prysm Documentation

Release 0.14.0

Brandon Dube

Mar 14, 2019

Contents

Release 0.14.0

Date Mar 14, 2019

prysm is an open-source library for physical and first-order modeling of optical systems and analysis of related data. It is an unaffiliated sister library to PROPER and POPPY, codes developed to do physical optics modeling for primarily space-based systems. Prysm has a more restrictive capability in that domain, notably lacking multi-plane diffraction propagation, but also offers a broader set of features.

Contents

CHAPTER 1

Use Cases

prysm aims to be a swiss army knife for optical engineers and students. Its primary use cases include:

- Analysis of optical data
- robust numerical modeling of optical and opto-electronic systems based on physical optics
- wavefront sensing

Please see the Features section for more details.

CHAPTER 2

Installation

prysm is available from either PyPi:

```
>>> pip install prysm
```

or github:

```
>>> pip install git+git://github.com/brandondube/prysm.git
```

It requires a minimal set of dependencies for scientific python; namely numpy, scipy, and matplotlib. It optionally depends on numba (for acceleration of some routines on CPUs), cupy (for experimental use with nVidia GPUs), imageio (for reading and writing images), h5py (for reading of Zygo's datx format), and pandas (for some advanced utility MTF functions). Pip can be instructed to install these alongside prysm,

```
>>> pip install prysm[cpu+]  # for numba
```

```
>>> pip install prysm[cuda] # for cupy
```

```
>>> pip install prysm[img] # for imageio
```

```
>>> pip install prysm[Mx]  # for h5py
```

```
>>> pip install prysm[mtf+] # for pandas
```

or they may be installed at any time.

CHAPTER 3

Features

3.1 Physical Optics

- Modeing of pupil planes via or with:
  - Hopkins' wave aberration expansion, "Seidel aberrations"
  - Fringe Zernike polynomials up to Z48, unit amplitude or RMS
  - Zemax Standard Zernike polynomials up to Z48, unit amplitude
  - apodization
  - masks
    - circles and ellipses
    - n sided regular polygons
    - user-provided
  - synthetic interferograms
  - PV, rms, standard deviation, strehl evaluation
  - plotting
- Propagation of pupil planes via Fresnel transforms to Point Spread Functions (PSFs), which support
  - calculation and plotting of encircled energy
  - evaluation and plotting of slices
  - 2D plotting with or without power law or logarithmic scaling
- Computation of MTF from PSFs via the FFT method
  - MTF Full-Field Displays
  - MTF vs Field vs Focus
    - Best Individual Focus

- – * Best Average Focus
- – evaluation at
- – * exact Cartesian spatial frequencies
- – * exact polar spatial frequencies
- – * Azimuthal average
- – 2D and slice plotting
- Rich tools for convolution of PSFs with images or synthetic objects:
- – pinholes
- – slits
- – Siemens stars
- – tilted squares
- – slanted edges
- read, write, and display of images
- Detector models for e.g. STOP analysis or image synthesis
- Interferometric analysis
- – cropping, masking
- – least-squares fitting and subtraction of Zernike modes, planes, and spheres
- – evaluation of PV, RMS, Sa, standard deviation, band-limited RMS
- – computation of PSD
- – * 2D
- – * x, y, azimuthally averaged slices
- – * evaluation and/or comparison to lorentzian model
- – spike clipping
- – plotting

3.2 First-Order Optics

- object-image distance relation
- F/#, NA
- lateral and longitudinal magnification
- defocus-deltaZ relation
- two lens EFL and BFL

3.3 Parsing Data from Commercial Instruments

- Trioptics ImageMaster MTF benches
- Zygo Fizeau and white light interferometers

CHAPTER 4

User's Guide

4.1 Convolvables

Prysm features a rich implemention of Linear Shift Invariant (LSI) system theory. Under this mathematical ideal, the transfer function is the product of the Fourier transform of a cascade of components, and the spatial distribution of intensity is the convolution of a cascade of components. These features are usually used to blur objects or images with Point Spread Functions (PSFs), or model the transfer function of an opto-electronic system. Within prysm there is a class `Convolvable` which objects and PSFs inherit from. You should rarely need to use the base class, except when subclassing it with your own models or objects.

```
>>> from prysm.convolution import Convolvable
```

The built-in convolvable objects are Slits, Pinholes, Tilted Squares, and Siemens Stars. There are also two components, PixelAperture and OLPF, used for system modeling.

```
>>> from prysm import Slit, Pinhole, TiltedSquare, SiemensStar, PixelAperture, OLPF
```

Each is initialized with object-specific parameters,

```
>>> s = Slit(width=1, orientation='crossed')  # diameter, um
>>> p = Pinhole(width=1)
>>> t = TiltedSquare(angle=8, background='white', sample_spacing=0.05, samples=256))
↪# degrees
>>> star = SiemensStar(num_spokes=32, sinusoidal=False, background='white', sample_
↪spacing=0.05, samples=256)
>>> pa = PixelAperture(width_x=5)  # diameter, um
>>> ol = OLPF(width_x=5*0.66)
```

Objects that take a background parameter will be black-on-white for `background=white`, or white-on-black for `background=black`. Two objects are convolved via the `conv` method, which returns `self` on a new `Convolvable` instance and is chainable,

```
>>> monstrosity = s.conv(p).conv(t).conv(star).conv(pa).conv(ol)
```

Some models require sample spacing and samples parameters while others do not. This is because prysm has many methods of executing an FFT-based Fourier domain convolution under the hood. If an object has a known analytical Fourier transform, the class has a method `(Convolvable).analytic_ft` which has abscissa units of reciprocal microns. If the analytic FT is present, it is used in lieu of numerical data. Models that have analytical Fourier transforms also accept sample_spacing and samples parameters, which are used to define a grid in the spatial domain. If two objects with analytical Fourier transforms are convolved, the output grid will have the finer sample spacing of the two inputs, and the larger span or window width of the two inputs.

The Convolvable constructor takes only four parameters,

```
>>> import numpy as np
>>> x = y = np.linspace(-20,20,256)
>>> z = np.random.uniform((256,256))
>>> c = Convolvable(data=z, unit_x=x, unit_y=y, has_analytic_ft=False)
```

`has_analytic_ft` has a default value of `False`.

