 – 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, --logger`	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

1.4.2.1 General Tab

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

It contains two Frames:

- `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.
- `Wavelength Setup`

Lists the wavelengths to simulate. This list can be altered by editing the wavelengths. The user can use the Buttons in the `Wavelengths Actions` Frame to modify the list content by adding new wavelength rows, pasting a list of wavelengths from the local clipboard (*comma*-separated or \n-separated) and can also be sort the list to increasing order.

Below we report a snapshot of this Tab.

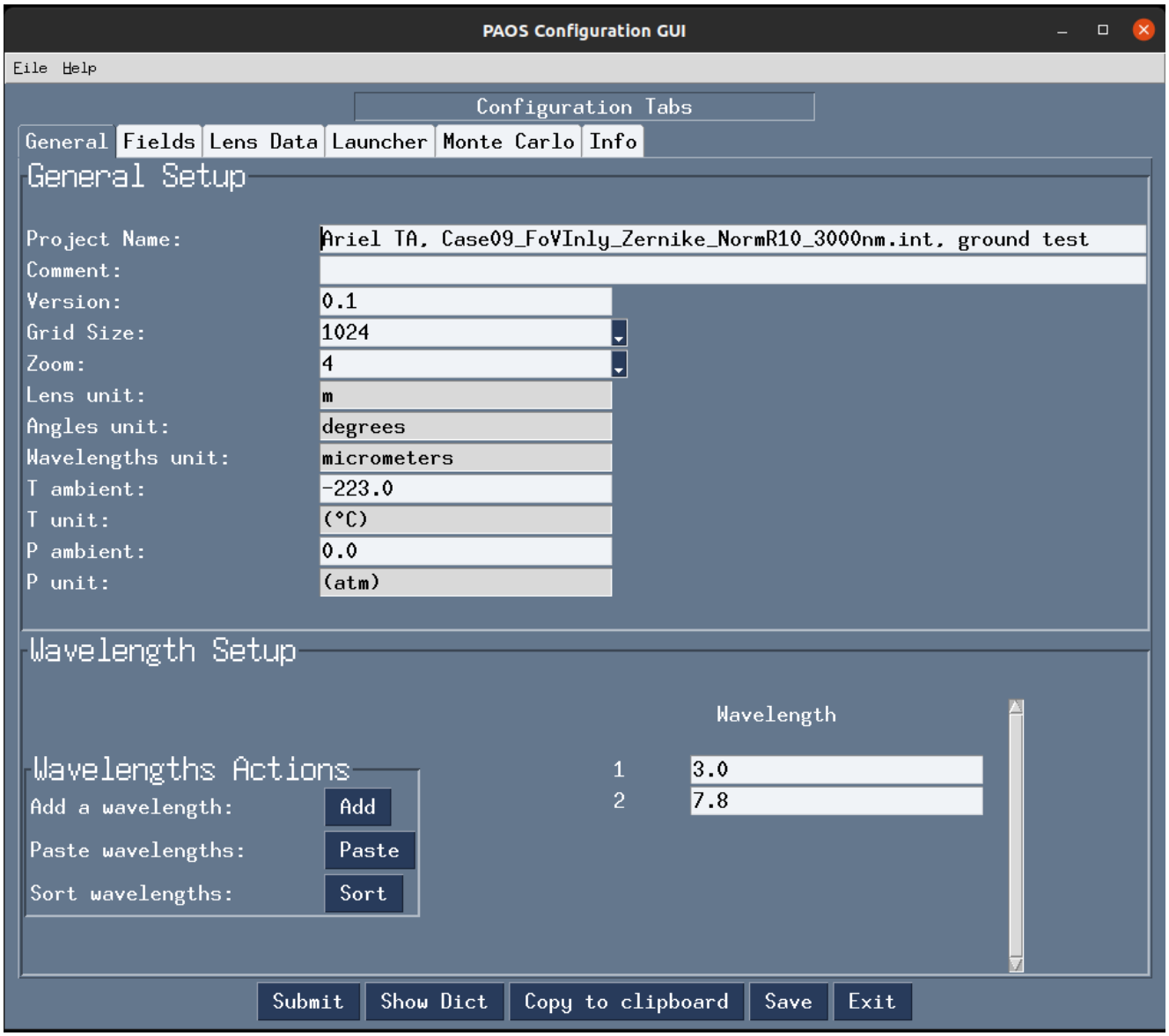


Fig. 1.7 – *General Tab*

1.4.2.2 Fields Tab

This GUI Tab describes the input fields to simulate.

In the `Fields Setup` Frame it lists the input fields, as defined in *Fields*.

The fields contents can be edited as necessary and new fields can be added by clicking on the *Add Field* Button in the `Fields Actions` Frame.

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.

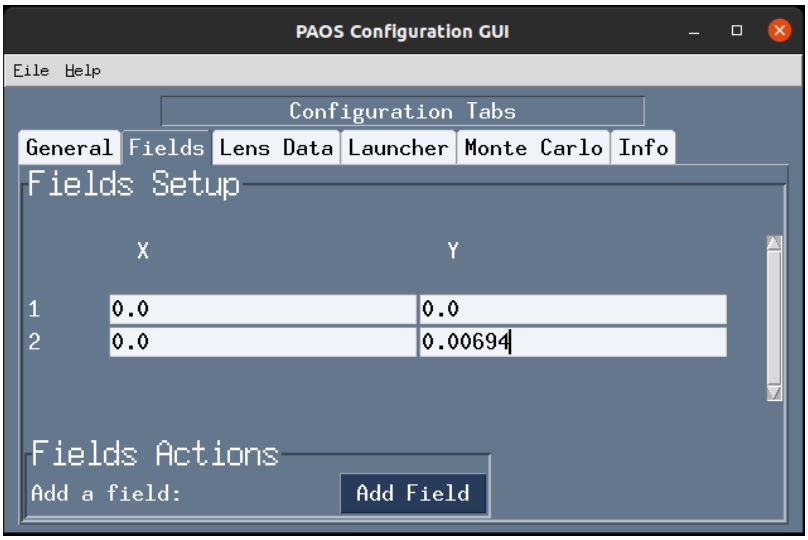


Fig. 1.8 – *Fields Tab*

1.4.2.3 Lens data Tab

This GUI Tab contains the list of the optical surfaces describing the optical chain to simulate, as defined in *Lens_xx*.

This information is organized in the `Lens Data Setup` Frame, whose structure tries to mimic that of Zemax OpticStudio ©. The columns are arranged as explained in *Lens_xx*, with horizontal and vertical scrollbars to allow any movement.

The contents of each row can be edited as necessary and new surfaces can be added by clicking on the *Add Surface* Button in the `Lens Data Actions` Frame.

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

Below we report a snapshot of this Tab.

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.

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

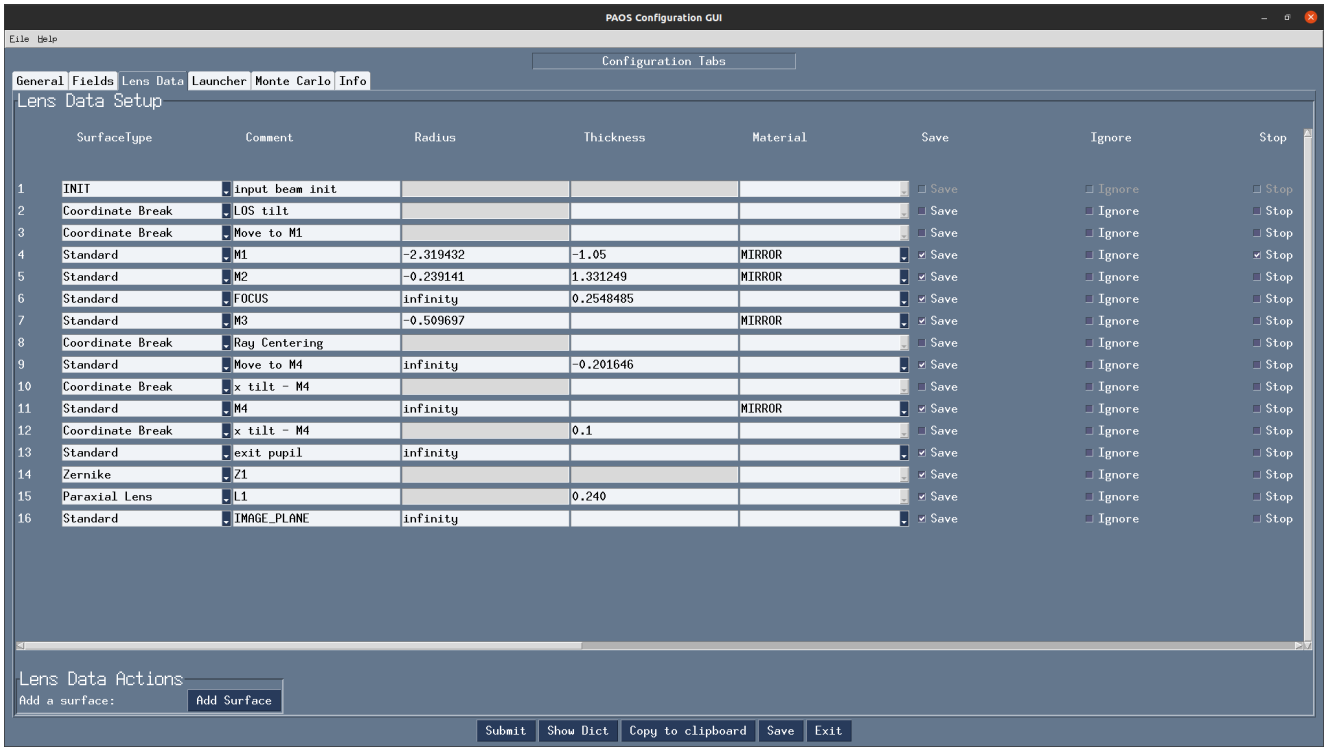


Fig. 1.9 – *Lens data Tab*

1.4.2.4 Zernike Tab

This GUI Tab can be accessed from the Lens Data Tab, by selecting a `Zernike` surface in the Dropdown menu from the `SurfaceType` column. Then, a small window appears asking to proceed with the insertion or modification of Zernike coefficients. A positive answer opens the Zernike Tab.

It contains two Frames:

- `Parameters`
  Displays the Zernike parameters as defined in the Lens Data Tab and serves as a reminder to the user. It is not enabled to be modified, which needs to be done beforehand in the Lens Data Tab.
- `Zernike Setup`
  Contains a Table that lists the Zernike polynomial index ("Zindex"), the Zernike coefficients ("Z"), and the azimuthal ("m") and radial ("n") polynomial orders, according to the specified Zernike ordering (one of *standard*, *ansi*, *fringe* and *noll*). Only the "Z" column is enabled to be modified as required by the user.
  The user can use the Buttons in the `Zernike Actions` Frame to modify the Table content by adding new rows, completing an unclosed Zernike radial order or adding a new one (available only if using *standard* or *ansi* ordering), and by pasting a list of Zernike coefficients from the local clipboard (*comma*-separated or \n-separated) in a cell from the "Z" column to automatically create and fill all necessary rows. The other columns will update accordingly.

Below we report a snapshot of this Tab.

1.4.2.5 Launcher Tab

This GUI Tab is designed to make preliminary, fast simulations to test a new configuration file or to simulate the propagation for a particular wavelength at a time.

It contains three Frames:

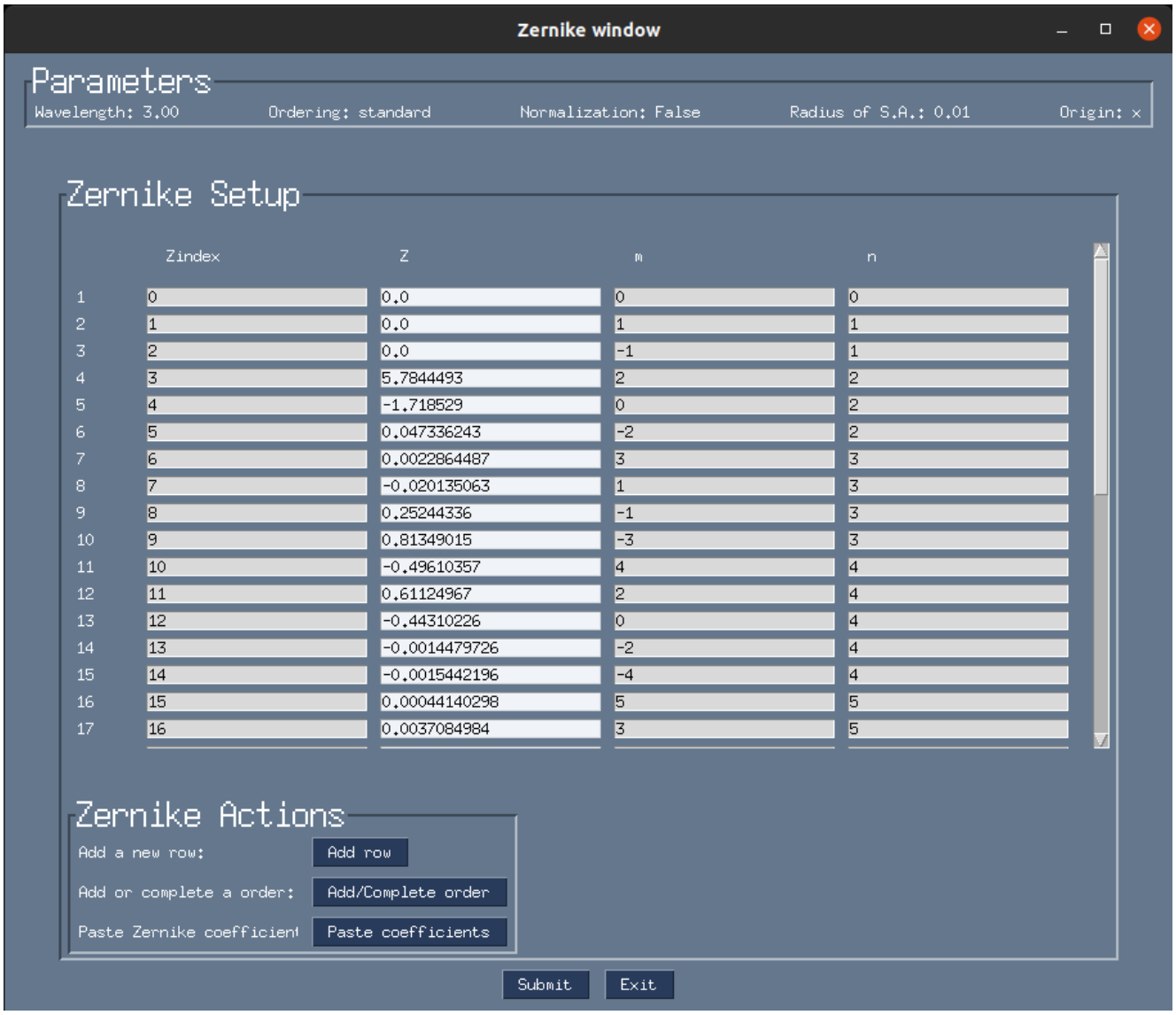


Fig. 1.10 – *Zernike Tab*

- `Select inputs`
  Allows to select the simulation wavelength and field. By selecting a new wavelength or field, the outputs of this Tab are reset, except for the raytrace output if the field has not changed.
- `Run and Save`
  Contains Buttons to call `PAOS` methods to run the simulation.
  The *Raytrace* Button runs a diagnostic ray-trace of the optical system, producing an output that is displayed in the Multiline element below it. This output can be saved to a text file by using the `Save raytrace` Button.
  The `POP` Button runs the wavefront propagation, producing an output dictionary that can be saved to a binary (.hdf5) file using the `Save POP` Button.
  The `Plot` Button plots the squared amplitude of the wavefront with the selected zoom factor at the selected surface from the Dropdown menu. The plot scale can be selected to be *logarithmic* or *linear*. Use the `Save Plot` Button to save the produced plot.
- `Display`
  Allows to see the simulation output plot. To display it, use the `Display plot` Button.

Below we report a snapshot of this Tab.

