prysm Documentation

Release 0.15.1

Brandon Dube

Mar 18, 2019

Contents

Release 0.15.1

Date Mar 18, 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.

prysm is on pypi:

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

prysm requires only [numpy](http://www.numpy.org/) and [scipy](https://www.scipy.org/).

If your environment has [numba](http://numba.pydata.org/) installed, it will automatically accelerate many of prysm's compuations. To use an nVidia GPU, you must have [cupy](https://cupy.chainer.org/) installed. Plotting uses [matplotlib](https://matplotlib.org/). Images are read and written with [imageio](https://imageio.github.io/). Some MTF utilities utilize [pandas](https://pandas.pydata.org/). Reading of Zygo datx files requires [h5py](https://www.h5py.org/).

pip can be directed to install these,

```
>>> 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
>>> pip install prysm[deluxe]   # I want it all
```

or they may be installed at any time.

CHAPTER 2

Features

2.1 Physical Optics

- Modeing of pupil planes via or with:
-   - Fringe Zernike polynomials up to Z48, unit amplitude or RMS
-   - Noll ("Zemax Standard") Zernike polynomials up to Z36, unit amplitude or RMS
-   - apodization
-   - masks
-   -   * circles and ellipses
-   -   * n sided regular polygons
-   -   * user-provided
-   - synthetic fringe maps
-   - PV, RMS, stdev, Sa, 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
- – lowpass/highpass/bandpass/band-reject filtering
- – least-squares fitting and subtraction of Zernike modes, planes, and spheres
- – evaluation of PV, RMS, stdev, Sa, band-limited RMS, total integrated scatter
- – computation of PSD
- – ∗ 2D
- – ∗ x, y, azimuthally averaged slices
- – ∗ evaluation and/or comparison to ab (power law) or abc (Lorentzian) models
- – spike clipping
- – plotting

2.2 First-Order Optics

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

2.3 Parsing Data from Commercial & Open Source Instruments

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

CHAPTER 3

User's Guide

3.1 User's Guide

3.1.1 Conventions

Here, we will outline some of the conventions used by prysm. These will be useful to understand if extending the library, or performing custom analysis.

prysm uses a large number of classes which carry data and metadata about the signals with their namesakes. They can be divided loosely into two caregories,

- phases
- images

Both have common properties of `unit_x` and `unit_y`, which are one dimensional arrays giving the gridded coordinates in x and y.

[1]:
```
# an example of a phase-type object and an image-type object
from prysm import Pupil, Slit

pu = Pupil()
sl = Slit(1, sample_spacing=0.075, samples=64)

pu.unit_x[:3], sl.unit_y[:3]  # only first three elements for brevity
```

[1]:
```
(array([-0.5       , -0.49212598, -0.48425197]),
 array([-2.4       , -2.32380952, -2.24761905]))
```

Each has an array that holds the numerical representation of the signal itself, for phaes-type objects this is `phase` and for image-type objects this is `data`. The convention is `y,x` indices, consistent with numpy. This is the opposite convention used by matlab.

[2]:
```
pu.phase[:3,:3], sl.data[:3,:3]
```

[2]:
```
(array([[nan, nan, nan],
        [nan, nan, nan],
        [nan, nan, nan]]), array([[0., 0., 0.],
        [0., 0., 0.],
        [0., 0., 0.]]))
```

both inherit from BasicData (in fact, just about every class in prysm does) which imbues them with a brevy of properties:

[3]:
```
from prysm._basicdata import BasicData
help(BasicData)
```

```
Help on class BasicData in module prysm._basicdata:

class BasicData(builtins.object)
 |  Abstract base class holding some data properties.
 |
 |  Methods defined here:
 |
 |  copy(self)
 |      Return a (deep) copy of this instance).
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
 |
 |  center_x
 |      Center "pixel" in x.
 |
 |  center_y
 |      Center "pixel" in y.
 |
 |  sample_spacing
 |      center-to-center sample spacing.
 |
 |  samples_x
 |      Number of samples in the x dimension.
 |
 |  samples_y
 |      Number of samples in the y dimension.
 |
 |  shape
 |      Proxy to phase or data shape.
 |
 |  size
 |      Proxy to phase or data size.
 |
 |  slice_x
 |      Retrieve a slice through the X axis of the phase.
 |
 |      Returns
 |      -----
 |      self.unit : `numpy.ndarray`
```