Minimal labor is required to subclass `Convolvable`. For example, the `Pinhole` implemention is simply:

```
class Pinhole(Convolvable):
    def __init__(self, width, sample_spacing=0.025, samples=0):
        self.width = width

        # produce coordinate arrays
        if samples > 0:
            ext = samples / 2 * sample_spacing
            x, y = m.linspace(-ext, ext, samples), m.linspace(-ext, ext, samples)
            xv, yv = m.meshgrid(x, y)
            w = width / 2
            # paint a circle on a black background
            arr = m.zeros((samples, samples))
            arr[m.sqrt(xv**2 + yv**2) < w] = 1
        else:
            arr, x, y = None, m.zeros(2), m.zeros(2)

        super().__init__(data=arr, unit_x=x, unit_y=y, has_analytic_ft=True)

    def analytic_ft(self, unit_x, unit_y):
        xq, yq = m.meshgrid(unit_x, unit_y)
        # factor of pi corrects for jinc being modulo pi
        # factor of 2 converts radius to diameter
        rho = m.sqrt(xq**2 + yq**2) * self.width * 2 * m.pi
        return m.jinc(rho).astype(config.precision)
```

which is less than 20 lines long.

`Convolvable` objects have a few convenience properties and methods. `(Convolvable).slice_x` and its y variant exist and behave the same as slices on subclasses of `OpticalPhase` such as `Pupil`. `(Convolvable).plot_slice_xy` also works the same way. `(Convolvable).shape` is a convenience wrapper for `(Convolvable).data.shape`, an `(Convolvable).support_x`, `.support_y`, an `.support` mimic the equivalent `diameter` properties on `OpticalPhase` inheritants.

`(Convolvable).show` and `(Convolvable).show_fourier` behave the same way as `plot2d` methods found throughout prysm, except there are `xlim` and `ylim` parameters, which may be either single values, taken to be symmetric axis limits, or length-2 iterables of lower and upper limits.

Finally, `Convolvable` objects may be initialized from images,

```
>>> c = Convolvable.from_file(path_to_your_image, scale=1)  # plate scale in um
```

and written out as 8-bit images,

```
>>> p = 'foo.png'  # or jpg, any format imageio can handle
>>> c.save(save_path)
```

In practical use, one will generally only use the `conv`, `from_file`, and `save` methods with any degree of regularity. The complete API documentation is below. Attention should be paid to the docstring of `conv`, as it describes some of the details associated with convolutions in prysm, their accuracy, and when they are used.

---

class prysm.convolution.**Convolvable**(*data*, *unit_x*, *unit_y*, *has_analytic_ft=False*)

Bases: object

A base class for convolvable objects to inherit from.

data [*numpy.ndarray*] numerical representation of object

has_analytic_ft [*bool*] whether this convolvable has an analytical Fourier transform

sample_spacing [*float*] center to center spacing of samples

unit_x [*numpy.ndarray*] x-axis unit

unit_y [*numpy.ndarray*] y-axis unit

conv(*other*)

Convolves this convolvable with another.

other [*Convolvable*] A convolvable object

Convolvable a convolvable that lacks an analytical fourier transform

The algoithm works according to the following cases:

1. Both self and other have analytical fourier transforms: - The analytic forms will be used to compute the output directly. - The output sample spacing will be the finer of the two inputs. - The output window will cover the same extent as the "wider"

   input. If this window is not an integer number of samples wide, it will be enlarged symmetrically such that it is. This may mean the output array is not of the same size as either input.

   - An input which contains a sample at (0,0) may not produce an output with a sample at (0,0) if the input samplings are not favorable. To ensure this does not happen confirm that the inputs are computed over identical grids containing 0 to begin with.

2. One of self and other have analytical fourier transforms: - The input which does NOT have an analytical fourier transform

   will define the output grid.

   - The available analytic FT will be used to do the convolution in Fourier space.

3. Neither input has an analytic fourier transform: 3.1, the two convolvables have the same sample spacing to within

   a numerical precision of 0.1 nm:

- the fourier transform of both will be taken. If one has fewer samples, it will be upsampled in Fourier space

3.2, the two convolvables have different sample spacing:

- The fourier transform of both inputs will be taken. It is assumed that the more coarsely sampled signal is Nyquist sampled or better, and thus acts as a low-pass filter; the more finaly sampled input will be interpolated onto the same grid as the more coarsely sampled input. The higher frequency energy would be eliminated by multiplication with the Fourier spectrum of the more coarsely sampled input anyway.

The subroutines have the following properties with regard to accuracy:

1. Computes a perfect numerical representation of the continuous output, provided the output grid is capable of Nyquist sampling the result.
2. If the input that does not have an analytic FT is unaliased, computes a perfect numerical representation of the continuous output. If it does not, the input aliasing limits the output.
3. Accuracy of computation is dependent on how much energy is present at nyquist in the worse-sampled input; if this input is worse than Nyquist sampled, then the result will not be correct.

deconv(*other*, *balance=1000*, *reg=None*, *is_real=True*, *clip=False*, *postnormalize=True*)

Perform the deconvolution of this convolvable object by another.

other [*Convolvable*] another convolvable object, used as the PSF in a Wiener deconvolution

balance [*float*, optional] regularization parameter; passed through to skimage

reg [*numpy.ndarray*, optional] regularization operator, passed through to skimage

is_real [*bool*, optional] True if self and other are both real

clip [*bool*, optional] clips self and other into (0,1)

postnormalize [*bool*, optional] normalize the result such that it falls in [0,1]

Convolvable a new Convolable object

See skimage: http://scikit-image.org/docs/dev/api/skimage.restoration.html#skimage.restoration.wiener

static from_file(*path*, *scale*)

Read a monochrome 8 bit per pixel file into a new Image instance.

path [*string*] path to a file

scale [*float*] pixel scale, in microns

Convolvable a new image object

plot_slice_xy(*axlim=20*, *lw=3*, *fig=None*, *ax=None*)

Create a plot of slices through the X and Y axes of the *PSF*.

axlim [*float* or *int*, optional] axis limits, in microns

lw [*float*, optional] linewidth provided directly to matplotlib

fig [*matplotlib.figure.Figure*, optional] Figure to draw plot in

ax [*matplotlib.axes.Axis*] Axis to draw plot in

fig [*matplotlib.figure.Figure*, optional] Figure containing the plot

ax [*matplotlib.axes.Axis*, optional] Axis containing the plot

save(*path*, *nbits=8*)
Write the image to a png, jpg, tiff, etc.

path [*string*] path to write the image to

nbits [*int*] number of bits in the output image

shape

show(*xlim=None*, *ylim=None*, *interp_method=None*, *power=1*, *show_colorbar=True*, *fig=None*, *ax=None*)
Display the image.

xlim [iterable, optional] x axis limits

ylim [iterable,] y axis limits

interp_method [*string*] interpolation technique used in display

power [*float*] inverse of power to stretch image by. E.g. power=2 will plot img ** (1/2)

show_colorbar [*bool*] whether to show the colorbar or not.