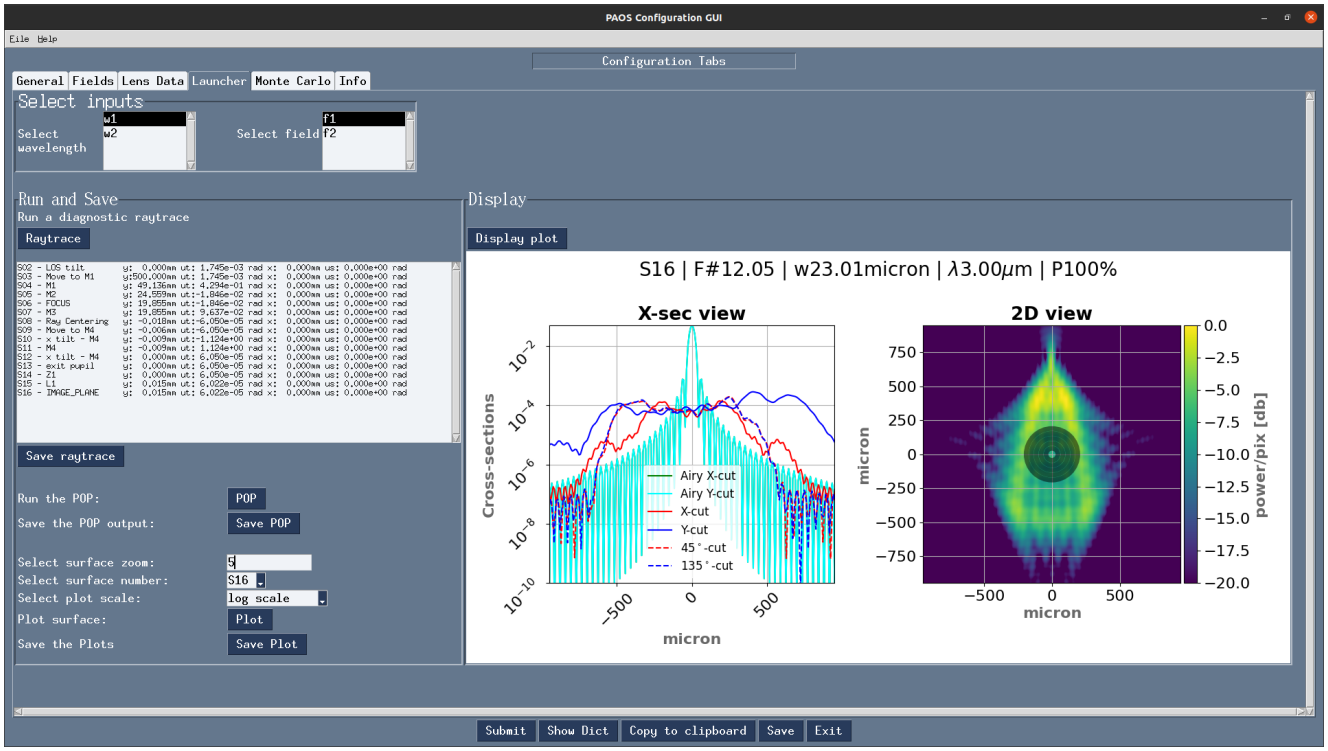


Fig. 1.11 – *Launcher Tab*

1.4.2.6 Monte Carlo Tab

This GUI Tab is designed to provide support for specific *Monte Carlo* simulations.

Two kinds of such simulations are currently supported:

1. Running the optical system at all provided wavelengths at once.
2. Running the optical system with different aberration realizations.

Therefore, the Tab contains two (collapsible) Frames, each with a layout similar to *Launcher Tab*:

- `MC Wavelengths`

Provides GUI support for running all provided wavelengths using parallel execution.
The user can select a field in the `Select Inputs` Frame, a number of parallel jobs, and then run the propagation by clicking on the `POP` Button. The simulation output can then be saved to a binary (.hdf5) file using the `Save POP` Button.
The `Plot` Button plots the squared amplitude of the wavefront for the selected range of simulations, which is automatically estimated from the simulation output but can be customized as needed. The plots can be customized by selecting the zoom factor, the surface to plot and the plot scale. Use the `Save Plot` Button to save the produced plots. To uniquely label the plots to be saved, please change the default figure prefix.
To display the plots, use the `Display plot` Button and the Slider element to see all plotted instances.
Below we report a snapshot of this Frame.

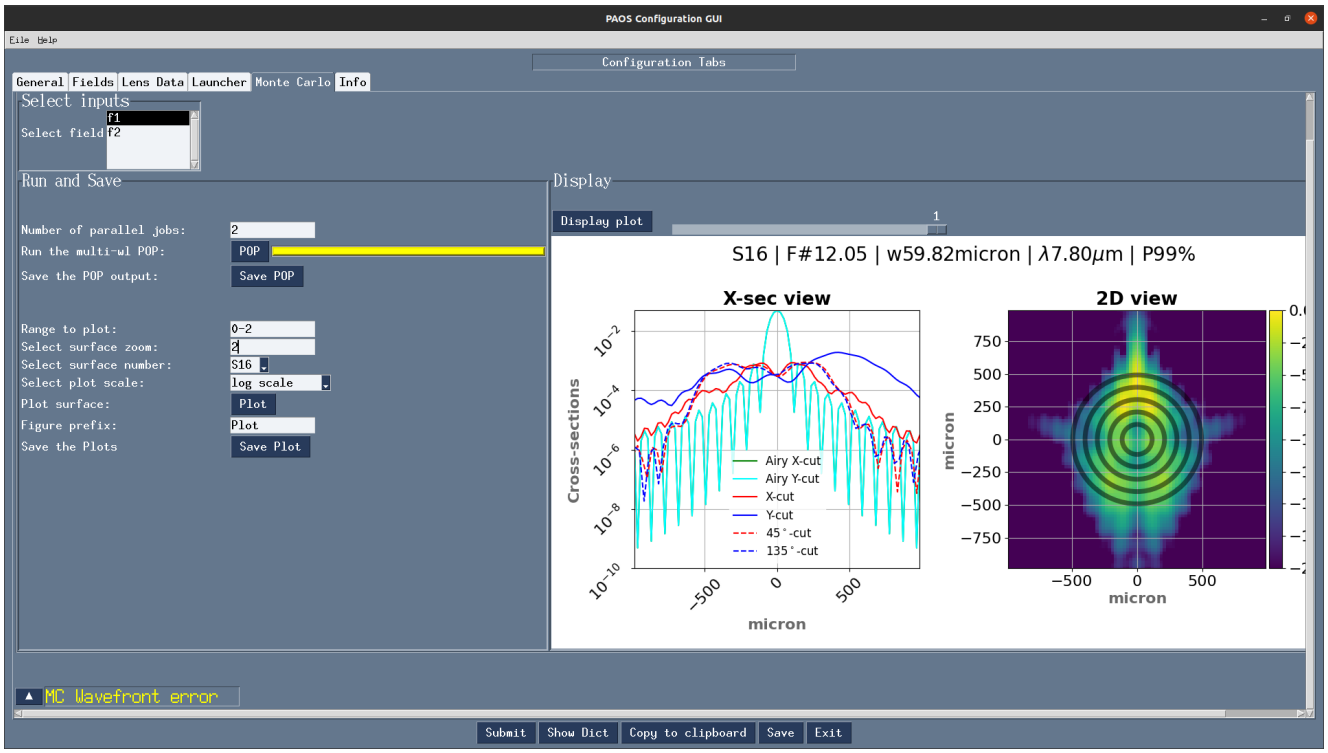


Fig. 1.12 – *Monte Carlo Tab (1)*

- `MC Wavelengths`
  Provides GUI support for running the propagation with different aberration realizations using parallel execution.
  The user can select the wavelength and field in the `Select Inputs` Frame.
  The .csv file with the aberration realizations can be imported using the `Import wfe` Button. To indicate the unit of the Zernike coefficients (r.m.s.), use the Dropdown menu below it.
  After this, select the number of parallel jobs, indicate the index of the Zernike surface (the corresponding row in the *Lens data Tab* and run the propagation using the `POP` Button. The simulation output can then be saved to a binary (.hdf5) file using the `Save POP` Button.
  To plot, save and display the simulation output, please refer to the preceding paragraph `MC Wavelengths`.
  Below we report a snapshot of this Frame.

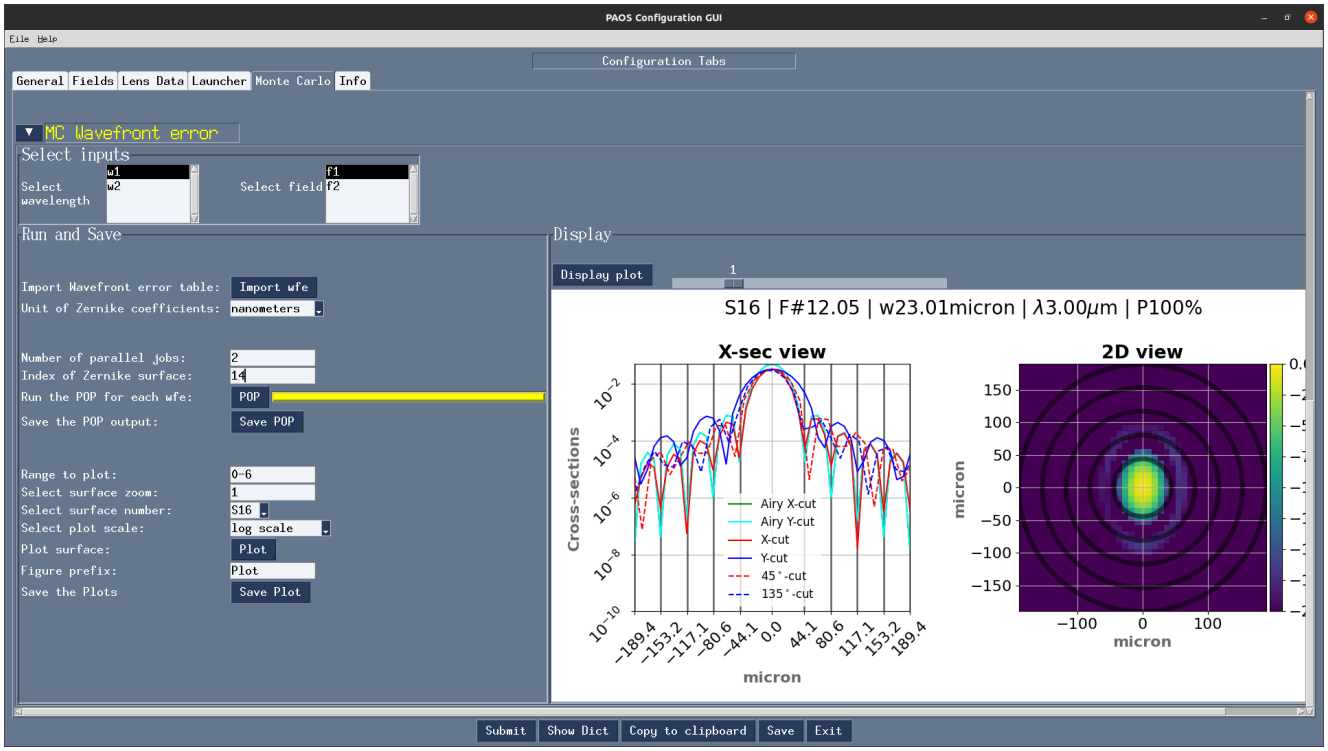


Fig. 1.13 – *Monte Carlo Tab (2)*

1.4.2.7 Info Tab

This GUI Tab contains information about the `PAOS` creators and the GUI.

It displays:

- The author names
- The `PAOS` version
- The Github repository
- The PySimpleGui version and release

Below we report a snapshot of this Tab.

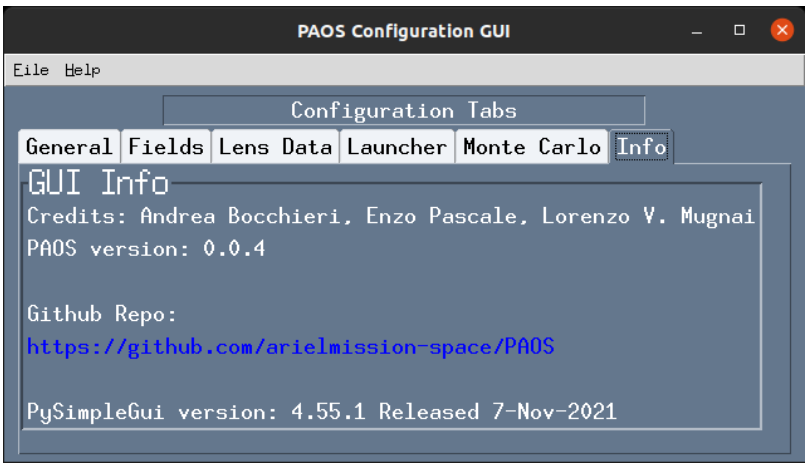


Fig. 1.14 – *Info Tab*

1.5 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 classes *ABCD* and WFO (see *POP description*).

1.5.1 Paraxial region

Following e.g. 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.

1.5.2 Optical coordinates

The PAOS code implementation assumes optical coordinates as defined in Fig. 1.15.

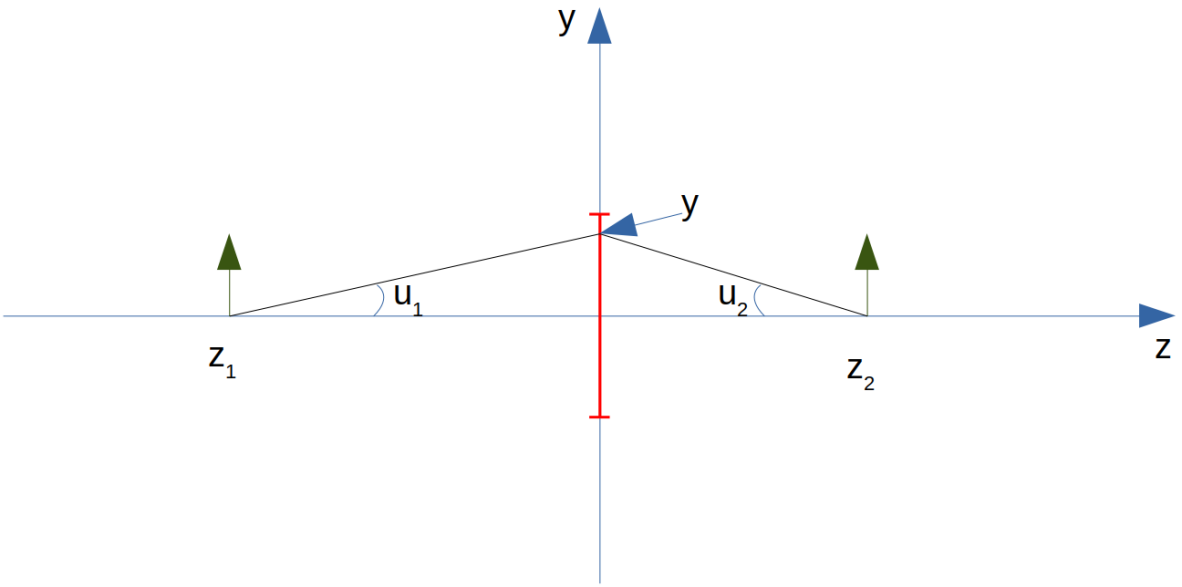


Fig. 1.15 – *Optical coordinates definition*

where

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

The (thin) lens equation is

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

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

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

1.5.3 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*).

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)

1.5.3.1 Example

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

```
from paos.core.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',
 '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']
```

1.5.4 Propagation

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

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

1.5.4.1 Example

Code example to initialize *ABCD* to propagate a light ray over a thickness $t = 50.0$ mm.

```
from paos.classes.abcd import ABCD
thickness = 50.0  # mm
abcd = ABCD(thickness=thickness)
print(abcd.ABCD)
```

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

1.5.5 Thin lens

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

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

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

1.5.5.1 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.classes.abcd import ABCD
radius = 20.0  # mm
abcd = ABCD(curvature=1.0/radius)
print(abcd.ABCD)
```

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

1.5.6 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

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

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

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

1.5.6.1 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.classes.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 ]]
```

1.5.7 Medium change

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

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

1.5.7.1 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.classes.abcd import ABCD
n1, n2 = 1.0, 1.25
abcd = ABCD(n1 = n1, n2 = n2)
print(abcd.ABCD)
```

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

1.5.8 Thick lens

A real (thick) lens is modelled as

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

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

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

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

1.5.8.1 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.classes.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.

1.5.9 Magnification

A magnification is modelled as

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

1.5.9.1 Example

Code example to initialize *ABCD* to simulate the effect of a magnification $M = 2.0$ on a light ray.

```
from paos.classes.abcd import ABCD
abcd = ABCD(M=2.0)
print(abcd.ABCD)
```

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

1.5.10 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:

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

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

where n is the refractive index of the prism, L is the geometrical path length of the prism, and the angles θ_i are as described in Fig.2 from the article, reproduced in Fig. 1.16.

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

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

where

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

1.5.10.1 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. 1.16.