(continues on next page)

(continued from previous page)

```
 |          ordinate axis
 |      slice of self.phase or self.data : `numpy.ndarray`
 |
 |  slice_y
 |      Retrieve a slice through the Y axis of the phase.
 |
 |      Returns
 |      -----
 |      self.unit : `numpy.ndarray`
 |          ordinate axis
 |      slice of self.phase or self.data : `numpy.ndarray`
```

prysm is a metadata-heavy library, with many functions and procedures taking a several arguments, most of which populated with sane default values. A number of these defaults can be controlled through prysm's config object,

```
[4]: from prysm import config
```

```
[5]: controlled_properties = [i for i in dir(config) if not i.startswith('_') and not i ==
     ↪'initialized']
     print(controlled_properties)
```

```
['Q', 'backend', 'chbackend_observers', 'image_colormap', 'lw', 'phase_colormap',␣
↪'precision', 'precision_complex', 'zernike_base', 'zorder']
```

To change the value used by prysm, simply assign to the property,

```
[6]: # use 32-bit floats instead of 64-bit, ~50% speedup to all operations in exchange for␣
     ↪accuracy
     config.precision = 32
```

3.1.2 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 or loading a source image.

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.

```
[1]: from prysm import Convolvable, Slit, Pinhole, TiltedSquare, SiemensStar,␣
     ↪PixelAperture, OLPF
```

```
[2]: 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(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,

[3]:
```
# 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 an `analytic_ft` method 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,

[4]:
```
import numpy as np
x = y = np.linspace(-20,20,256)
z = np.random.uniform(size=256**2).reshape(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:

[5]:
```
# need from prysm import mathops as m if you want to actually use this

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. `slice_x` and its y variant exist and behave the same as slices on subclasses of `OpticalPhase` such as `Pupil` and `Interferogram`. `plot_slice_xy` also works the same way. `shape` is a convenience wrapper for `.data.shape`, and `support_x`, `support_y`, and `support` mimic the equivalent diameter properties on `OpticalPhase`.

`show` and `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,

```
[6]: from prysm import sample_files
     c = Convolvable.from_file(sample_files('boat.png'), scale=5)  # plate scale in um --␣
     ↪5microns/pixel
     c.data /= 255  # when initializing from files, normalization is left to the user
     c.show()
```

```
[6]: (<Figure size 640x480 with 2 Axes>,
      <matplotlib.axes._subplots.AxesSubplot at 0x7f47e53fcdd8>)
```

and written out as 8 or 16-bit images,

```
[7]: p = 'foo.png'  # or jpg, any format imageio can handle
     c.save(p, nbits=16)
```

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.

```
[8]: help(c.conv)
```

```
Help on method conv in module prysm.convolution:

conv(other) method of prysm.convolution.Convolvable instance
    Convolves this convolvable with another.

    Parameters
    -------
    other : `Convolvable`
        A convolvable object

    Returns
    -----
    `Convolvable`
        a convolvable that lacks an analytical fourier transform

    Notes
    ---
    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
```

(continues on next page)

(continued from previous page)

```
                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.
```

[ ]:

```
```

3.1.3 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:

[1]:

```
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, x=x, y=y, scale='mm', phase_unit='nm',
↪ meta={'wavelength': 632.8e-9})
```

Notable are the scale, and phase unit, which define the x, y, and z units. 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:

[2]:

```
from prysm import sample_files

interf = Interferogram.from_zygo_dat(sample_files('dat'))
```