fig [*matplotlib.figure.Figure*, optional:] Figure containing the plot

ax [*matplotlib.axes.Axis*, optional:] Axis containing the plot

fig [*matplotlib.figure.Figure*, optional:] Figure containing the plot

ax [*matplotlib.axes.Axis*, optional:] Axis containing the plot

show_fourier(*freq_x=None*, *freq_y=None*, *interp_method='lanczos'*, *fig=None*, *ax=None*)
Display the fourier transform of the image.

interp_method [*string*] method used to interpolate the data for display.

freq_x [iterable] x frequencies to use for convolvable with analytical FT and no data

freq_y [iterable] y frequencies to use for convolvable with analytic FT and no data

fig [*matplotlib.figure.Figure*] Figure containing the plot

ax [*matplotlib.axes.Axis*] Axis containing the plot

fig [*matplotlib.figure.Figure*] Figure containing the plot

ax [*matplotlib.axes.Axis*] Axis containing the plot

freq_x and freq_y are unused when the convolvable has a .data field.

slice_x
Retrieve a slice through the x axis of the PSF.

self.unit_x [*numpy.ndarray*] ordinate data

self.data [*numpy.ndarray*] coordinate data

slice_y
Retrieve a slice through the y axis of the PSF.

self.unit_y [*numpy.ndarray*] ordinate data

self.data [*numpy.ndarray*] coordinate data

support

support_x

support_y

4.2 Interferograms

Prysm offers rich features for analysis of interferometric data. `Interferogram` objects are conceptually similar to *Pupils* and both inherit from the same base class, as they both have to do with optical phase. The construction of an Interferogram requires only a few parameters:

```
>>> import numpy as np
>>> from prysm import Interferogram
>>> x = y = np.arange(129)
>>> z = np.random.uniform((128,128))
>>> interf = Interferogram(phase=z, intensity=None, unit_x=x, unit_y=y, scale='mm',␣
↪phase_unit='nm', meta={'wavelength': 632.8e-9})
```

Notable are the scale, and phase unit, which define the xy and z units, respectively. Any SI unit is accepted as well as angstroms. Imperial units not accepted. `meta` is a dictionary to store metadata. For several interferogram methods to work, the wavelength must be present *in meters.* This departure from prysm's convention of preferred units is used to maintain compatability with Zygo dat files. Interferograms are usually created from such files:

```
>>> interf = Interferogram.from_zygo_dat(your_path_file_object_or_bytes)
```

and both the dat and datx format from Zygo are supported. Dat carries no dependencies, while datx requries the installation of h5py. In addition to properties inherited from the `OpticalPhase` class (pv, rms, Sa, std), `Interferograms` have a `dropout_percentage` property, which gives the percentage of NaN values within the phase array. These NaNs may be filled,

```
>>> interf.fill(_with=0)
```

with 0 as a default value; only constants are supported. The modification is done in-place and the method returns `self`. Piston, tip-tilt, and power may be removed:

```
>>> interf.fill()\
>>>     .remove_piston()\
>>>     .remove_tiptilt()\
>>>     .remove_power()
```

again done in-place and returning self, so methods can be chained. One line convenience wrappers exist:

```
>>> interf.remove_piston_tiptilt()
>>> interf.remove_piston_tiptilt_power()
```

spikes may also be clipped,

```
>>> interf.spike_clip(nsigma=3)  # default is 3
```

setting points with a value more than nsigma standard deviations from the mean to `NaN`.

If the data did not have a lateral calibration baked into it, you can provide one in prysm,

```
>>> i.latcal(plate_scale=0.1, unit='mm')
>>> i.latcal(0.1, 'mm')  # these two invocations are equal
```

Masks may be applied:

```
>>> your_mask = np.ones(interf.phase.shape)
>>> interf.mask(your_mask)
>>> interf.mask('circle', diameter=100)  # 100 <spatial_unit> diameter circle
>>> interf.mask('hexagon', diameter=5)
```

The truecircle mask should not be used on interferometric data. the phase is deleted (replaced with NaN) wherever the mask is equal to zero.

Interferograms may be cropped, deleting empty (NaN) regions around a measurment;

```
>>> interf.crop()
```

Convenience properties are provided for data size,

```
>>> interf.shape, interf.diameter_x, interf.diameter_y, interf.diameter, interf.
↪semidiameter
```

`shape` mirrors the shape of the underlying ndarray. The x and y diameters are in units of `interf.spatial_unit` and `diameter` is the greater of the two.

The two dimensional Power Spectral Density (PSD) may be computed. The data may not contain NaNs, and piston tip and tilt should be removed prior. A 2D Welch window is used, so there is no need for concern about zero values creating a discontinuity at the edge of circular or other nonrectangular apertures.

```
>>> interf.crop().remove_piston_tiptilt_power().fill()
>>> ux, uy, psd = interf.psd()
```

x, y, and azimuthally averaged cuts of the PSD are also available

```
>>> psd_dict = interf.psd_xy_avg()
>>> ux, psd_x = psd_dict['x']
>>> uy, psd_y = psd_dict['y']
>>> ur, psd_r = psd_Dict['avg']
```

and the PSD may be plotted,

```
>>> interf.plot_psd2d(axlim=1, interp_method='lanczos', fig=None, ax=None)
>>> interf.plot_psd_xyavg(xlim=(1e0,1e3), ylim=(1e-7,1e2), fig=None, ax=None)
```

For the x/y and average plot, a Lorentzian model may be plotted alongside the data for e.g. visual verification of a requirement:

```
>>> interf.plot_psd_xyavg(a=1,b=1,c=1)
```

A bandlimited RMS value derived from the 2D PSD may also be evaluated,

```
>>> interf.bandlimited_rms(wllow=1, wlhigh=10, flow=1, fhigh=10)
```

only one of wavelength (wl; spatial period) or frequency (f) should be provided. f will overrule wavelength.

The complete API documentation is below.

---

class prysm.interferogram.**Interferogram**(*phase*, *intensity=None*, *x=None*, *y=None*, *scale='px'*, *phase_unit='nm'*, *meta=None*)

Bases: `prysm._phase.OpticalPhase`

Class containing logic and data for working with interferometric data.

bandlimited_rms (*wllow=None*, *wlhigh=None*, *flow=None*, *fhigh=None*)
Calculate the bandlimited RMS of a signal from its PSD.

wllow [*float*] short spatial scale

wlhigh [*float*] long spatial scale

flow [*float*] low frequency

fhigh [*float*] high frequency

float band-limited RMS value.

crop ()
Crop data to rectangle bounding non-NaN region.

dropout_percentage
Percentage of pixels in the data that are invalid (NaN).

fill (*_with=0*)
Fill invalid (NaN) values.