```
import numpy as np
from paos.classes.abcd import ABCD

thickness = 2.0e-3  # m
n = 1.5

theta_1 = np.deg2rad(60.0)
```

(continues on next page)

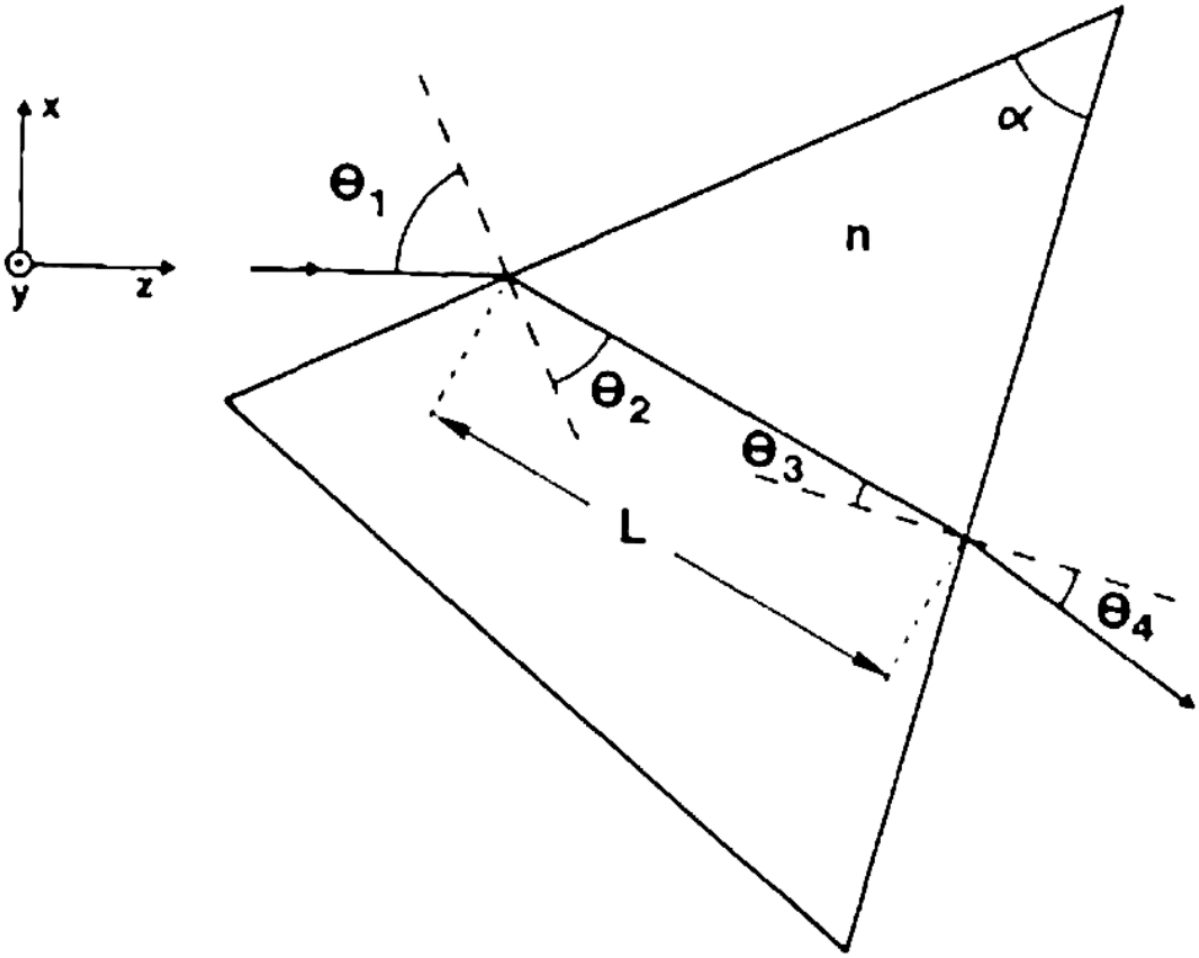


Fig. 1.16 – *Ray propagation through a prism*

(continued from previous page)

```
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 = thickness * np.cos(theta_1) * np.cos(theta_4) / (np.cos(theta_2) * np.cos(theta_
↪3)) / n
C = 0.0
D = 1.0 / A

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

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

```
[[1.59626666e+00 7.09451848e-04]
 [0.00000000e+00 6.26461747e-01]]
[[1.         0.00133333]
 [0.         1.        ]]
```

1.5.11 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)

`PAOS` adopts the following factorization:

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

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

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

The parameters are estimated as follows:

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

With these definitions, the effective focal length is

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

1.5.11.1 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.classes.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 ]]
```

1.5.12 Thick lens equivalent

A thick lens can be implemented as two spherical surfaces separated by some distance, and a medium change.

The ABCD matrix is

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ -\Phi_2/n_1 & n_2/n_1 \end{pmatrix} \begin{pmatrix} 1 & L \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ -\Phi_1/n_2 & n_1/n_2 \end{pmatrix} = \begin{pmatrix} 1 - L\Phi_1/n_2 & Ln_1/n_2 \\ L\frac{\Phi_1\Phi_2}{n_1 n_2} - \frac{1}{n_1}(\Phi_1 + \Phi_2) & n_1/n_2 \end{pmatrix} \tag{1.15}$$

This is equivalent to two thin lenses separated by some distance, described by the ABCD matrix

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ -1/f_2 & 1 \end{pmatrix} \begin{pmatrix} 1 & Ln_1/n_2 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ -1/f_1 & 1 \end{pmatrix} \tag{1.16}$$

where

$$\begin{aligned} \frac{1}{f_1} &= \Phi_1 = \frac{n_2 - n_1}{R_1} \\ \frac{1}{f_2} &= \Phi_2 = \frac{n_1 - n_2}{R_2} \end{aligned} \tag{1.17}$$

The curvature radii, R_1 and R_2, follow the usual sign convention: positive if the centre lies in the image space, and negative if it lies in the object space.

1.6 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`*.

1.6.1 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.

1.6.2 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.

1.6.3 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

$$\vec{R_0} = (x - x_{dec}, y - y_{dec}, 0) \tag{1.18}$$

and the unit vector of the light ray

$$\vec{n_0} = (zu_x, zu_y, z) \tag{1.19}$$

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

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

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

$$U^T \vec{R_0} + \rho U^T \vec{n_0} = \vec{R_0}' \tag{1.20}$$

that can be solved as

$$\begin{aligned} U^T \vec{n_0} = \vec{n_0}' = z'(u'_x, u'_y, 1) \\ \rho = -\frac{z'_0}{z'} \end{aligned} \tag{1.21}$$

1.6.3.1 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.core.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]
```

1.6.4 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))

$$I(r) = I_0 e^{-\frac{2r^2}{w(z)^2}} = \frac{2P}{\pi w(z)^2} e^{-\frac{2r^2}{w(z)^2}} \tag{1.22}$$

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

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

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

A Gaussian beam spreads out as

$$w(z)^2 = w_0^2 \left[1 + \left(\frac{\lambda z}{\pi w_0^2}\right)^2\right] = w_0^2 \left[1 + \left(\frac{z}{z_R}\right)^2\right] \tag{1.23}$$

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. 1.17 (from Edmund Optics, Gaussian beam propagation).

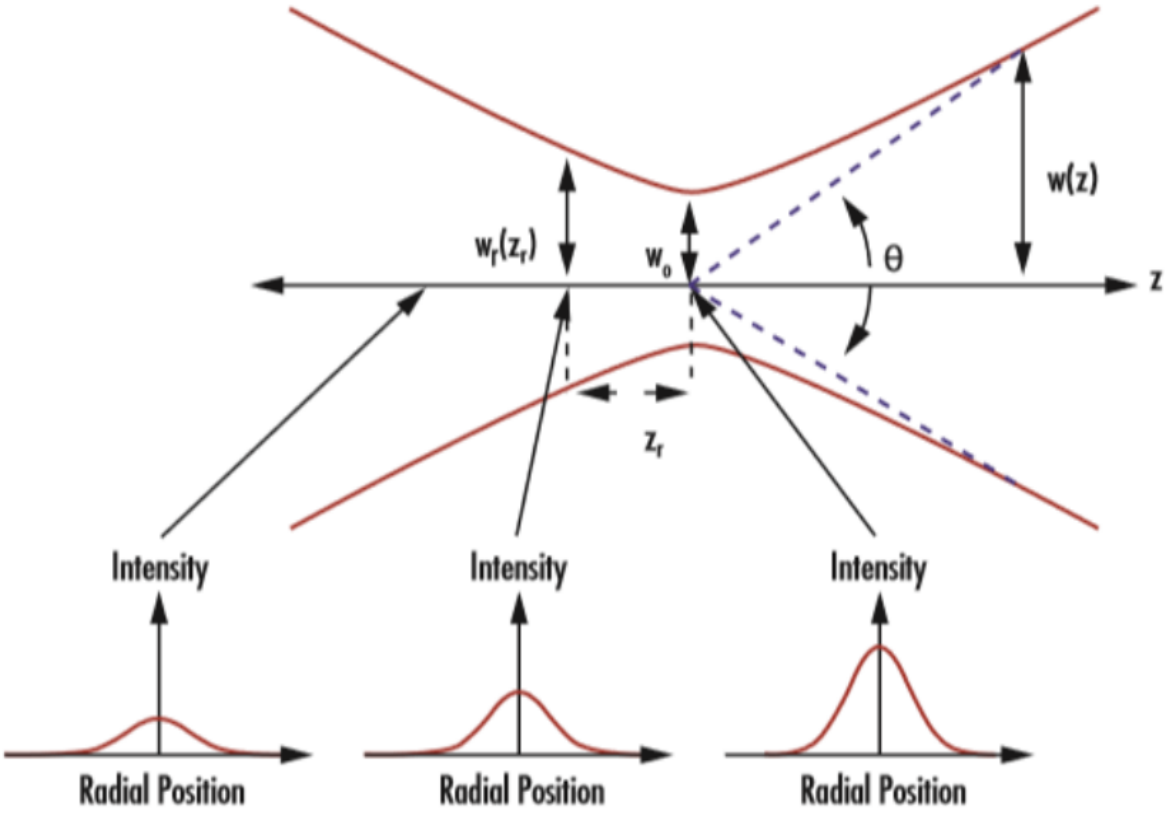


Fig. 1.17 – *Gaussian beam diagram*

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

$$a(r,0) = e^{-\frac{r^2}{w_0^2}} e^{-\frac{jkr^2}{R}} \tag{1.24}$$

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

$$a(r,0) = e^{-\frac{r^2}{w_0^2}} \tag{1.25}$$

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

1.6.4.1 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:

$$z_R = \frac{\pi w_0^2}{\lambda} \tag{1.26}$$

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

$$R(z) = z + \frac{z_R^2}{z} \tag{1.27}$$

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*).

1.6.4.2 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))

$$a(r,z) = e^{-j[kz-\theta(z)]} e^{-\frac{r^2}{w(z)^2}} e^{-\frac{jkr^2}{R(z)}} \tag{1.28}$$

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

$$\theta(z) = tan^{-1}\left(\frac{z_R}{z}\right) \tag{1.29}$$

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

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

$$\frac{1}{q(z)} = \frac{1}{R(z)} - \frac{j\lambda}{\pi n w(z)^2} \tag{1.30}$$

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

$$\frac{1}{q_2} = \frac{C + D/q_1}{A + B/q_1} \tag{1.31}$$

1.6.4.3 Example

Code example to use *WFO* to estimate Gaussian beam properties for a given beam with diameter $d = 1.0$ m, before and after inserting a Paraxial lens with focal length $f = 1.0$ m, and after propagating to the lens focus. The zoom parameter is set to $z = 4$.

Important: The zoom parameter is the ratio between the grid's linear dimension and the beam size.

```
from paos.classes.wfo import WFO

beam_diameter = 1.0  # m
wavelength = 3.0e-6
grid_size = 512
zoom = 4

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

print('Pilot Gaussian beam properties\n')

print('Before lens\n')
print(f'Beam waist: {wfo.w0:.1e}')
print(f'Beam waist at current beam position: {wfo.wz:.1f}')
print(f'z-coordinate of the beam waist: {wfo.zw0:.1f}')
print(f'Rayleigh distance: {wfo.zr:.1e}')
print(f'Focal ratio: {wfo.fratio}')

fl = 1.0  # m
wfo.lens(lens_fl=fl)

print('\nAfter lens\n')
print(f'Beam waist: {wfo.w0:.1e}')
print(f'Beam waist at current beam position: {wfo.wz:.1f}')
print(f'z-coordinate of the beam waist: {wfo.zw0:.1f}')
print(f'Rayleigh distance: {wfo.zr:.1e}')
print(f'Focal ratio: {wfo.fratio:.1f}')

wfo.propagate(dz=fl)

print('\nAfter propagation to lens focus\n')
print(f'Beam waist: {wfo.w0:.1e}')
print(f'Beam waist at current beam position: {wfo.wz:.1e}')
print(f'z-coordinate of the beam waist: {wfo.zw0:.1f}')
print(f'Rayleigh distance: {wfo.zr:.1e}')
print(f'Focal ratio: {wfo.fratio:.1f}')
```

```
Pilot Gaussian beam properties

Before lens

Beam waist: 5.0e-01
Beam waist at current beam position: 0.5
z-coordinate of the beam waist: 0.0
Rayleigh distance: 2.6e+05
Focal ratio: inf

After lens

Beam waist: 1.9e-06
Beam waist at current beam position: 0.5
z-coordinate of the beam waist: 1.0
Rayleigh distance: 3.8e-06
Focal ratio: 1.0

After propagation to lens focus

Beam waist: 1.9e-06
Beam waist at current beam position: 1.9e-06
z-coordinate of the beam waist: 1.0
Rayleigh distance: 3.8e-06
Focal ratio: 1.0
```

1.6.4.4 Gaussian beam magnification

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

$$\begin{aligned} A &= M \\ D &= 1/M \\ B &= C = 0 \end{aligned} \tag{1.32}$$

Therefore, for the complex radius of curvature we have that

$$q_2 = M^2 q_1 \tag{1.33}$$

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.

1.6.4.5 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.classes.wfo import WFO

beam_diameter = 1.0  # m
wavelength = 3.0e-6
grid_size = 512
zoom = 4

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

print('Before magnification\n')
print(f'Beam waist: {wfo.w0}')

Ms, Mt = 1.0, 3.0
wfo.Magnification(Ms, Mt)

print('\nAfter magnification\n')
print(f'Beam waist: {wfo.w0}')
```

```
Before magnification

Beam waist: 0.5

After magnification

Beam waist: 1.5
```

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

1.6.4.6 Gaussian beam change of medium

As seen in *Medium change*, a change of medium from n_1 to n_2 can be described using an ABCD matrix with

$$\begin{aligned} A &= 1 \\ D &= n_1/n_2 \\ B &= C = 0 \end{aligned} \tag{1.34}$$

Therefore, for the complex radius of curvature we have that

$$q_2 = q_1 n_2/n_1 \tag{1.35}$$

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

1. $R_2 = R_1 n_2/n_1$
2. $w_2 = w_1$
3. $z_{R,2} = z_{R,1} n_2/n_1$
4. $w_{0,2} = w_{0,1}$
5. $z_2 = z_1 n_2/n_1$

For the phase radius, the semi-diameter of the beam, the Rayleigh distance, the Gaussian beam waist and the distance to focus, respectively.

Moreover, since $\lambda_2 = \lambda_1 n_2/n_1$, it follows that

$$f_{num,2} = f_{num,1} n_1/n_2 \tag{1.36}$$

1.6.4.7 Example

Code example to use *WFO* to simulate a change of medium from $n_1 = 1.0$ to $n_2 = 1.5$, to point out the change in distance to focus.

```
from paos.classes.wfo import WFO

beam_diameter = 1.0  # m
wavelength = 3.0e-6
grid_size = 512
zoom = 4

wfo = WFO(beam_diameter, wavelength, grid_size, zoom)
fl = 1.0  # m
wfo.lens(lens_fl=fl)

print('Before medium change\n')
print(f'Distance to focus: {wfo.distancetofocus:.1f}')

n1, n2 = 1.0, 1.5
wfo.ChangeMedium(n1n2=n1/n2)

print('\nAfter medium change\n')
print(f'Distance to focus: {wfo.distancetofocus:.1f}')
```

```
Before medium change

Distance to focus: 1.0

After medium change

Distance to focus: 1.5
```

1.6.5 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:

$$\begin{aligned} \text{inside} &\leftrightarrow |z - z(w)| \leq z_R \\ \text{outside} &\leftrightarrow |z - z(w)| > z_R \end{aligned} \tag{1.37}$$

The situation is described in Fig. 1.18, taken from Lawrence et al., Applied Optics and Optical Engineering, Volume XI (1992).

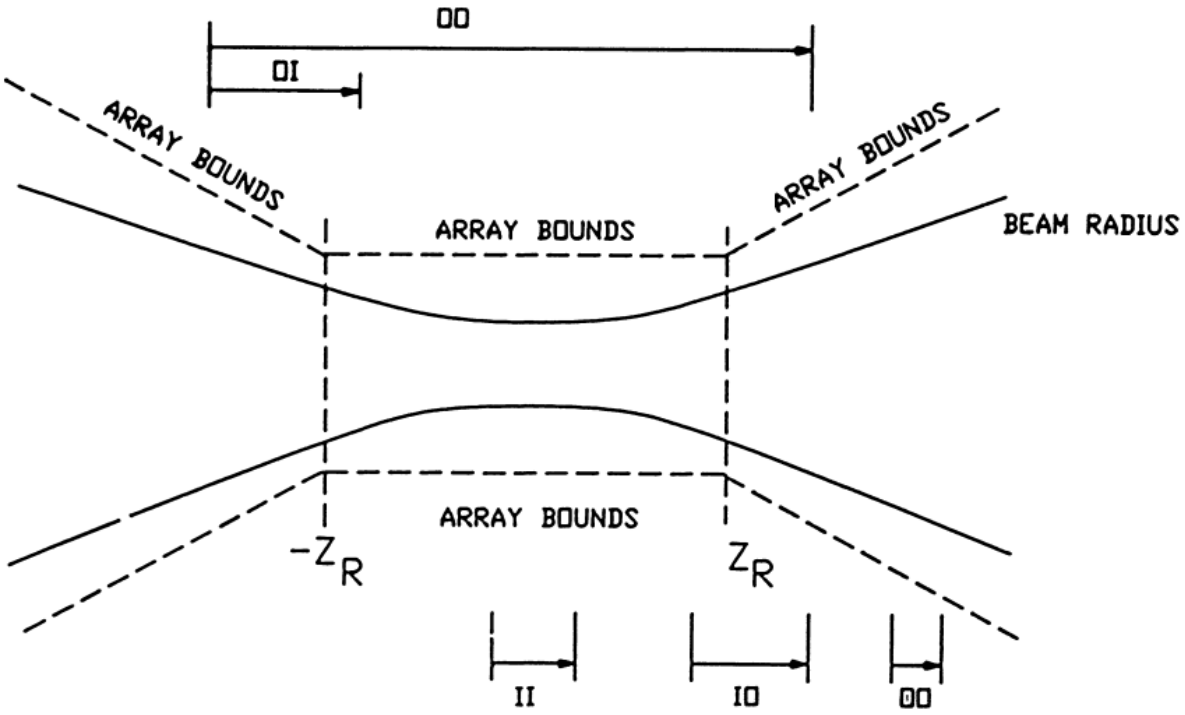


Fig. 1.18 – *Wavefront propagators*

Explicitly, these possibilities are:

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

4. OO(z_1, z_2): outside RR to outside RR

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

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

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

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

1.6.5.1 Example

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

```
from paos.classes.wfo import WFO

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

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

```
Initial beam position: 0.0
Final beam position: 0.01
```

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

1.6.6 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. 1.19.