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 `PVr` property, for C. Evan's Robust PV metric, and `dropout_percentage` property, which gives the percentage of NaN values within the phase array. These NaNs may be filled,

```
[3]: interf.fill(_with=0)
```

```
[3]: <prysm.interferogram.Interferogram at 0x7f3f089f3630>
```

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:

```
[4]: interf.remove_piston().remove_tiptilt().remove_power()
```

```
[4]: <prysm.interferogram.Interferogram at 0x7f3f089f3630>
```

again done in-place and returning self, so methods can be chained. You should remove these terms (or, indeed do anything with Zernikes) before filling NaNs. One line convenience wrappers exist:

```
[5]: interf.remove_piston_tiptilt()
     interf.remove_piston_tiptilt_power()
```

```
[5]: <prysm.interferogram.Interferogram at 0x7f3f089f3630>
```

spikes may also be clipped,

```
[6]: interf.spike_clip(nsigma=3)  # default is 3
```

```
[6]: <prysm.interferogram.Interferogram at 0x7f3f089f3630>
```

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`,

```
[7]: # this is not going to do what you want if your data is already calibrated.
     interf.latcal(plate_scale=0.1, unit='mm')
```

```
[7]: <prysm.interferogram.Interferogram at 0x7f3f089f3630>
```

Masks may be applied:

```
[8]: your_mask = np.ones(interf.phase.shape)
     interf.mask(your_mask)
     interf.mask('circle', diameter=4)  # 4 <spatial_unit> diameter circle
```

```
[8]: <prysm.interferogram.Interferogram at 0x7f3f089f3630>
```

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;

```
[9]: interf.crop()
```

```
[9]: <prysm.interferogram.Interferogram at 0x7f3f089f3630>
```

Convenience properties are provided for data size,

```
[10]: interf.shape, interf.size, interf.diameter_x, interf.diameter_y, interf.diameter,␣
      ↪interf.semidiameter
```

```
[10]: ((339, 339),
       114921,
       3.9858380492660217,
       3.9858380492660217,
       3.9858380492660217,
       1.9929190246330108)
```

`shape` and `size` mirrors the underlying `interf.phase` 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 for circular data and a 2D Hanning window for rectangular data, so there is no need for concern about zero values creating a discontinuity at the edge of circular or other nonrectangular apertures.

```
[11]: interf.crop().remove_piston_tiptilt_power().fill()
      ux, uy, psd = interf.psd()
```

several slices are also available,

```
[12]: psd_dict = interf.psd_slices(x=True, y=True, azavg=True, azmin=True, azmax=True)
      ux, psd_x = psd_dict['x']
      uy, psd_y = psd_dict['y']
      ur, psd_r = psd_dict['azavg']
```

and the PSD may be plotted,

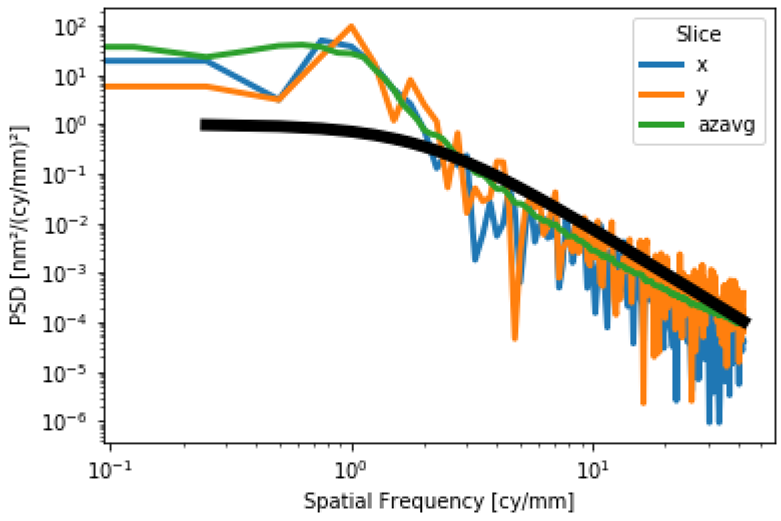
```
[13]: interf.plot_psd2d(axlim=1, interp_method='lanczos', fig=None, ax=None)
      interf.plot_psd_slices(xlim=(1e0,1e3), ylim=(1e-7,1e2), fig=None, ax=None)
```