_with [*float*, optional] value to fill with

Interferogram self

static from_zygo_dat (*path*, *multi_intensity_action='first'*, *scale='mm'*)
Create a new interferogram from a zygo dat file.

path [path_like] path to a zygo dat file

multi_intensity_action [str, optional] see *io.read_zygo_dat*

scale [*str*, optional, {'um', 'mm'}] what xy scale to label the data with, microns or mm

Interferogram new Interferogram instance

latcal (*plate_scale*, *unit='mm'*)
Perform lateral calibration.

This probably won't do what you want if your data already has spatial units of anything but pixels (px).

plate_scale [*float*] center-to-center sample spacing of pixels, in (unit)s.

unit [*str*, optional] unit associated with the plate scale.

self modified *Interferogram* instance.

mask (*shape=None*, *diameter=0*, *mask=None*)
Mask the signal.

The mask will be inscribed in the axis with fewer pixels. I.e., for a interferogram with 1280x1000 pixels, the mask will be 1000x1000 at largest.

shape [*str*] valid shape from prysm.geometry

diameter [*float*] diameter of the mask, in self.spatial_units

mask [*numpy.ndarray*] user-provided mask

self modified Interferogram instance.

plot_psd2d(*axlim=None*, *clim=(1e-09*, *100.0)*, *interp_method='lanczos'*, *fig=None*, *ax=None*)
Plot the two dimensional PSD.

axlim [*float*, optional] symmetrical axis limit

power [*float*, optional] inverse of power to stretch image by

interp_method [*str*, optional] method used to interpolate the image, passed directly to matplotlib imshow

fig [*matplotlib.figure.Figure*] Figure containing the plot

ax [*matplotlib.axes.Axis*] Axis containing the plot

fig [*matplotlib.figure.Figure*] Figure containing the plot

ax [*matplotlib.axes.Axis*] Axis containing the plot

plot_psd_xy_avg(*a=None*, *b=None*, *c=None*, *lw=3*, *xlim=None*, *ylim=None*, *fig=None*, *ax=None*)
Plot the x, y, and average PSD on a linear x axis.

a [*float*, optional] a coefficient of Lorentzian PSD model plotted alongside data

b [*float*, optional] b coefficient of Lorentzian PSD model plotted alongside data

c [*float*, optional] c coefficient of Lorentzian PSD model plotted alongside data

lw [*float*, optional] linewidth provided directly to matplotlib

xlim [*tuple*, optional] len 2 tuple of low, high x axis limits

ylim [*tuple*, optional] len 2 tuple of low, high y axis limits

fig [*matplotlib.figure.Figure*] Figure containing the plot

ax [*matplotlib.axes.Axis*] Axis containing the plot

fig [*matplotlib.figure.Figure*] Figure containing the plot

ax [*matplotlib.axes.Axis*] Axis containing the plot

if a, b given but not c, an AB / inverse power model will be used for the PSD. If a, b, c are given the Lorentzian model will be used.

psd()
Power spectral density of the data., units (self.phase_unit^2)/((cy/self.spatial_unit)^2).

unit_x [*numpy.ndarray*] ordinate x frequency axis

unit_y [*numpy.ndarray*] ordinate y frequency axis

psd [*numpy.ndarray*] power spectral density

psd_xy_avg()
Power spectral density of the data., units (self.phase_unit^2)/((cy/self.spatial_unit)^2).

dict with keys x, y, avg. Each containing a tuple of (unit, psd)

pvr
Peak-to-Valley residual.

See: C. Evans, "Robust Estimation of PV for Optical Surface Specification and Testing" in Optical Fabrication and Testing, OSA Technical Digest (CD) (Optical Society of America, 2008), paper OWA4. http://www.opticsinfobase.org/abstract.cfm?URI=OFT-2008-OWA4

remove_piston()
Remove piston from the data by subtracting the mean value.

remove_piston_tiptilt()
Remove piston/tip/tilt from the data, see remove_tiptilt and remove_piston.

remove_piston_tiptilt_power()
Remove piston/tip/tilt/power from the data.

remove_power()
Remove power from the data by least squares fitting.

remove_tiptilt()
Remove tip/tilt from the data by least squares fitting and subtracting a plane.

spike_clip(*nsigma=3*)
Clip points in the data that exceed a certain multiple of the standard deviation.

nsigma [*float*] number of standard deviations to keep

self this Interferogram instance.

total_integrated_scatter(*wavelength*, *incident_angle=0*)
Calculate the total integrated scatter (TIS) for an angle or angles.

wavelength [*float*] wavelength of light in microns.

incident_angle [*float* or *numpy.ndarray*] incident angle(s) of light.

float* or *numpy.ndarray TIS value.

4.3 MTFs

Prysm models often include analysis of Modulation Transfer Function (MTF) data. The MTF is formally defined as:

> the normalized magnitude of the Fourier transform of the point spread function

It is nothing more and nothing less. It may not be negative, complex-valued, or equal to any value other than unity at the origin.

Initializing an MTF model should feel similar to a *PSF*,

```
>>> import numpy as np
>>> from prysm import MTF
>>> x = y = 1/np.linspace(-1,1,128)
>>> z = np.random.random((128,128))
>>> mt = MTF(data=z, unit_x=x, unit_y=y)
```

MTFs are usually created from a PSF instance

```
>>> mt = MTF.from_psf(your_psf_instance)
```

If modeling the MTF directly from a pupil plane, the intermediate PSF plane may be skipped;

```
>>> mt = MTF.from_pupil(your_pupil_instance, Q=2, efl=2)
```

Much like a PSF or other Convolvable, MTFs have quick-access slices

```
>>> print(mt.tan)
    (array([...]), array([...]))
```

```
>>> print(mt.sag)
    (array([...]), array([...]))
```

The tangential MTF is a slice through the x=0 axis, and assumes the usual optics sign convention of an object extended in y. The sagittal MTF is a slice through the y=0 axis.

The MTF at exact frequencies may be queried through any of the following methods: `(MTF).exact_polar`, takes arguments of freqs and azimuths. If there is a single frequency and multiple azimuths, the MTF at each azimuth and and the specified radial spatial frequency will be returned. The reverse is true for a single azimuth and multiple frequencies. `(MTF).exact_xy` follows the same semantics, but with Cartesian coordinates instead of polar. `(MTF).exact_tan` and `(MTF).exact_sag` both take a single argument of freq, which may be an int, float, or ndarray.