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

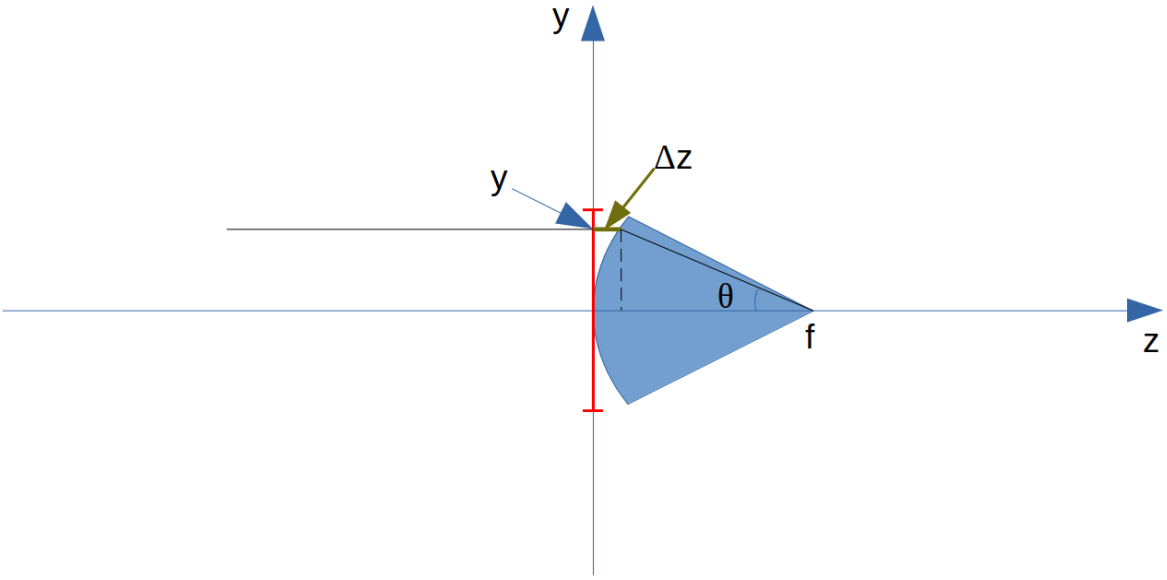


Fig. 1.19 – *Diagram for convergent beam*

4. θ is the angle corresponding to the sag

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

$$a_2(x, y, z) = a_1(x, y, z)e^{2\pi j \Delta z/\lambda} \tag{1.38}$$

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

$$\Delta z = f - \sqrt{f^2 - y^2} \simeq \frac{y^2}{2f} \tag{1.39}$$

The phase delay over the whole lens aperture is then

$$\Delta\Phi = -\Delta z/\lambda = -\frac{x^2 + y^2}{2f\lambda} \tag{1.40}$$

1.6.6.1 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. 1.20.

The total phase delay is then

$$\Delta\Phi = -\frac{x^2 + y^2}{2f\lambda} + \frac{yu_1}{\lambda} = -\frac{x^2 + (y - fu_1)^2}{2f\lambda} + \frac{yu_1^2}{2\lambda} = -\frac{x^2 + (y - y_0)^2}{2f\lambda} + \frac{y_0^2}{2f\lambda} \tag{1.41}$$

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.

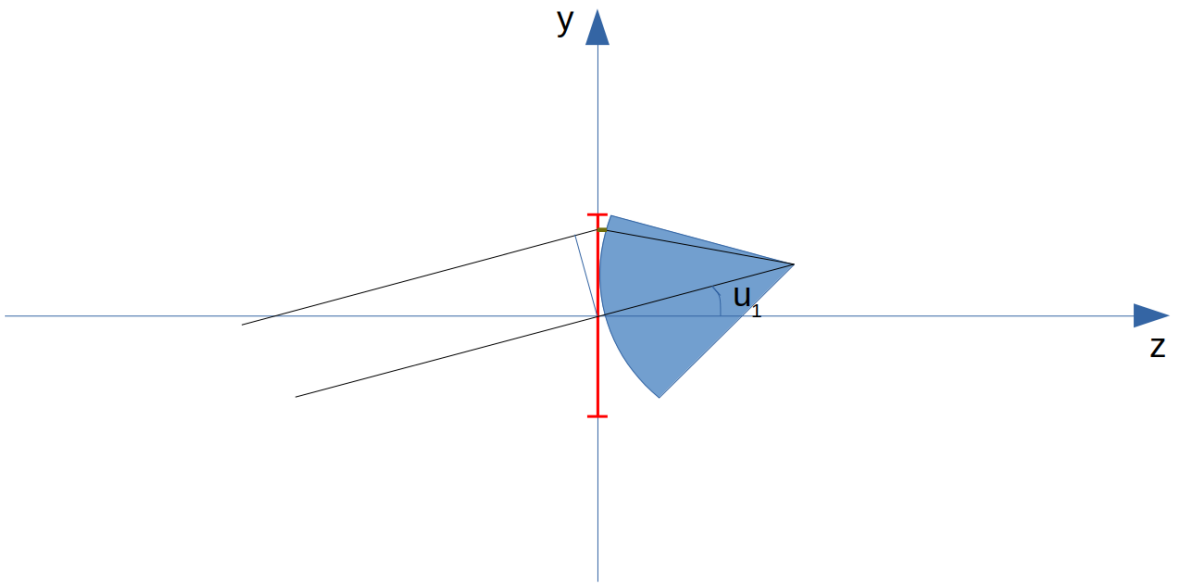


Fig. 1.20 – *Diagram for convergent sloped beam*

1.6.6.2 Off-axis incoming beam

The case of off-axis optics is described in Fig. 1.21.

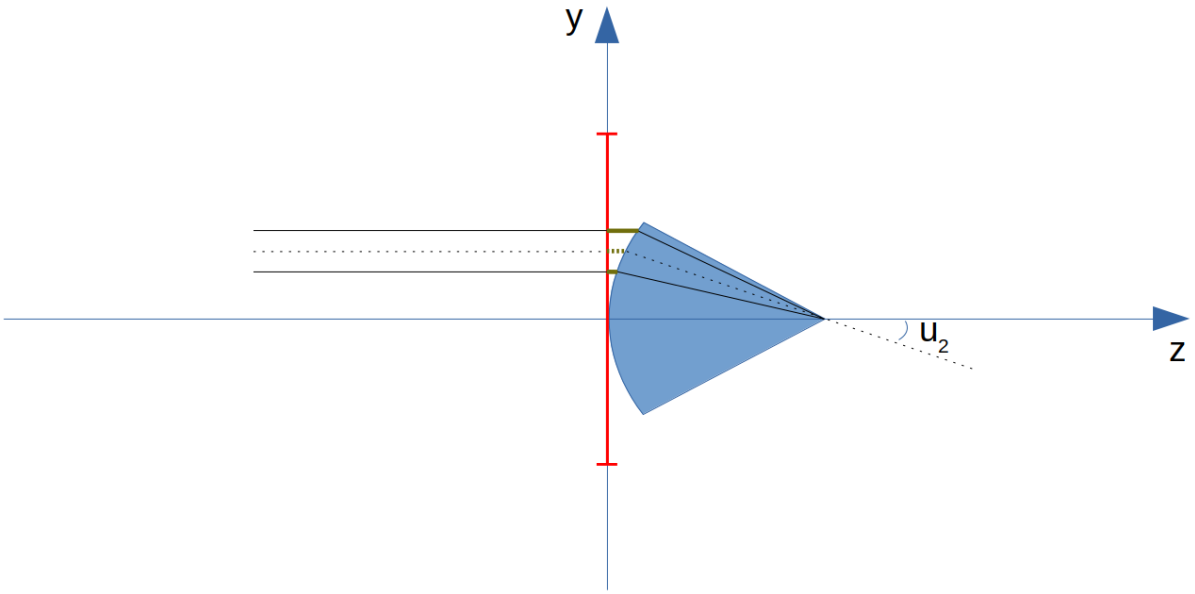


Fig. 1.21 – *Diagram for off-axis beam*

In this case, the beam centre is at y_c.

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

$$\Delta\Phi = -\frac{x^2+y^2}{2f\lambda} = -\frac{x^2+(y_c-\delta y)^2}{2f\lambda} = -\frac{x^2+\delta y^2}{2f\lambda} + \frac{\delta y u_2}{\lambda} - \frac{y_c^2}{2f\lambda} \tag{1.42}$$

If the incoming beam has a slope u_1, then

$$\Delta\Phi = -\frac{x^2+\delta y^2}{2f\lambda} + \frac{\delta y(u_1+u_2)}{\lambda} - \frac{y_c^2}{2f\lambda} + y_c u_1 \tag{1.43}$$

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.

1.6.6.3 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

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

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

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

1. estimate the Gaussian beam curvature after the element (object space) using Eq. (1.27)
2. check the initial position using Eq. (1.37)
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.

1.6.7 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.

1.6.7.1 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.classes.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
```

1.6.8 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.

1.6.8.1 Example

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

```
import numpy as np
from paos.classes.wfo import WFO

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

print('Before stop\n')
print(f'Total throughput: {np.sum(wfo.amplitude**2)}')

wfo.make_stop()

print('\nAfter stop\n')
print(f'Total throughput: {np.sum(wfo.amplitude**2)}')
```

```
Before stop

Total throughput: 262144.0

After stop

Total throughput: 1.0
```

1.6.9 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:

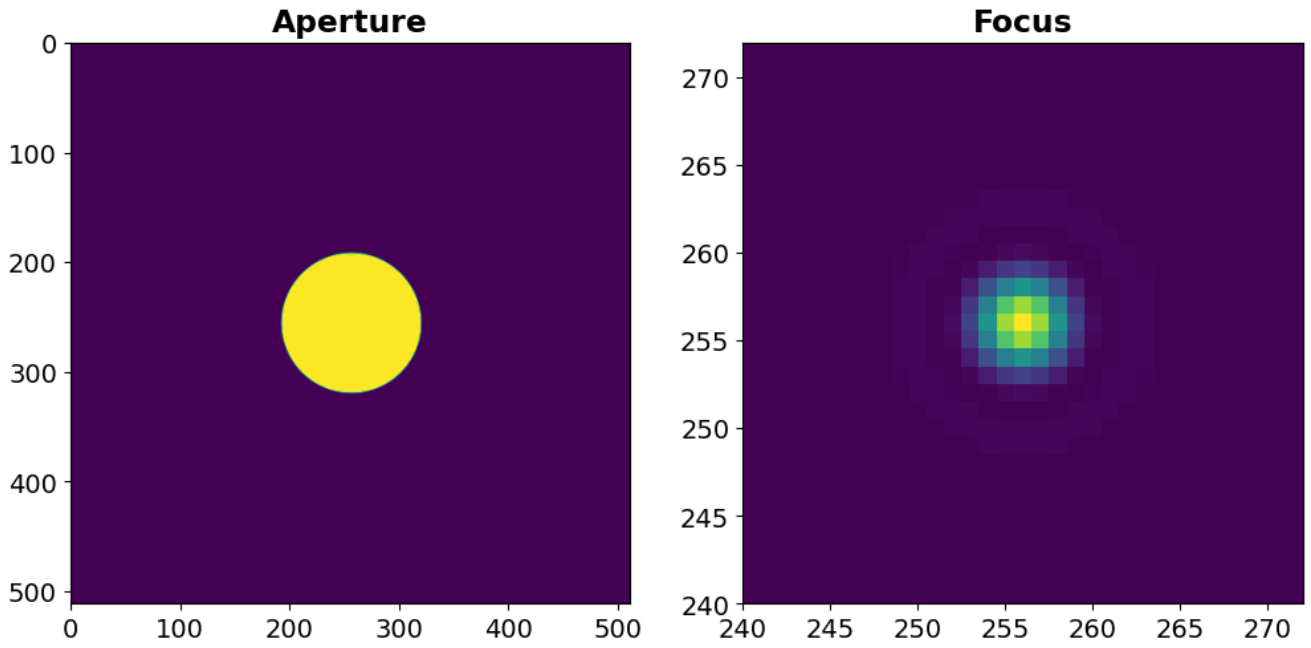
1. Apply coordinate break
2. Apply aperture
3. Apply stop
4. Apply aberration (see *Aberration description*)
5. Save wavefront properties
6. Apply magnification
7. Apply medium change
8. Apply lens
9. Apply propagation by thickness
10. Update ray vectors and ABCD matrices (and save them)
11. Repeat over all optical elements

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

1.6.9.1 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
from paos.classes.wfo import WFO

fig, (ax0, ax1) = plt.subplots(nrows=1, ncols=2, figsize=(12, 6))
wfo = 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()
```



1.7 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*.

1.7.1 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 self-consistency.

1.7.1.1 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.

Image quality can be assessed using 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

$$\text{Strehl ratio} \simeq 1 - k^2\sigma_W^2 \tag{1.44}$$

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.

1.7.1.2 Encircled energy

Another way to assess image quality is by estimating 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

$$f = \int_0^{2\pi} \int_0^{R_f} PSF(r, \phi)\ r\ dr\ d\phi \tag{1.45}$$

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

$$r = R_f F_\# \lambda \tag{1.46}$$

Fig. 1.22 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.

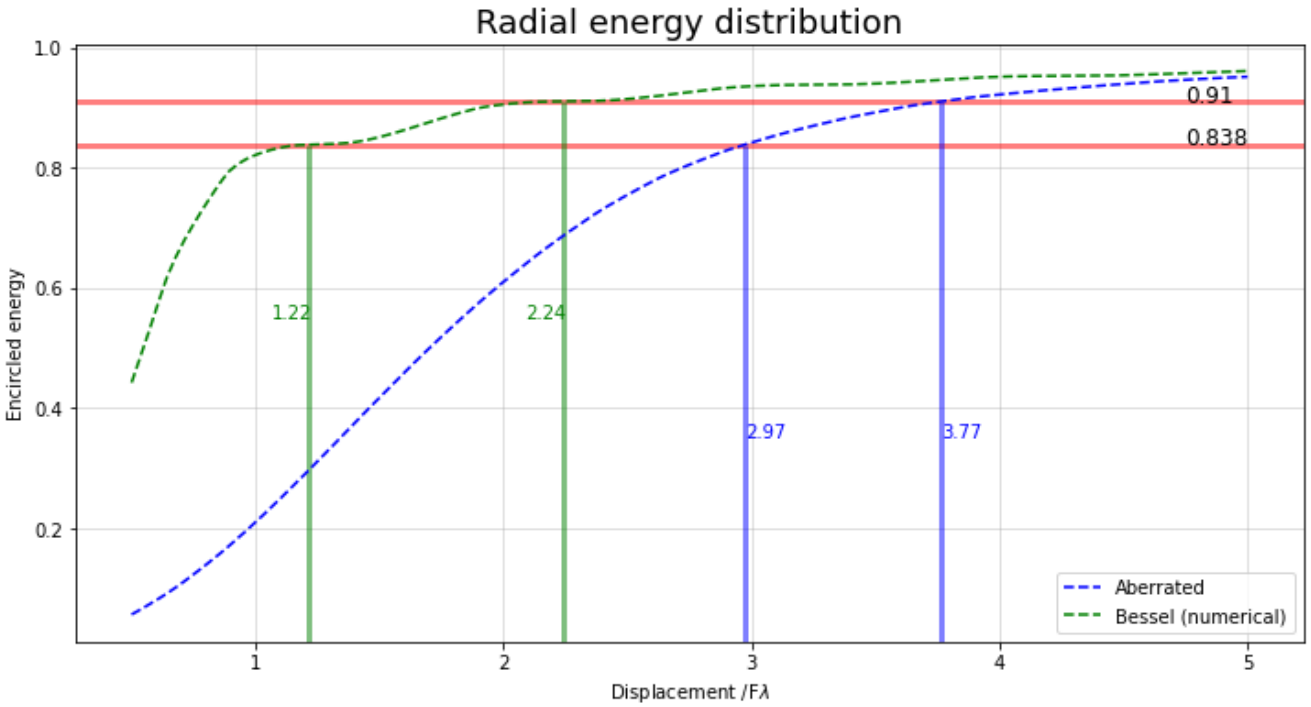


Fig. 1.22 – *Encircled energy*

1.7.2 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 $\mathrm{W}(r, \theta)$ can be expanded in terms of a sequence of Zernike polynomials as

$$W(\rho, \theta) = \sum_{n,m} C_n^m Z_n^m(\rho, \theta) \tag{1.47}$$

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

The first three terms in (1.47) describe Piston and Tilt aberrations and can be neglected. Non-normalised Zernike polynomials are defined in `PAOS` as:

$$Z_n^m = \begin{cases} R_n^m(\rho)\ cos(m\phi) & m \geq 0 \\ R_n^{-m}(\rho)\ cos(m\phi) & m < 0 \end{cases} \tag{1.48}$$

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

$$\left\langle [Z_n^m(\rho, \phi)]^2 \right\rangle = 2\frac{n+1}{1+\delta_{m0}} \tag{1.49}$$

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:

$$\rho^2 = \frac{x_{pup}^2}{a^2} + \frac{y_{pup}^2}{b^2} \tag{1.50}$$

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. 1.23 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)).