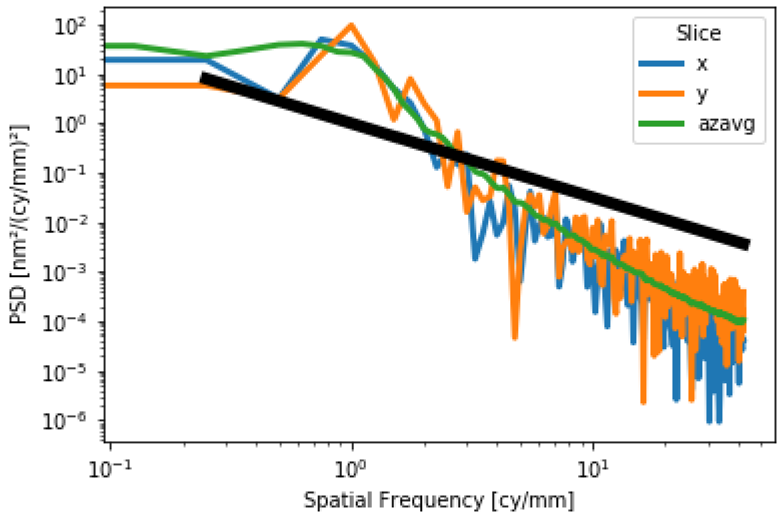
```
[13]: (<Figure size 640x480 with 1 Axes>,
       <matplotlib.axes._subplots.AxesSubplot at 0x7f3edb1f1240>)
```

For the slices plot, a Lorentzian or power law model may be plotted alongside the data for e.g. visual verification of a requirement:

```
[14]: interf.plot_psd_slices(a=1,b=2,c=3)
      interf.plot_psd_slices(a=1,b=1.5)
```

```
[14]: (<Figure size 432x288 with 1 Axes>,
       <matplotlib.axes._subplots.AxesSubplot at 0x7f3edafb17f0>)
```





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

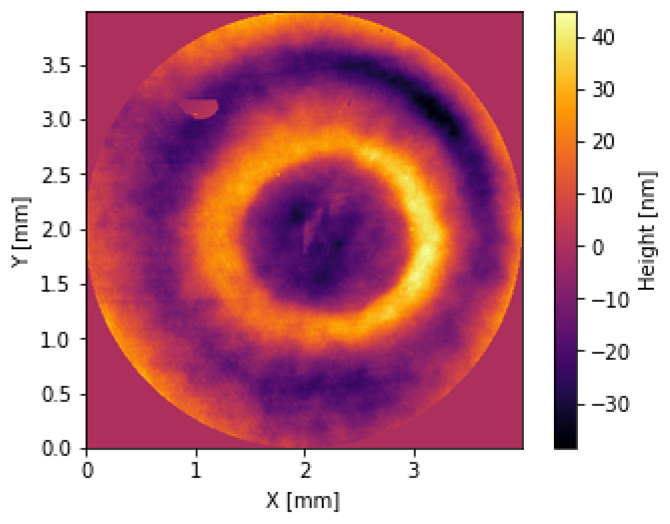
```
[15]: interf.bandlimited_rms(wllow=1, wlhigh=10, flow=1, fhigh=10)
```

```
[15]: 9.42102378723147
```

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

```
[16]: interf.plot2d()
```

```
[16]: (<Figure size 432x288 with 2 Axes>,
       <matplotlib.axes._subplots.AxesSubplot at 0x7f3edada8ac8>)
```