Finally, MTFs may be plotted:

```
>>> mt.plot_tan_sag(max_freq=200, fig=None, ax=None, labels=('Tangential', 'Sagittal
↪'))
>>> mt.plot2d(max_freq=200, power=1, fig=None, ax=None)
```

all arguments have these default values. The axes of plot2d will span (-max_freq, max_freq) on both x and y.

The complete API documentation is below.

---

class prysm.otf.**MTF**(*data*, *unit_x*, *unit_y=None*)

Bases: object

Modulation Transfer Function.

tan: slice along the X axis of the MTF object.

sag: slice along the Y axis of the MTF object.

center_x [*int*] x center pixel location (MTF == 1 here by definition)

center_y [*int*] y center pixel location (MTF == 1 here by definition)

data [*numpy.ndarray*] MTF data array, 2D

interpf_2d [*scipy.interpolate.RegularGridInterpolator*] 2D interpolation function used to compute exact values of the 2D MTF

interpf_sag [*scipy.interpolate.interp1d*] 1D interpolation function used to compute exact values of the sagittal MTF

interpf_tan [*scipy.interpolate.interp1d*] 1D interpolation function used to compute exact values of the tangential MTF

unit_x [*numpy.ndarray*] ordinate x coordinates

unit_y [TYPE] ordinate y coordinates

azimuthal_average()

Return the azimuthally averaged MTF.

nu [*numpy.ndarray*] spatial frequencies

mtf [*numpy.ndarray*] mtf values

exact_polar(*freqs*, *azimuths=None*)

Retrieve the MTF at the specified frequency-azimuth pairs.

freqs [iterable] radial frequencies to retrieve MTF for

azimuths [iterable] corresponding azimuths to retrieve MTF for

list MTF at the given points

exact_sag (*freq*)
Return data at an exact y coordinate along the x=0 axis.

freq [*number* or *numpy.ndarray*] frequency or frequencies to return

numpy.ndarray ndarray of MTF values

exact_tan (*freq*)
Return data at an exact x coordinate along the y=0 axis.

freq [*number* or *numpy.ndarray*] frequency or frequencies to return

numpy.ndarray ndarray of MTF values

exact_xy (*x*, *y=None*)
Retrieve the MTF at the specified X-Y frequency pairs.

x [iterable] X frequencies to retrieve the MTF at

y [iterable] Y frequencies to retrieve the MTF at

list MTF at the given points

static from_psf (*psf*)
Generate an MTF from a PSF.

psf [*PSF*] PSF to compute an MTF from

MTF A new MTF instance

static from_pupil (*pupil*, *efl*, *Q=2*)
Generate an MTF from a pupil, given a focal length (propagation distance).

pupil [*Pupil*] A pupil to propagate to a PSF, and convert to an MTF

efl [*float*] Effective focal length or propagation distance of the wavefunction

Q [*number*] ratio of pupil sample count to PSF sample count. Q > 2 satisfies nyquist

MTF A new MTF instance

plot2d (*max_freq=200*, *power=1*, *fig=None*, *ax=None*)
Create a 2D plot of the MTF.

max_freq [*float*] Maximum frequency to plot to. Axis limits will be ((-max_freq, max_freq), (-max_freq, max_freq)).

power [*float*] inverse of power to stretch the MTF to/by, e.g. power=2 will plot MTF^(1/2)

fig [*matplotlib.figure.Figure*, optional:] Figure to draw plot in

ax [*matplotlib.axes.Axis*, optional:] Axis to draw plot in

fig [*matplotlib.figure.Figure*] Figure to draw plot in

ax [*matplotlib.axes.Axis*] Axis to draw plot in

plot_tan_sag(*max_freq=200*, *fig=None*, *ax=None*, *labels=('Tangential', 'Sagittal')*)
Create a plot of the tangential and sagittal MTF.

max_freq [*float*] Maximum frequency to plot to. Axis limits will be ((-max_freq, max_freq), (-max_freq, max_freq))

fig [*matplotlib.figure.Figure*, optional:] Figure to draw plot in

ax [*matplotlib.axes.Axis*, optional:] Axis to draw plot in

labels [*iterable*] set of labels for the two lines that will be plotted

fig [*matplotlib.figure.Figure*] Figure to draw plot in

ax [*matplotlib.axes.Axis*] Axis to draw plot in

sag
Retrieve the sagittal MTF.

Assumes the object is extended in y. If the object is extended along a different azimuth, this will not return the sagittal MTF.

self.unit_x [*numpy.ndarray*] ordinate

self.data [*numpy.ndarray*] coordiante

tan
Retrieve the tangential MTF.

Assumes the object is extended in y. If the object is extended along a different azimuth, this will not return the tangential MTF.

self.unit_x [*numpy.ndarray*] ordinate

self.data [*numpy.ndarray*] coordiante

4.4 PSFs

PSFs in prysm have a very simple constructor;

```
>>> import numpy as np
>>> from prysm import PSF
>>> x = y = np.linspace(-1,1,128)
>>> z = np.random.random((128,128))
>>> ps = PSF(data=z, unit_x=x, unit_y=y)
```

PSFs are usually created from a *pupil* instance in a model, but the constructor can be used with e.g. experimental data.

```
>>> ps = PSF.from_pupil(your_pupil)
```

The encircled energy can be computed, for either a single point or an iterable (tuple, list, numpy array, ...) of points;

```
>>> print(ps.encircled_energy(0.1), ps.encircled_energy([0.1, 0.2, 0.3])
12.309576159990891, array([12.30957616, 24.61581586, 36.92244558])
```

encircled energy is computed via the method described in V Baliga, B D Cohn, *"Simplified Method For Calculating Encircled Energy,"* Proc. SPIE 0892, *Simulation and Modeling of Optical Systems*, (9 June 1988).