1.7.2.1 Example of an aberrated pupil

An example of aberrated PSFs at the *Ariel* Telescope exit pupil is shown in Fig. 1.24.

In this figure, the same Surface Form Error (SFE) of 50 nm 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.

1.7.3 Surface roughness

Optical elements exhibit surface roughness, i.e. medium to high frequency defects produced during manufacturing (e.g. using diamond turning machines). These types of defects reduce the Strehl ratio without significantly altering the PSF's fundamental shape.

The resulting aberrations can be statistically described using a zero-mean random Gaussian field with variance σ_G or relative to the spatial scales of interest using e.g. a parameterized Power Spectral Density (PSD) specification (Church1991).

Users can easily implement this using the PAOS API if necessary, and there are potential plans for inclusion in future PAOS releases.

1.8 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*.

1.8.1 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 lens*.

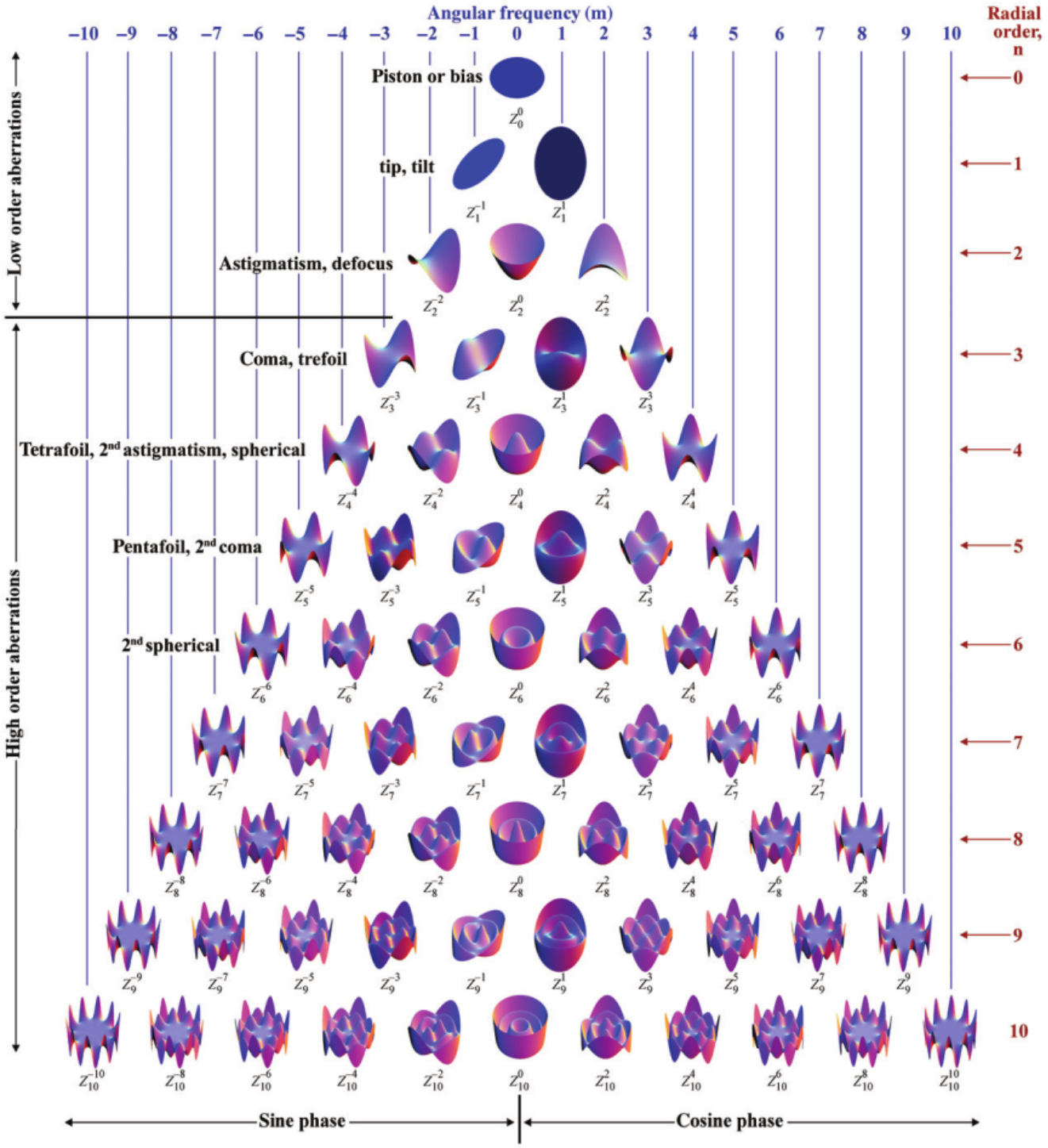


Fig. 1.23 – *Zernike polynomials surface plots*

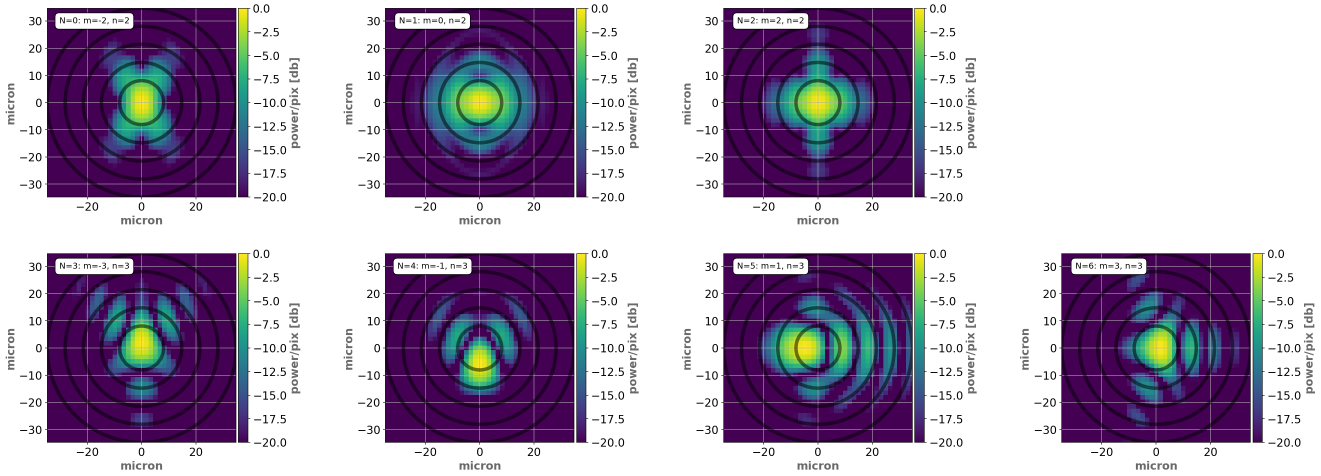


Fig. 1.24 – *Ariel Telescope exit pupil PSFs for different aberrations and same SFE*

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`.

1.8.2 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

$$n^2(\lambda) = 1 + \sum_i \frac{K_i \lambda^2}{\lambda^2 - L_i} \tag{1.51}$$

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 © notation) to estimate the index of refraction relative to air for a particular optical glass at the glass reference temperature and pressure

$$\begin{aligned} T_{ref} &= 20°C \\ P_{ref} &= 1 \text{ atm} \end{aligned} \tag{1.52}$$

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

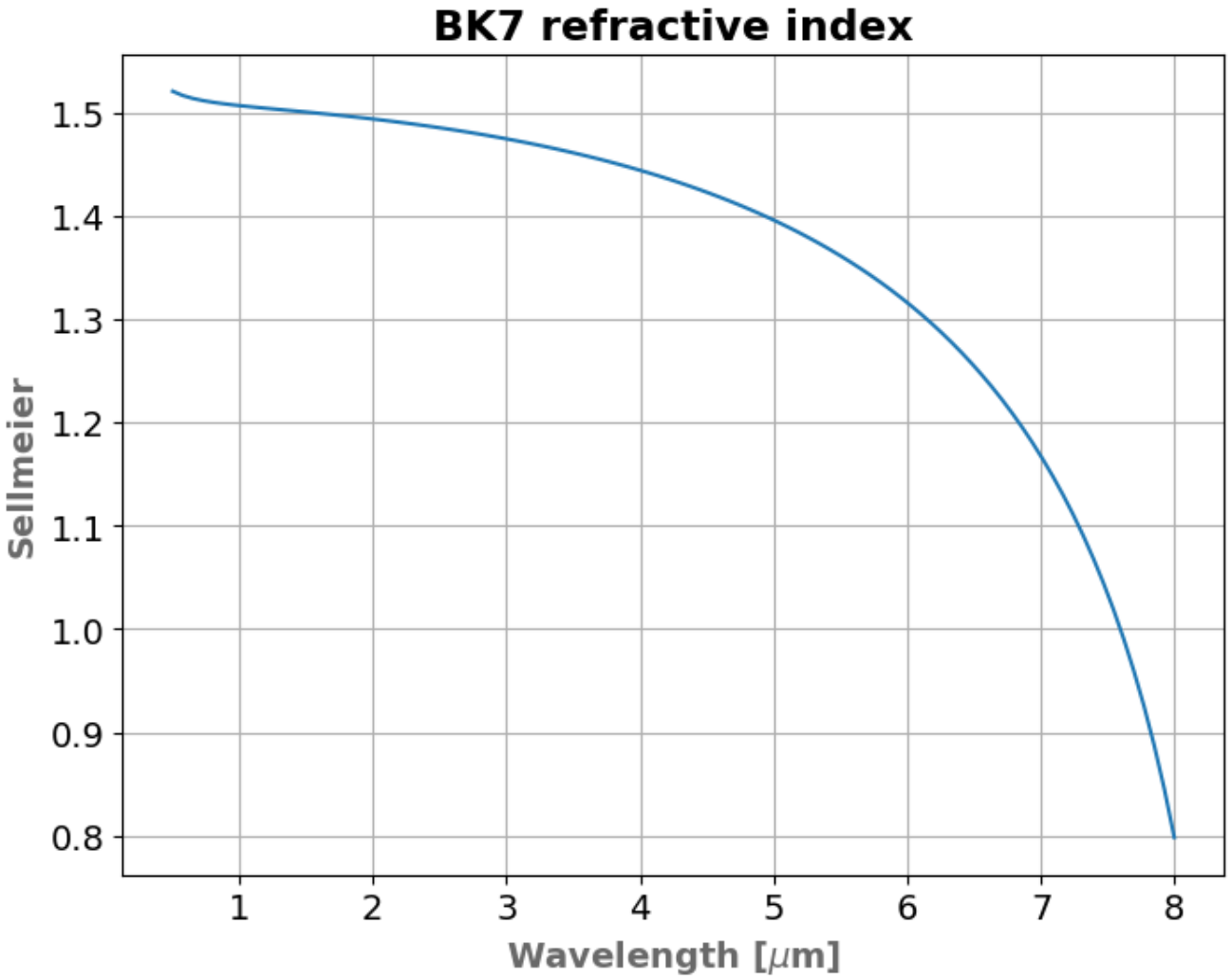
$$n^2(\lambda) = 1 + \frac{K_1\lambda^2}{\lambda^2 - L_1} + \frac{K_2\lambda^2}{\lambda^2 - L_2} + \frac{K_3\lambda^2}{\lambda^2 - L_3} \tag{1.53}$$

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).

1.8.2.1 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'
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()
```



1.8.3 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

$$n_{ref} = 1.0 + 1.0 \cdot 10^{-8} \left(6432.8 + \frac{2949810\lambda^2}{146\lambda^2 - 1} + 25540 \frac{\lambda^2}{41\lambda^2 - 1} \right) \tag{1.54}$$

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

$$n_{air} = 1 + \frac{P\,(n_{ref} - 1)}{1.0 + 3.4785 \cdot 10^{-3}(T - 15)} \tag{1.55}$$

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

$$\frac{dn}{dT}, \text{absolute} = \frac{dn}{dT}, \text{relative} + n\left(\frac{dn}{dT}, \text{air}\right) \tag{1.56}$$

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

$$n(\Delta T) = \frac{n^2 - 1}{2n} D_0 \Delta T + n \tag{1.57}$$

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 (1.53) and D_0 is a temperature constant of the material.

1.8.3.1 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 $T_{amb} = -223.0 : n_{BK7,0} = 1.4955$

1.8.4 Pressure and refractive index

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

1.8.4.1 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)
```

(continues on next page)

(continued from previous page)

```
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)))
```

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

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

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

1.8.5 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. 1.25 reports their transmission range (from Thorlabs, Optical Substrates).