The inverse can also be computed using the L-BFGS-B nonlinear optimization routine, but the wavelength and F/# must be provided for the initial guess,

```
>>> ps.wavelength, ps.fno = 0.5, 2
>>> print(ps.ee_radius(1))
0.008104694339936169
```

Baliga's method is relatively slow for large arrays, so a dictionary is kept of all computed encircled energies at `ps._ee`. The encircled energy can be plotted. An axis limit must be provided if no encircled energy values have been computed. If some have, by default prysm will plot the computed values if no axis limit is given

```
>>> ps.plot_encircled_energy()
```

or

```
>>> ps.plot_encircled_energy(axlim=1, npts=50)
```

The PSF can be plotted in 2D,

```
>>> # ~0.838, exact value of energy contained in first airy zero
>>> from prysm.psf import FIRST_AIRY_ENCIRCLED
```

```
>>> ps.plot2d(axlim=0.8, power=2, interp_method='sinc', pix_grid=0.2, show_
↪axlabels=True, show_colorbar=True, circle_ee=FIRST_AIRY_ENCIRCLED)
```

Both `plot_encircled_energy` and `plot2d` take the usual `fig` and `ax` kwargs as well. For plot2d, the axlim arg sets the x and y axis limits to symmetrical values of `axlim`, i.e. the limits above will be [0.8, 0.8], [0.8, 0.8]. `power` controls the stretch of the color scale. The image will be stretched by the 1/power power, e.g. 2 plots psf^(1/2). `interp_method` is passed to matplotlib. `pix_grid` will use the minor axis ticks to draw a light grid over the PSF, intended to show the size of a PSF relative to the pixels of a detector. Units of microns. `show_axlabels` and `show_colorbar` both default to True, and control whether the axis labels are set and if the colorbar is drawn. `circle_ee` will draw a dashed circle at the radius containing the specified portion of the energy, and another at the diffraction limited radius for a circular aperture.

PSFs are a subclass of `Convolvable` and inherit all methods and attributes. The complete documentation is below.

---

class prysm.psf.**PSF**(*data*, *unit_x*, *unit_y*)

Bases: *prysm.convolution.Convolvable*

A Point Spread Function.

center_x [*int*] center sample along x

center_y [*int*] center sample along y

data [*numpy.ndarray*] PSF normalized intensity data

sample_spacing [*float*] center to center spacing of samples

unit_x [*numpy.ndarray*] x Cartesian axis locations of samples, 1D ndarray

unit_y ***numpy.ndarray*** y Cartesian axis locations of samples, 1D ndarray

ee_radius(*energy=0.8377850436212378*)

ee_radius_diffraction(*energy=0.8377850436212378*)

ee_radius_ratio_to_diffraction(*energy=0.8377850436212378*)

encircled_energy(*radius*)

Compute the encircled energy of the PSF.

radius [*float* or iterable] radius or radii to evaluate encircled energy at

encircled energy if radius is a float, returns a float, else returns a list.

implementation of "Simplified Method for Calculating Encircled Energy," Baliga, J. V. and Cohn, B. D., doi: 10.1117/12.944334

static from_pupil(*pupil*, *efl*, *Q=2*)
Use scalar diffraction propogation to generate a PSF from a pupil.

pupil [*Pupil*] Pupil, with OPD data and wavefunction

efl [*int* or *float*] effective focal length of the optical system

Q [*int* or *float*] ratio of pupil sample count to PSF sample count; Q > 2 satisfies nyquist

PSF A new PSF instance

plot2d(*axlim=25*, *power=1*, *clim=(None*, *None)*, *interp_method='lanczos'*, *pix_grid=None*, *invert=False*, *fig=None*, *ax=None*, *show_axlabels=True*, *show_colorbar=True*, *circle_ee=None*, *circle_ee_lw=None*)
Create a 2D plot of the PSF.

axlim [*float*] limits of axis, symmetric. xlim=(-axlim,axlim), ylim=(-axlim, axlim)

power [*float*] power to stretch the data by for plotting

clim [iterable] limits to use for log color scaling. If power != 1 and clim != (None, None), clim (log axes) takes precedence

interp_method [*string*] method used to interpolate the image between samples of the PSF

pix_grid [*float*] if not None, overlays gridlines with spacing equal to pix_grid. Intended to show the collection into camera pixels while still in the oversampled domain

invert [*bool*, optional] whether to invert the color scale

fig [*matplotlib.figure.Figure*, optional:] Figure containing the plot

ax [*matplotlib.axes.Axis*, optional:] Axis containing the plot

show_axlabels [*bool*] whether or not to show the axis labels

show_colorbar [*bool*] whether or not to show the colorbar

circle_ee [*float*, optional] relative encircled energy to draw a circle at, in addition to diffraction limited airy radius (1.22*λ*F#). First airy zero occurs at circle_ee=0.8377850436212378

circle_ee_lw [*float*, optional] linewidth passed to matplotlib for the encircled energy circles

fig [*matplotlib.figure.Figure*, optional] Figure containing the plot

ax [*matplotlib.axes.Axis*, optional] Axis containing the plot

plot_encircled_energy(*axlim=None*, *npts=50*, *lw=3*, *fig=None*, *ax=None*)
Make a 1D plot of the encircled energy at the given azimuth.

azimuth [*float*] azimuth to plot at, in degrees

axlim [*float*] limits of axis, will plot [0, axlim]

npts [*int*, optional] number of points to use from [0, axlim]

lw [*float*, optional] linewidth provided directly to matplotlib

fig [*matplotlib.figure.Figure*, optional] Figure containing the plot

ax [*matplotlib.axes.Axis*, optional:] Axis containing the plot

fig [*matplotlib.figure.Figure*, optional] Figure containing the plot

ax [*matplotlib.axes.Axis*, optional:] Axis containing the plot

4.5 Pupils

Most any physical optics model begins with a description of a wave at a pupil plane. This page will cover the core functionality of pupils; each analytical variety has its own documentation.

All *Pupil* parameters have default values, so one may be created with no arguments;

```
>>> from prysm import Pupil
>>> p = Pupil()
```

Pupils will be modeled using square arrays, the shape of which is controlled by a *samples* argument. They also accept an *epd* argument which controls their diameter in mm, as well as a *wavelength* which sets the wavelength of light used in microns. There is also an *opd_unit* argument that tells prysm what units are used to describe the *phase* associated with the pupil. Finally, a *mask* may be specified, either as a string using prysm's built-in masking capabilities, or as an array of the same shape as the pupil. Putting it all together,

```
>>> p = Pupil(samples=123, epd=456.7, wavelength=1.0, opd_unit='nm', mask='dodecagon')
```

p is a pupil with a 12-sided aperture backed by a 123x123 array which spans 456.7 mm and is impinged on by light of wavelength 1 micron.

Pupils have some more advanced parameters. *mask_target* determines if the phase ('phase'), wavefunction ('fcn'), or both ('both') will be masked. When embedding prysm in a task that repeatedly creates pupils, e.g. an optimizer for wavefront sensing, applying the mask to the phase is wasted computation and can be avoided.

If you wish to provide your own data for a pupil model, simply provide the *ux*, *uy*, and *phase* arguments, which are the x and y unit axes of shape (n,) and (m,), and *phase* is in units of opd_unit and of shape (m,n).

```
>>> p = Pupil(ux=x, uy=y, phase=phase, opd_unit='um')
```

Once a pupil is created, you can access quick access slices,

```
>>> p.slice_x  # returns tuple of unit, slice_of_phase
>>> p.slice_y
```

or evaluate the wavefront,

```
>>> p.pv, p.rms # in units of opd_unit
```

or Strehl ratio under the approximation given in *[Welford]*,

```
>>> p.strehl  #  [0,1]
```

The pupil may also be plotted. Plotting functions have defaults for all arguments, but may be overriden

```
>>> p.plot2d(cmap='RdYlBu', clim=(-100,100), interp_method='sinc')
```

cmap, *clim* and *interp_method* are passed directly to matplotlib. A figure and axis may also be provided if you would like to control these, for e.g. making a figure with multiple axes for different stages of the model

```
>>> from matplotlib import pyplot as plt
>>> fig, ax = plt.subplots(figsize=(10,10))
>>> p.plot2d(fig=fig, ax=ax)
```

A synthetic interferogram may be generated,

```
>>> fig, ax = plt.subplots(figsize=(4,4))
>>> p.interferogram(passes=2, visibility=0.5, fig=fig, ax=ax)
```

Pupils also support addition and subtraction with other pupil objects,

```
>>> p2 = p + p - p
```

The complete API documentation is below.

---

class prysm.pupil.**Pupil**(*samples=128*, *dia=1.0*, *wavelength=0.55*, *opd_unit='waves'*, *mask='circle'*, *mask_target='both'*, *ux=None*, *uy=None*, *phase=None*)

Bases: prysm._phase.OpticalPhase

Pupil of an optical system.

slice_x: ***numpy.ndarray*** slice through the x axis of the pupil. Returns (x,y) data where x is the sample coordinate and y is the phase.

slice_y: ***numpy.ndarray*** slice through the y axis of the pupil. Returns (x,y) data where x is the sample coordinate and y is the phase.

pv: ***float*** Peak-To-Valley wavefront error.

rms: ***float*** Root Mean Square wavefront error.

Sa: ***float*** Sa wavefront error.

subclasses should implement a build() method and their own way of expressing OPD.

center [*int*] index of the center sample, may be sheared by 1/2 for even sample counts

dia [*float*] entrance pupil diameter, mm

fcn [*numpy.ndarray*] wavefunction, complex 2D array

opd_unit [*str*] unit used to m.express phase errors

phase [*numpy.ndarray*] phase, real 2D array

rho [*numpy.ndarray*] radial ordinate axis, normalized units

sample_spacing [*float*] spacing of samples, mm

samples [*int*] number of samples across the pupil diameter

unit [*numpy.ndarray*] 1D array which gives the sample locations across the 2D pupil region

wavelength [*float*] wavelength of light, um

_mask [*numpy.ndarray*] mask used to define pupil extent and amplitude

mask_target [*str*] target for automatic masking on pupil creation

build()

Construct a numerical model of a *Pupil*.

The method should be overloaded by all subclasses to impart their unique mathematical models to the simulation.

Pupil this pupil instance

clone()

Create a copy of this pupil.

Pupil a deep copy duplicate of this pupil

fcn
Complex wavefunction associated with the pupil.

static from_interferogram(*interferogram*, *wvl=None*)
Create a new Pupil instance from an interferogram.

interferogram [*Interferogram*] an interferogram object

wvl [*float*, optional] wavelength of light, in micrometers, if not present in interferogram.meta

Pupil new Pupil instance

ValueError wavelength not present

mask(*mask*, *target*, *nanify=True*)
Apply a mask to the pupil.

Used to implement vignetting, chief ray angles, etc.

mask [*str* or *numpy.ndarray*] if a string, uses geometry.mcache for high speed access to a mask with a given shape, e.g. mask='circle' or mask='hexagon'. If an ndarray, directly use the mask.

target [*str*, {'phase', 'fcn', 'both'}] which array to mask

nanify: *bool*, optional if True, make (target) equal to NaN where the mask is zero.

Pupil self, the pupil instance

strehl
Strehl ratio of the pupil.

4.6 Zernikes

Prysm supports two flavors of Zernike polynomials; the Fringe set up to the 49th term, and the Zemax Standard set up to the 48th term. They have identical interfaces, so only one will be shown here.

Zernike notations are a subclass of `Pupil`, so they support the same arguments to `__init__`;

```
>>> from prysm import FringeZernike, StandardZernike
>>> p = FringeZernike(samples=123, epd=456.7, wavelength=1.0, opd_unit='nm', mask=
↪'dodecagon')
```

There are additional keyword arguments for each term, and the base (0 or 1) can be supplied. With base 0, the terms start at Z0 and range to Z48. With base 1, they start at Z1 and range to Z49 (or Z48, for Standard Zernikes). The Fringe set can also be used with unit RMS amplitude via the `rms_norm` keyword argument. Both notations also have nice print statements.

```
>>> p2 = FringeZernike(Z1=1, Z9=1, Z48=1, base=0, rms_norm=True)
>>> print(p2)
rms normalized Fringe Zernike description with:
        +1.000 Z1 - Tip (Y)
        +1.000 Z9 - Primary Trefoil Y
        +1.000 Z48 - Quinternary Spherical
        13.300 PV, 1.722 RMS
```

Notice that the RMS value is equal to sqrt(1^2 + 1^2 + 1^2) = sqrt(3) = 1.732 ~= 1.722. The difference of ~1% is due to the array sizes used by prysm by default, if increased, e.g. by adding `samples=1204` to the above `p2 = FringeZernike(...)` line, the value would converge to the analytical one.

A Zernike pupil can also be initalized with an iterable (list) of coefficients,

```
>>> import numpy as np
>>> terms = np.random.rand(49)  # 49 zernike coefficients, e.g. from a wavefront␣
↪sensor
>>> fz3 = FringeZernike(terms)
```

FringeZernike has many features StandardZernike does not. At the module level are two functions,

```
>>> from prysm.fringezernike import fzname, fzset_to_magnitude_angle
```

`fzname` takes an index and optional base (default 0) kwarg and returns the name of that term. `fzset_to_magnitude_angle` takes a non-sparse iterable of fringe zernike coefficients, starting with piston, and returns a dictionary with keywords of terms (e.g. "Primary Astigmatism") and items that are length 2 tuples of (magnitude, angle) where magnitude is in the same units as the zernike coefficients and the angle is in degrees. `fzset_to_magnitude_angle`'s output format can be seen below on the example of `FringeZernike.magnitudes`.