1.8.5.1 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
SF6
SF11
BAF2
```

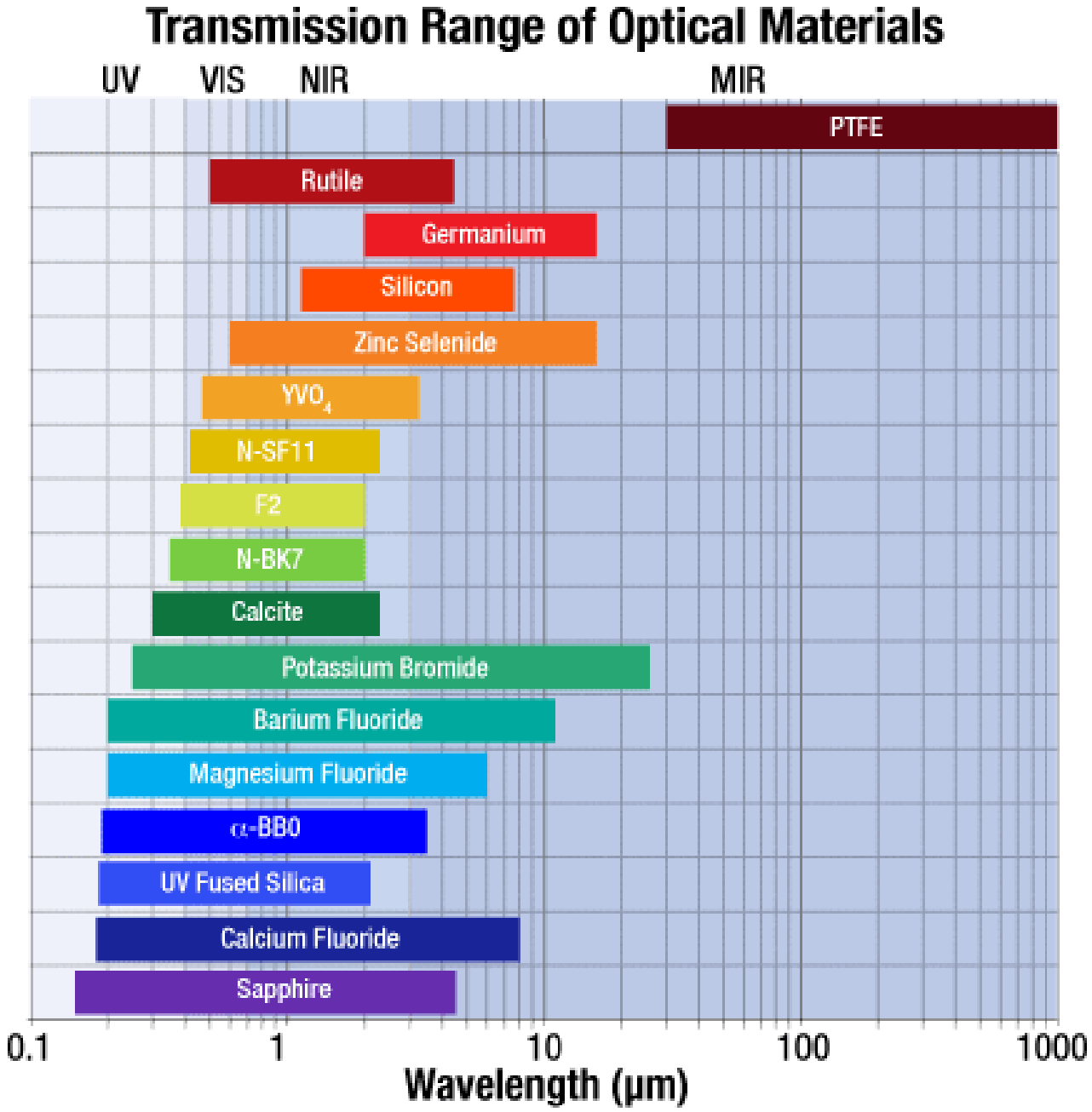
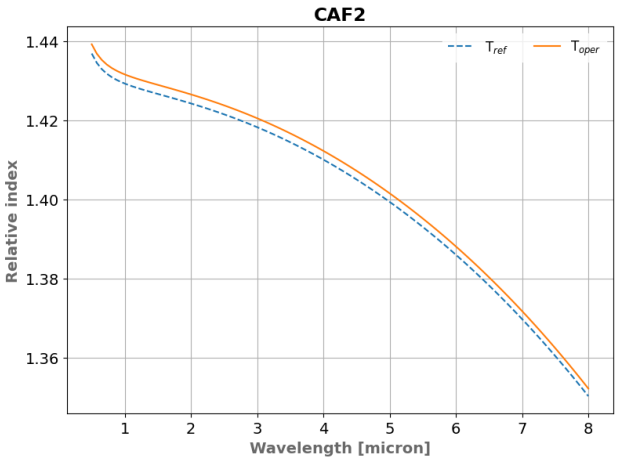


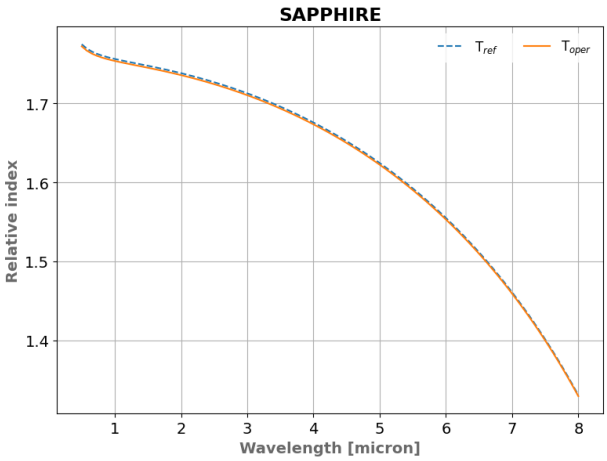
Fig. 1.25 – *Transmission range of optical substrates (Thorlabs)*

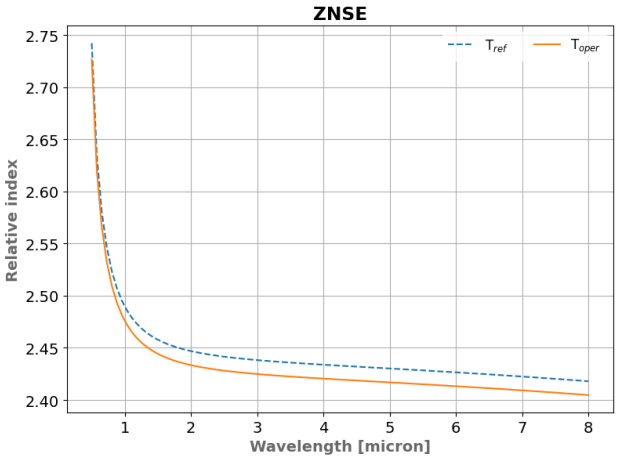
1.8.5.2 Example

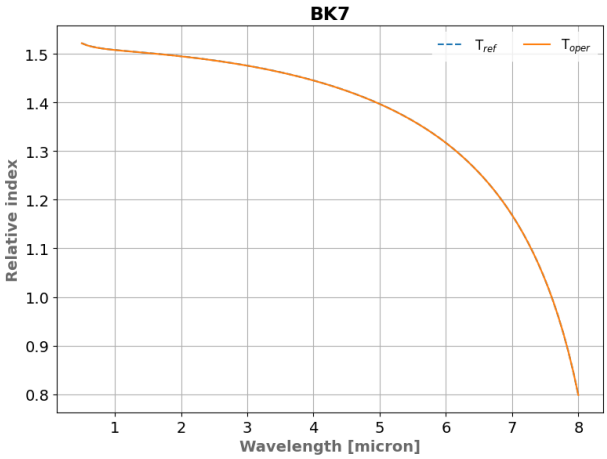
Code example to use *Material* to plot the refractive index for all available optical materials, at their operating and reference temperature.

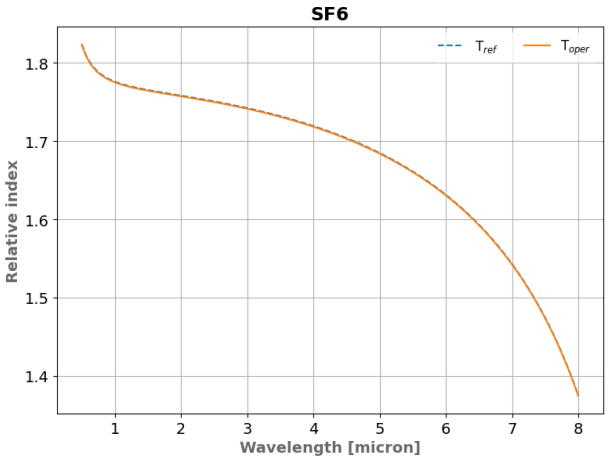
```
from paos.util.material import Material

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

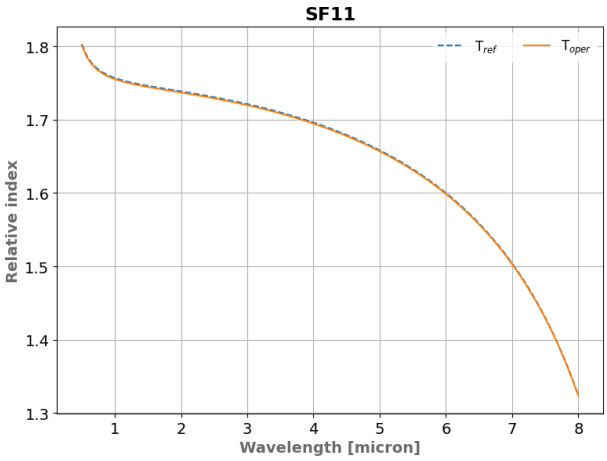

CAF2
Relative index
Wavelength [micron]
T_{ref}
T_{oper}

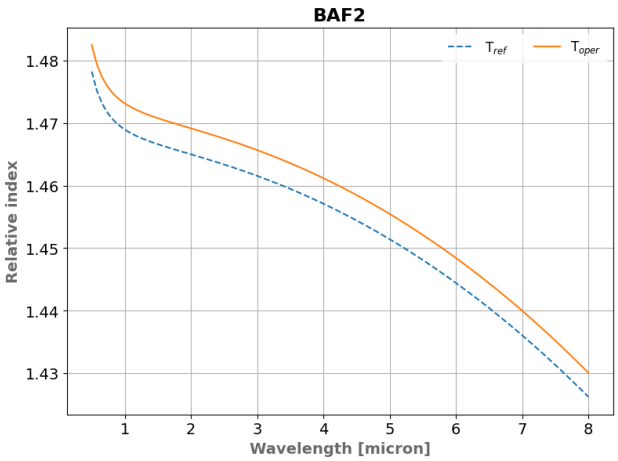

SAPPHIRE
Relative index
Wavelength [micron]
T_{ref}
T_{oper}


ZNSE
Relative index
Wavelength [micron]
T_{ref}
T_{oper}


BK7
Relative index
Wavelength [micron]
T_{ref}
T_{oper}


SF6
Relative index
Wavelength [micron]
T_{ref}
T_{oper}


SF11
Relative index
Wavelength [micron]
T_{ref}
T_{oper}


BAF2
Relative index
Wavelength [micron]
T_{ref}
T_{oper}

1.9 Monte Carlo simulations

PAOS is designed to easily accommodate customized simulations such as Monte Carlo runs to test the performance of an optical system with varying parameters. This is particularly useful as it overcomes one major drawback from using commercial propagation software such as Zemax OpticStudio, which requires preliminary knowledge of Zemax Programming Language (ZPL). Instead, PAOS can be used *out of the box* as a standard Python library, interleaved with user-written code to suit a specific simulation. Moreover, PAOS's routines can be easily run in parallel by leveraging standard Python libraries such as joblib and tqdm, for computational efficiency.

1.9.1 Multi-wavelength simulations

PAOS uses the method *parse_config* to parse the .ini configuration file and return a list of optical chains, where each list entry is a dictionary of the optical surfaces in the file, estimated at the given wavelength.

This output can be readily used to run POP simulations at each different wavelength, to test that the system properties and optical performance are always compliant to specification. For instance, wavelength-dependent total throughput for systems with optical diaphragms and variations in effective focal ratio for systems with diffractive elements.

1.9.2 Wavefront error simulations

PAOS can be used to evaluate the performance of an optical system for a given number of wavefront error realizations, to test the compatibility of the aberrated PSFs with some performance requirement. For instance, PAOS provides an ensemble of wavefront error realizations (see Fig. 1.26) that are compatible or nearly compatible with the encircled energy (EE) requirement at the *Ariel* telescope exit pupil.

The recommended way to access this dataset is using the astropy method `ascii` as in the following code example.

```
import os
from astropy.io import ascii

wfe_file = os.path.join('path/to/wfe_file.csv')
wfe = ascii.read(wfe_file)
```

The whole set provides an effective way to test subsystems optical performances ahead of a measurement of the surface deviation of the Ariel telescope assembly (TA).

For example, it has been used to derive the rEE (radius of encircled energy) requirement for the *Ariel* Optical Ground Support Equipment (OGSE), whose primary goal is to provide end-to-end testing of the integrated *Ariel* telescope, optical bench and spectrometers. To account for gravity effect (*potato chip*), vertical astigmatism was fixed to 3 μm root mean square (r.m.s.) as a rough estimate that will be replaced in the future with an input from Structural, Thermal and Optical Performance (STOP) analysis.

PAOS was used to simulate the wavefront propagation through the OGSE module at $500nm$, where diffraction effects are smallest. To simulate the OGSE beam, the *Ariel* primary mirror M1 was illuminated with a perfect beam with footprint 1/4 the M1 diameter and the OGSE beam expander was modeled as a lens doublet giving an expansion of 4.

Below, we report the histogram of aperture sizes that give an EE $\sim 90\%$ at the OGSE exit pupil. The difference between these aperture sizes and the TA rEE requirement informs on how aberrated the OGSE beam can be.

	A	B	C	D	E	F	...	ALN	ALO
1	#			EE000	EE001	EE002		EE998	EE999
2	#			86.7	87.5	87.6		94.5	94.6
3	# J	N	M	WFE000	WFE001	WFE002		WFE998	WFE999
4	4	2	2	77.2	-25.6	-16.2		29.2	14.8
5	5	2	0	70	-29.4	-15.4		36.6	15
6	6	2	-2	-3.6	6.4	33		-49.8	25.4
7	7	3	3	-7.6	14	-2.8		44	-54.2
8	8	3	1	-6.2	28.8	13.6		-29.4	-11.6
9	9	3	-1	13.6	-1.2	-53.6		-13.2	-10.2
10	10	3	-3	43	66	-20		17	10.8
11	11	4	4	13.4	-5.8	-14.2		-15.4	-17
12	12	4	2	2.2	4.4	-29.2		0.6	-2.4
13	13	4	0	7.8	-2	-37.4		-1.6	0.8
14	14	4	-2	7	-11.2	-7.8		-7.4	5.6
15	15	4	-4	36.6	17.6	4		2.6	-54.6
16	16	5	5	-46.4	33.2	31.6	. . .	-4	16.4
17	17	5	3	2.6	23.8	1.2		-13.8	-9.6
18	18	5	1	4	10.2	-8.2		35.4	13.2
19	19	5	-1	-15.2	24.2	18.4		17.6	-0.6
20	20	5	-3	-14.6	7.8	-16.4		-8.4	-43.6
21	21	5	-5	13.4	47.6	-22.8		36	14.6
22	22	6	6	15	8	-29.8		25.2	27.6
23	23	6	4	19	10	-24.2		-12.4	-3.6
24	24	6	2	14.8	16.4	-6.6		11.8	3.8
25	25	6	0	-11	17.2	13.4		-6.6	-13.2
26	26	6	-2	17	5.2	-0.2		-10.8	-11.6
27	27	6	-4	-11.2	3.4	28.2		12	3.4
28	28	6	-6	-0.8	2.6	13.6		-13.2	6.2
29	29	7	7	-7.4	-5.2	3.2		9.8	-0.6
30	30	7	5	5.8	13.8	-6		-0.6	19.2
31	31	7	3	7.6	3	-2.4		-1.2	4.8
32	32	7	1	2.4	2.8	-5.6		-2.2	-12.8
33	33	7	-1	-12.2	-11.4	-15.6		3	3.4
34	34	7	-3	0.8	-9.4	-9		7.4	-20.4
35	35	7	-5	-15.4	14	0.2		7.6	16.6
36	36	7	-7	1.4	0.6	-4.6		-18.2	-9.8