`FringeZernike` instances have a `truncate` method which discards terms with indices higher than n. For example,

```
>>> fz3.truncate(16)
```

this is less efficient, however, than simply slicing the coefficient vector,

```
>>> fz4 = FringeZernike(terms[:16])
```

and this slicing alternative should be used when one is sensitive to performance.

The top few terms may be extracted,

```
>>> fz4.top_n(5)
    [(0.9819642202790644, 5, 'Defocus'),
     (0.9591004803909723, 7, 'Primary Astigmatism 45°'),
     (0.9495975015239123, 28, 'Primary Pentafoil X'),
     (0.889828680566406, 8, 'Primary Coma Y'),
     (0.8831549729997366, 21, 'Secondary Trefoil X')]
```

or the terms listed by their pairwise magnitudes and clocking angles,

```
>>> fz4.magnitudes
    {'Piston': (0.7546621028832683, 90.0),
     'Tilt': (0.6603839752967117, 26.67150180654403),
     'Defocus': (0.14327809298942284, 90.0),
     'Primary Astigmatism': (0.7219964602989639, 85.19763587918983),
     'Primary Coma': (1.1351347586960325, 48.762867211696374),
     ...
     'Quinternary Spherical': (0.4974741523638292, 90.0)}
```

These things may be (bar) plotted;

```
>>> fz4.barplot(orientation='h', buffer=1, zorder=3)
>>> fz4.barplot_magnitudes(orientation='h', buffer=1, zorder=3)
>>> fz4.barplot_topn(n=5, orientation='h', buffer=1, zorder=3)
```

`orientation` controls the axis on which the terms are enumerated. `h` results in vertical bars, `v` is also accepted, as are `horizontal` and `vertical`. `buffer` is the number of terms' worth of spaces left on each side. The default of 1 leaves a one bar margin. `zorder` is passed to matplotlib – the default of 3 places the bars above any gridlines, which is an aesthetic choice. Matplotlib has a general default of 1.

If you would like direct access to the underlying functions, there are two paths. `prysm._zernike` contains functions for the first 49 (Fringe ordered) zernike polynomials, for example

```
>>> from prysm._zernike import defocus
```

each of these takes arguments of (rho, phi). They have names which end with _x, or `_00` and `_45` for the terms which have that naming convention.

Perhaps more convenient is a dictionary which numbers them,

```
>>> from prysm._zernike import _zernikes
>>> from prysm.fringezernike import zernmap
>>> zfunction = _zernikes[zernmap[8]]
>>> zfunction, zfunction.name, zfunction.norm
    <function prysm._zernike.primary_spherical(rho, phi)>, 'Primary Spherical', 2.
↪23606797749979
```

If these will always be evaluted over a square region enclosing the unit circle, a cache is available to speed computation,

```
>>> from prysm.fringezernike import fcache
>>> fcache(8, norm=True, samples=128)  # base 0
>>> fcache.clear()  # you should never have to do this unless you want to release␣
↪memory
```

This cache instance is used internally by prysm, if you modify the returned arrays in-place, you probably won't like the result. You can create your own independent instance,

```
>>> from prysm.fringezernike import FZCache
>>> my_fzcache = FZCache()
```

See *pupils* for information about general pupil functions. Below, the Fringe type of Zernike description has its full documentation printed.

---

class prysm.fringezernike.**FringeZernike**(**args*, ***kwargs*)

Bases: *prysm.pupil.Pupil*

barplot(*orientation='h'*, *buffer=1*, *zorder=3*, *fig=None*, *ax=None*)

Creates a barplot of coefficients and their names.

orientation [*str*, {'h', 'v', 'horizontal', 'vertical'}] orientation of the plot

buffer [*float*, optional] buffer to use around the left and right (or top and bottom) bars

zorder [*int*, optional] zorder of the bars. Use zorder > 3 to put bars in front of gridlines

fig [*matplotlib.figure.Figure*] Figure containing the plot

ax [*matplotlib.axes.Axis*] Axis containing the plot

fig [*matplotlib.figure.Figure*] Figure containing the plot

ax [*matplotlib.axes.Axis*] Axis containing the plot

barplot_magnitudes(*orientation='h'*, *sort=False*, *buffer=1*, *zorder=3*, *fig=None*, *ax=None*)
Create a barplot of magnitudes of coefficient pairs and their names.

E.g., astigmatism will get one bar.

orientation [*str*, {'h', 'v', 'horizontal', 'vertical'}] orientation of the plot

sort [*bool*, optional] whether to sort the zernikes in descending order

buffer [*float*, optional] buffer to use around the left and right (or top and bottom) bars

zorder [*int*, optional] zorder of the bars. Use zorder > 3 to put bars in front of gridlines

fig [*matplotlib.figure.Figure*] Figure containing the plot

ax [*matplotlib.axes.Axis*] Axis containing the plot

fig [*matplotlib.figure.Figure*] Figure containing the plot

ax [*matplotlib.axes.Axis*] Axis containing the plot

barplot_topn(*n=5*, *orientation='h'*, *buffer=1*, *zorder=3*, *fig=None*, *ax=None*)
Plot the top n terms in the wavefront.

n [*int*, optional] plot the top n terms.

orientation [*str*, {'h', 'v', 'horizontal', 'vertical'}] orientation of the plot

buffer [*float*, optional] buffer to use around the left and right (or top and bottom) bars

zorder [*int*, optional] zorder of the bars. Use zorder > 3 to put bars in front of gridlines

fig [*matplotlib.figure.Figure*] Figure containing the plot

ax [*matplotlib.axes.Axis*] Axis containing the plot

fig [*matplotlib.figure.Figure*] Figure containing the plot

ax [*matplotlib.axes.Axis*] Axis containing the plot

build()

Uses the wavefront coefficients stored in this class instance to build a wavefront model.

self.phase [*numpy.ndarray*] arrays containing the phase associated with the pupil

self.fcn [*numpy.ndarray*] array containing the wavefunction of the pupil plane

magnitudes
Returns the magnitude and angles of the zernike components in this wavefront.

top_n(*n=5*)
Identify the top n terms in the wavefront.

n [*int*, optional] identify the top n terms.

list list of tuples (magnitude, index, term)

truncate(*n*)
Truncate the wavefront to the first n terms.

n [*int*] number of terms to keep.

self modified FringeZernike instance.

truncate_topn(*n*)
Truncate the pupil to only the top n terms.

n [*int*] number of parameters to keep

self modified FringeZernike instance.

Bibliography

[Welford] 23. (t) Welford, Aberrations of Optical Systems. CRC Press, 1986.

Index

R

S

T