Fig. 1.26 – *Wfe realizations table*

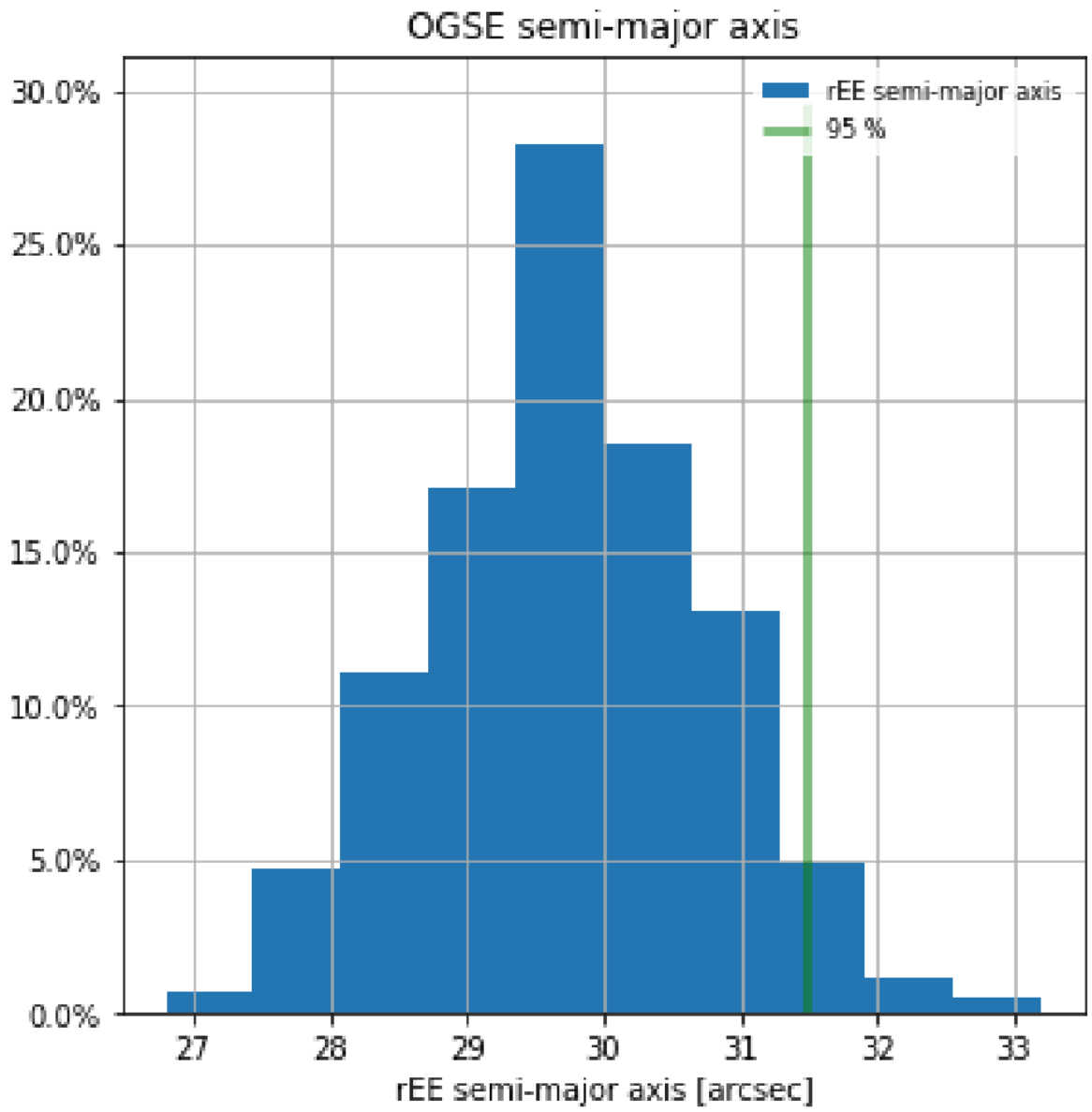


Fig. 1.27 – *Histogram of aperture sizes for OGSE*

1.10 Plotting results

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

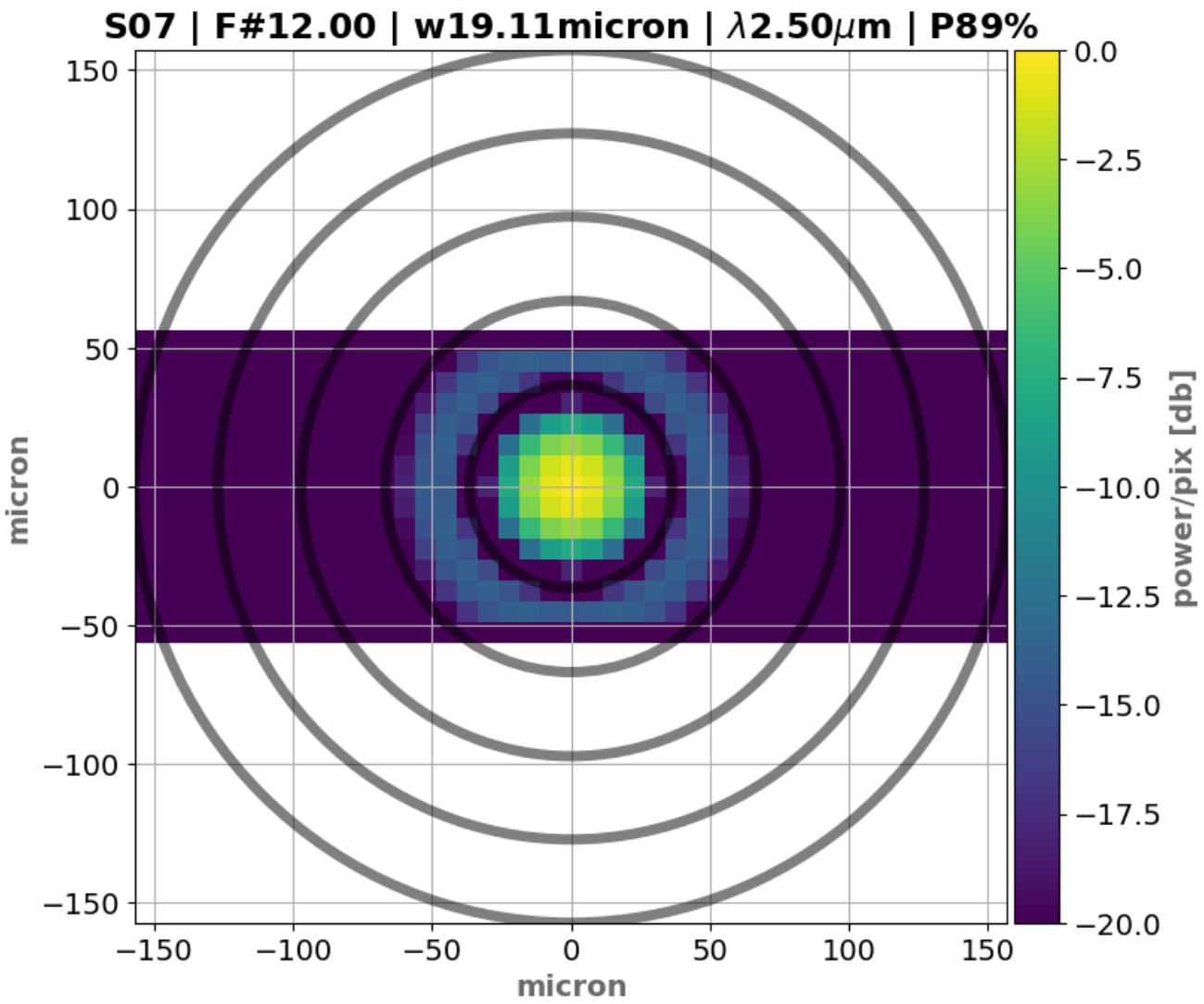
1.10.1 Base plot

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.

1.10.1.1 Example

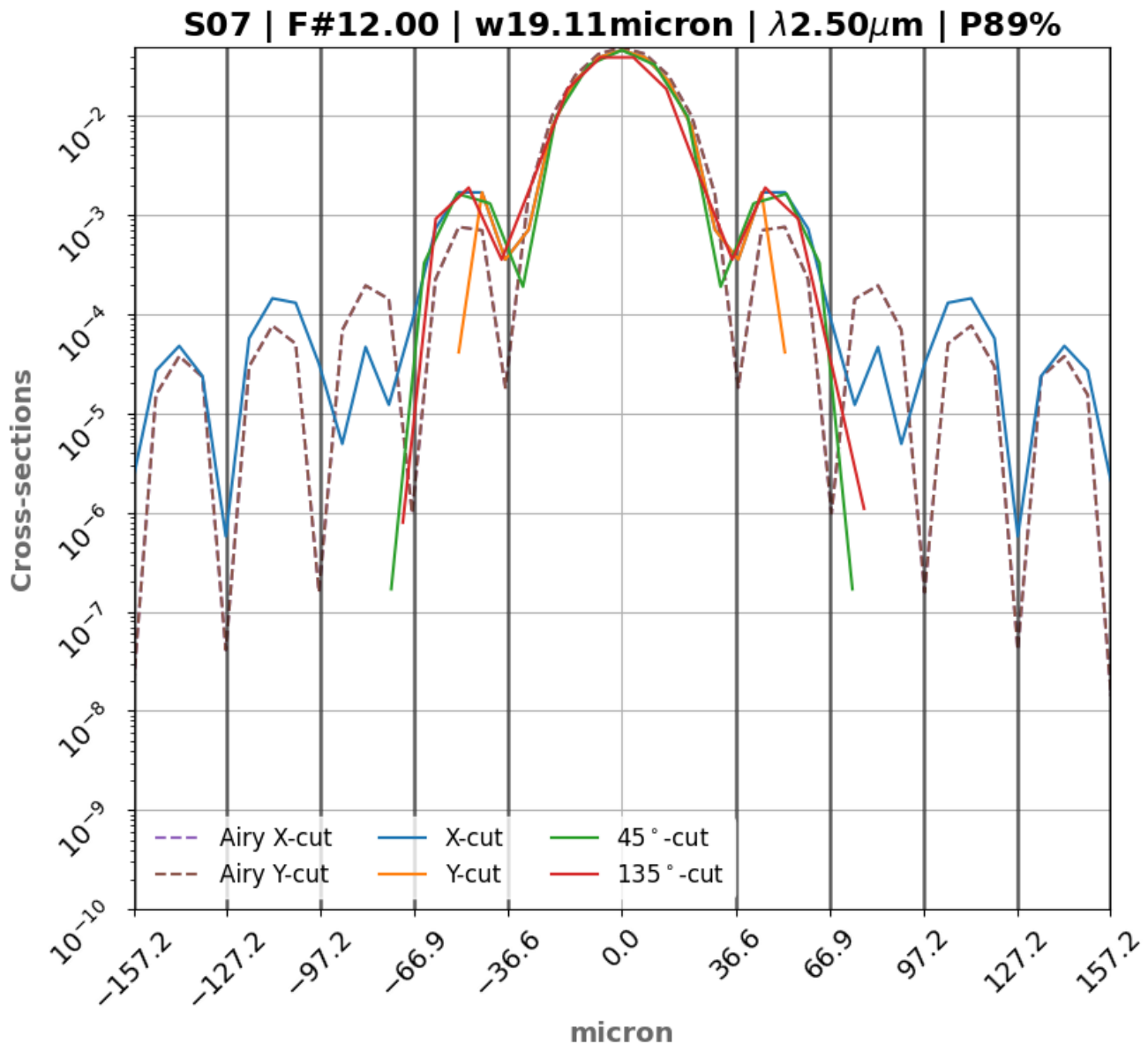
Code example to use *simple_plot* to plot the expected PSF at the image plane of the EXCITE optical chain.

```
import matplotlib.pyplot as plt
from paos.core.plot 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.core.plot 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()
```



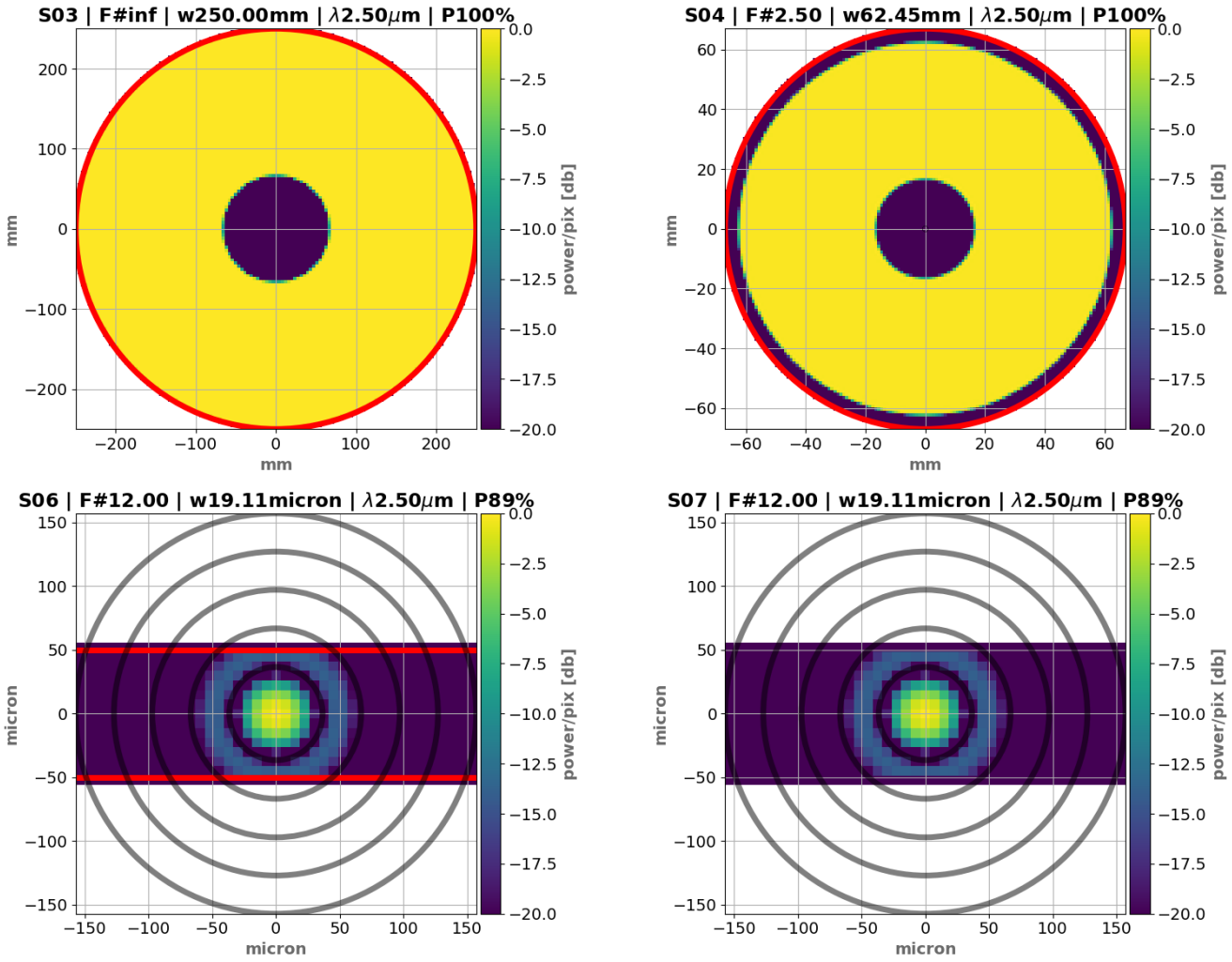
1.10.2 POP plot

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.

1.10.2.1 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.core.plot import plot_pop
plot_pop(ret_val, ima_scale='log', ncols=2)
```



1.11 Saving results

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

1.11.1 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. 1.28. 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. 1.29.

The 'info' folder contains the data that are needed for traceability and versioning of the results, as shown in Fig. 1.30.

This includes:

1. The HDF5 package version

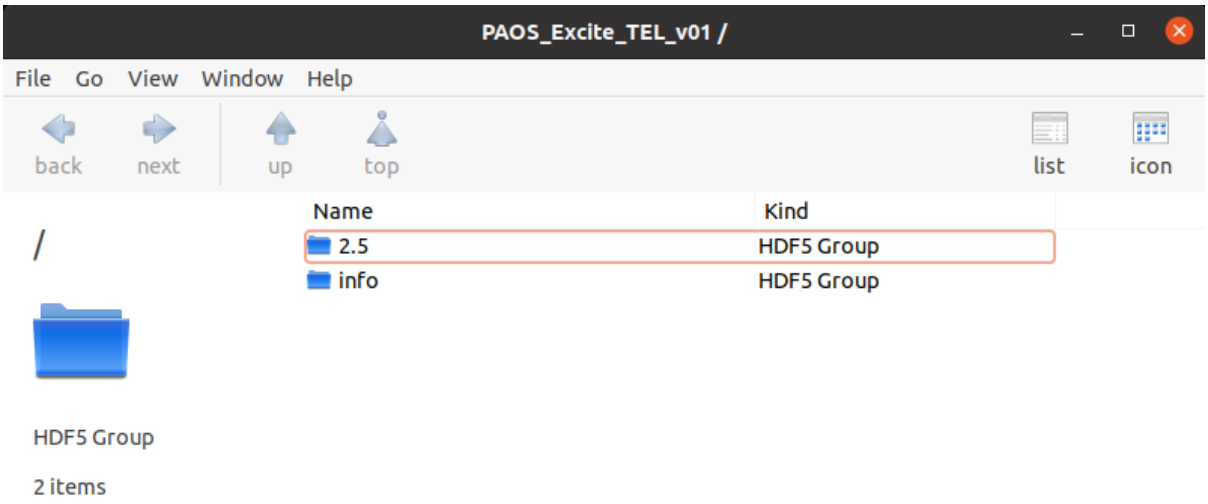


Fig. 1.28 – *Output file general interface*

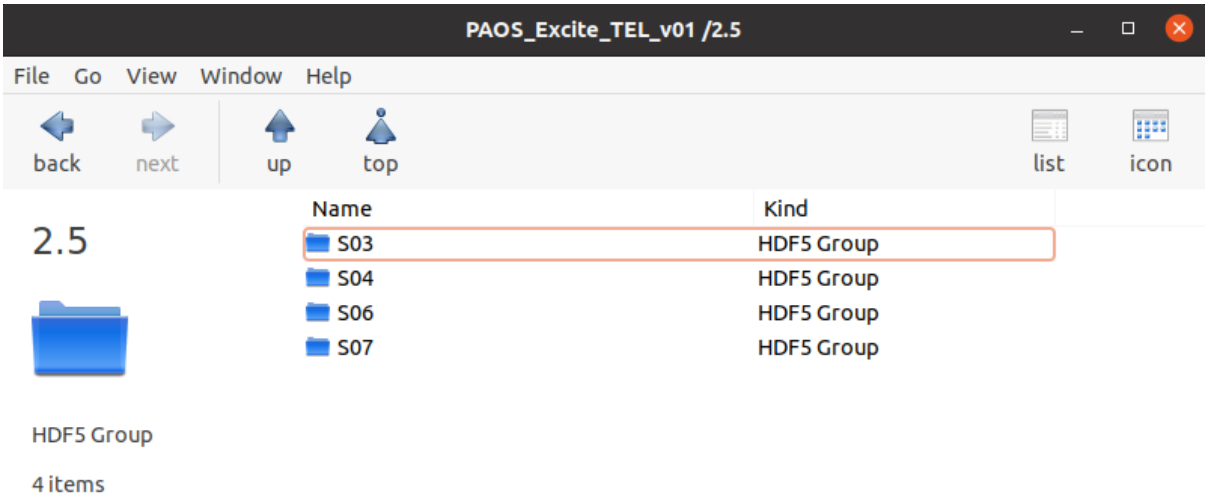


Fig. 1.29 – *Output file surfaces interface*

2. The PAOS creator names
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

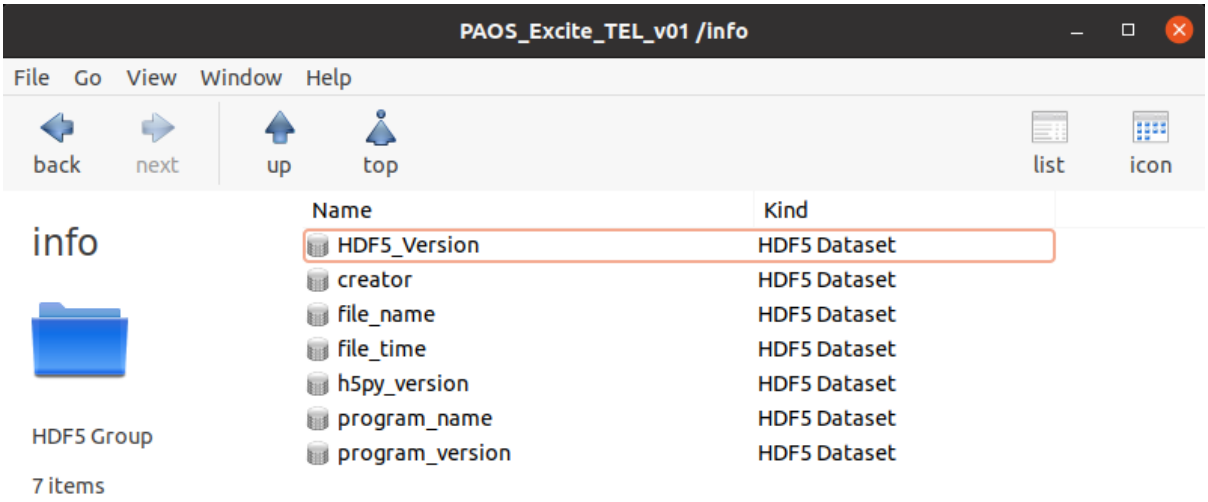


Fig. 1.30 – *Output file info interface*

1.11.1.1 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.core.saveOutput import save_output
save_output(ret_val,
            file_name='path/to/hdf5',
            keys_to_keep=['wfo', 'dx', 'dy'],
            overwrite=True)
```

1.11.2 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. 1.31

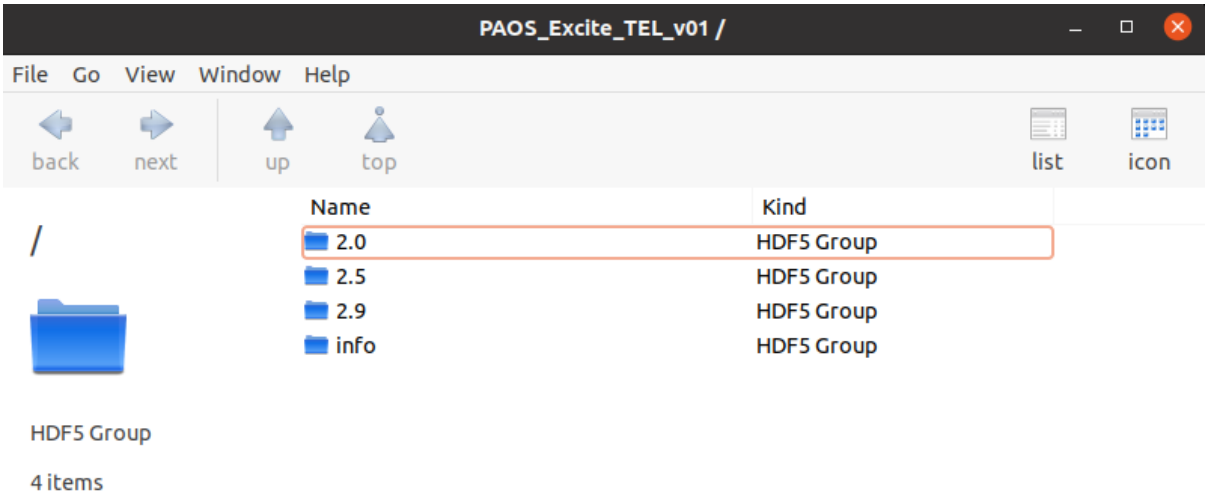


Fig. 1.31 – *Output file cube general interface*

1.11.2.1 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).

```
from paos.core.saveOutput import save_datacube

save_datacube(retval_list=ret_val_list,
              file_name='path/to/hdf5',
              group_names=['2.5', '3.0'],
              keys_to_keep=['amplitude', 'dx', 'dy'],
              overwrite=True)
```

1.12 Automatic pipeline

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

1.12.1 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 optical chain for the POP run automatizing the input of an aberration (optional);
5. Runs the POP in parallel or using a single thread;
6. Produces an output where all (or a subset) of the products are stored;
7. If indicated, the output includes plots.

1.12.1.1 Example

Code example to the method *pipeline* to run a simulation using the configuration file for AIRS-CH1.

```
from paos.core.pipeline import pipeline

pipeline(passvalue={'conf':'path/to/ini/file',
                    'output': 'path/to/hdf5',
                    'plot': True,
                    'loglevel': 'info',
                    'n_jobs': 2,
                    'store_keys': 'amplitude,dx,dy,wl',
                    'return': False})
```

Chapter 2

Developer guide

In this section we report some general guidelines for contributing to PAOS development.

The section is inspired by the package ExoSim2.0.

2.1 Coding conventions

The PAOS code has been developed following the PeP8 standard and the python Zen.

```
import this
```

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

2.2 Documentation

Every PAOS function or class should be documented using docstrings which follow numpydoc structure. This web page is written using the reStructuredText format, which is parsed by sphinx. If you want to contribute to this documentation, please refer to sphinx documentation first. You can improve this pages by digging into the *docs* directory in the source.

2.3 Testing

Unit-testing is very important to make sure that each code addition is tested and validated and the code never breaks. This shall be provided after *PAOS v1.0.0*.

2.4 Logging

To keep a logger is very important when coding, hence we include a *paos.log.logger.Logger* class to inherit.

```
import paos.log as log

class MyClass(log.Logger):
    ...
```

This newly created class has the logging methods from the main *Logger* class. Here are some examples of how to use them:

```
self.info("info message")
self.debug("debug message")
self.warning("warning message")
self.error("error message")
self.critical("critical message")
```

The logger output will be printed on the run or stored in the log file, if the log file option is enabled. To enable the log file, the user can refer to `paos.log.addLogFile`.

Note: The logger implemented in PAOS is inspired by the logging classes in ExoSim2.0, which is originally inspired by the ones in TauREx3 (developed by Ahmed Al-Refaie).

The user can also set the level of the printed messaged using `paos.log.setLogLevel`, or enable or disable the messaged with `paos.log.enableLogging` or `paos.log.disableLogging`

2.5 Versioning conventions

The versioning convention used (after *PAOS v1.0.0*) shall be the one described in Semantic Versioning (semver) and shall be compliant to PEP440 standard. In the X.Y.Z scheme, for each modification to the previous release we increase one of the numbers.

- *X*

increased only if the code in not compatible anymore with the previous version. This is considered a Major change.

- *Y*
  increased for minor changes. These are for the addition of new features that may change the results from previous versions. This are still hard edits, but not enough to justify the increase of an *X*.
- *Z*
  the patches. This number should increase for any big fixed, or minor addition or change to the code. It won't affect the user experience in any way.

2.6 Source Control

The code is hosted on GitHub (https://github.com/arielmission-space/PAOS) and structured as following.

The source has two main branches:

- `main`
  branch for stable and releases. It is the public branch and should be handled carefully.
- `develop`
  working branch where the new features are tested before they are moved to the *master* branch

2.6.1 Adding new features

New features can be added to the code following the schemes designed above.

If the contributor has writing rights to the repository, should create a new branch starting from the *develop* one. In the new *feature* branch the user should produce the new functionalities, according to the above guidelines. When the feature is ready, the branch can be merged into the official *develop* one.

To create the new feature starting from the current develop version, the contributor should run

```
$ git checkout develop
$ git checkout -b feature/<branchname>
```

The completed feature shall then be merged to the develop:

```
$ git checkout develop
$ git merge feature/<branchname>
$ git push
```

Once a feature is completed and merged, the contributor should archive the branch and remove it, to keep the repository clean. The usual procedure is:

```
$ git tag archive/<branchname> feature/<branchname>
$ git push --tags
$ git branch -d feature/<branchname>
```

Remember to delete the branch also from the remote repository. If needed, the feature branch can be restored as

```
$ git checkout -b <branchname> archive/<branchname>
```

If the contributor does not have writing rights to the repository, should use the Fork-and-Pull model. The contributor should fork the main repository and clone it. Then the new features can be implemented. When the code is ready, a pull request can be raised.

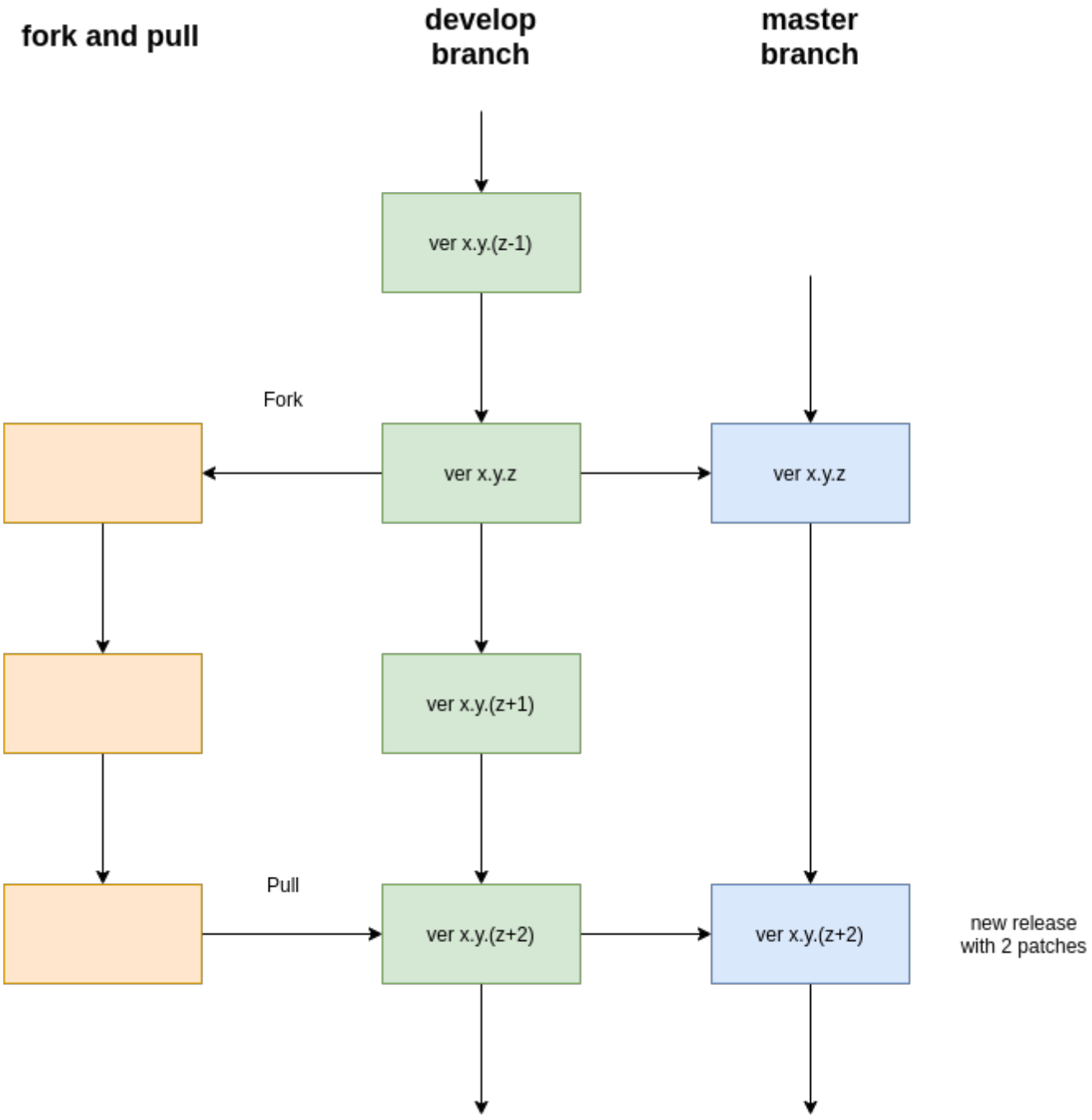


Fig. 2.1 – *Forking and pulling*

Chapter 3

API guide

3.1 ABCD (paos.classes.abcd)

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.classes.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

3.2 WFO (paos.classes.wfo)

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
>>> 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', or 'standard'.
- **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

psd(*A=10.0*, *B=0.0*, *C=0.0*, *fknee=1.0*, *fmin=None*, *fmax=None*, *SR=0.0*, *units=Unit('m')*)

Add a WFE represented by a power spectral density (PSD) and surface roughness (SR) specification.

Parameters

- **A** (*float*) – The amplitude of the PSD.
- **B** (*float*) – PSD parameter. If B = 0, the PSD is a power law.
- **C** (*float*) – PSD parameter. It sets the slope of the PSD.
- **fknee** (*float*) – The knee frequency of the PSD.
- **fmin** (*float*) – The minimum frequency of the PSD.
- **fmax** (*float*) – The maximum frequency of the PSD.
- **SR** (*float*) – The rms of the surface roughness.
- **units** (*astropy.units*) – The units of the SFE. Default is meters.

Returns

out – the WFE

Return type

masked array

3.3 Zernike (paos.classes.zernike)

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 either be:

  ANSI (ordering='ansi', this is the default); Noll (ordering='noll'). Used in Zemax as "Zernike Standard Coefficients",

R. Noll, "Zernike polynomials and atmospheric turbulence", J. Opt. Soc. Am., Vol. 66, No. 3, p207 (1976);
Fringe (ordering='fringe'), AKA the "Fringe" or "University of Arizona" notation; Standard (ordering='standard'). Used in CodeV, Born and Wolf, Principles of Optics (Pergamon Press, New York, 1989).

- **normalize** (*bool*) – Set to True generates ortho-normal polynomials. Set to False generates orthogonal polynomials as described in Laksminarayan & Fleck, Journal of Modern Optics (2011). The radial polynomial is estimated using the Jacobi polynomial expression as in their Equation in Equation 14.

Returns

out – An instance of Zernike.

Return type

masked array

Example

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

3.4 Core (`paos.core`)

3.4.1 parseConfig

`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 given wavelength. (Relevant only for diffractive components)

Examples

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

3.4.2 coordinateBreak

`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.

3.4.3 raytrace

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

3.4.4 run

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.core.parseConfig.parse_config

Returns

out – dictionary containing the results of the POP

Return type

dict

Examples

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

3.4.5 plot

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

Create a nice legend for the plots

Parameters

- **axis** (Axes) – An instance of matplotlib axis
- **ncol** (*int*) – The number of legend columns

Returns

out – Produces a nice matplotlib legend

Return type

None

simple_plot(*fig*, *axis*, *key*, *item*, *ima_scale*, *origin='lower'*, *cmap='viridis'*, *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'
- **origin** (*str*) – matplotlib plot origin. Defaults to 'lower'

- **cmap** (*str*) – matplotlib plot color map. Defaults to 'viridis'
- **options** (*dict*) – dictionary containing the options to override the plotting default for one or more surfaces, specified by the dictionary key. Available options are the surface scale, an option to display physical units, the surface zoom(out), the plot scale and whether to plot dark rings in correspondance to the zeros of the Airy diffraction pattern. 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', 'dark_rings': False}}

Returns

updates the Figure object

Return type

None

Examples

```
>>> from paos.core.parseConfig import parse_config
>>> from paos.core.run import run
>>> from paos.core.plot 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.core.parseConfig import parse_config
>>> from paos.core.run import run
>>> from paos.core.plot 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'*, *surface_zoom=1*)

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/($F_{num}\lambda$).
- **surface_zoom** (*scalar*) – Surface zoom: more increases the axis limits

Returns

out – updates the *~matplotlib.figure.Figure* object

Return type

None

Examples

```
>>> import matplotlib.pyplot as plt
>>> from paos.core.parseConfig import parse_config
>>> from paos.core.run import run
>>> from paos.core.plot 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
>>> 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')
```

plot_surface(*key*, *retval*, *ima_scale*, *origin='lower'*, *zoom=1*, *figname=None*)

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'
- **origin** (*str*) – matplotlib plot origin. Defaults to 'lower'
- **zoom** (*scalar*) – the surface zoom factor: more increases the axis limits
- **figname** (*str*) – name of figure to save

Returns

out – the figure with the squared amplitude of the wavefront at the given surface

Return type

Figure

3.4.6 saveOutput

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.core.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.core.parseConfig import parse_config
>>> from paos.core.run import run
>>> from paos.core.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.core.parseConfig import parse_config
>>> from paos.core.run import run
>>> from paos.core.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)
```

3.4.7 pipeline

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

3.5 GUI (paos.gui)

3.5.1 paosGui

3.5.2 simpleGui

3.5.3 zernikeGui

3.6 Material (paos.util.material)

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 ($°C$)
- **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 $T_{ref} = 20°C$ 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 $°C$
- **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 $T_{ref} = 20°C$ (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'])
```

3.7 Logger (paos.log.logger)

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`

Chapter 4

License

BSD 3-Clause License

Copyright (c) 2022, 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 5

Changelog

All notable changes to this project will be documented in this file.

The format is based on Keep a Changelog (keepachangelog), and this project adheres to Semantic Versioning (semver).

5.1 [Unreleased]

5.1.1 0.0.2 [15/09/2021]

Setting up new `PAOS` repository

5.1.2 0.0.2.1 [20/10/2021]

First documented `PAOS` release

5.1.3 0.0.3 [23/12/2021]

5.1.3.1 Added

- Added support for optical materials

5.1.4 0.0.4 [22/01/2022]

5.1.4.1 Changed

- Changed configuration file to .ini

5.1.5 1.0.0 [01/07/2023]

`PAOS` production ready

Chapter 6

Acknowledgements

The development of PAOS has been possible thanks to Andrea Bocchieri, Lorenzo V. Mugnai, and Enzo Pascale.

Their work was supported by the Italian Space Agency (ASI) with Ariel grant n. 2021.5.HH.0.

Python Module Index

p

Index

R

S

T

W

Z