3.1.4 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*,

[1]:
```
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

[2]:
```
from prysm import Pupil, PSF
pu = Pupil(dia=10, wavelength=0.5)
ps = PSF.from_pupil(pu, efl=20)
mt = MTF.from_psf(ps)
```

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

[3]:
```
mt = MTF.from_pupil(pu, Q=2, efl=20)  # Q, efl same as PSF.from_pupil
```

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

[4]:
```
mt.tan, mt.sag
```

[4]:
```
((array([   0.        ,    7.87401575,   15.7480315 ,   23.62204724,
           31.49606299,   39.37007874,   47.24409449,   55.11811024,
           62.99212598,   70.86614173,   78.74015748,   86.61417323,
           94.48818898,  102.36220472,  110.23622047,  118.11023622,
          125.98425197,  133.85826772,  141.73228346,  149.60629921,
          157.48031496,  165.35433071,  173.22834646,  181.1023622 ,
```

(continues on next page)

(continued from previous page)

```
        188.97637795,  196.8503937 ,  204.72440945,  212.5984252 ,
        220.47244094,  228.34645669,  236.22047244,  244.09448819,
        251.96850394,  259.84251969,  267.71653543,  275.59055118,
        283.46456693,  291.33858268,  299.21259843,  307.08661417,
        314.96062992,  322.83464567,  330.70866142,  338.58267717,
        346.45669291,  354.33070866,  362.20472441,  370.07874016,
        377.95275591,  385.82677165,  393.7007874 ,  401.57480315,
        409.4488189 ,  417.32283465,  425.19685039,  433.07086614,
        440.94488189,  448.81889764,  456.69291339,  464.56692913,
        472.44094488,  480.31496063,  488.18897638,  496.06299213,
        503.93700787,  511.81102362,  519.68503937,  527.55905512,
        535.43307087,  543.30708661,  551.18110236,  559.05511811,
        566.92913386,  574.80314961,  582.67716535,  590.5511811 ,
        598.42519685,  606.2992126 ,  614.17322835,  622.04724409,
        629.92125984,  637.79527559,  645.66929134,  653.54330709,
        661.41732283,  669.29133858,  677.16535433,  685.03937008,
        692.91338583,  700.78740157,  708.66141732,  716.53543307,
        724.40944882,  732.28346457,  740.15748031,  748.03149606,
        755.90551181,  763.77952756,  771.65354331,  779.52755906,
        787.4015748 ,  795.27559055,  803.1496063 ,  811.02362205,
        818.8976378 ,  826.77165354,  834.64566929,  842.51968504,
        850.39370079,  858.26771654,  866.14173228,  874.01574803,
        881.88976378,  889.76377953,  897.63779528,  905.51181102,
        913.38582677,  921.25984252,  929.13385827,  937.00787402,
        944.88188976,  952.75590551,  960.62992126,  968.50393701,
        976.37795276,  984.2519685 ,  992.12598425, 1000.        ]),
 array([1.00000000e+00, 9.90034799e-01, 9.80069598e-01, 9.70104397e-01,
       9.60139196e-01, 9.50173996e-01, 9.40208795e-01, 9.30243594e-01,
       9.20278393e-01, 9.10313192e-01, 9.00347991e-01, 8.90382790e-01,
       8.80417589e-01, 8.70452388e-01, 8.60487188e-01, 8.50521987e-01,
       8.40556786e-01, 8.30591585e-01, 8.20626384e-01, 8.10661183e-01,
       8.00695982e-01, 7.90730781e-01, 7.80765581e-01, 7.70958557e-01,
       7.61151534e-01, 7.51344511e-01, 7.41537488e-01, 7.31730465e-01,
       7.21923442e-01, 7.12116419e-01, 7.02309396e-01, 6.92502373e-01,
       6.82695350e-01, 6.73046504e-01, 6.63397659e-01, 6.53748814e-01,
       6.44099968e-01, 6.34451123e-01, 6.24802278e-01, 6.15311610e-01,
       6.05820943e-01, 5.96330275e-01, 5.86839608e-01, 5.77348940e-01,
       5.67858273e-01, 5.58525783e-01, 5.49193293e-01, 5.39860804e-01,
       5.30528314e-01, 5.21195824e-01, 5.11863334e-01, 5.02689022e-01,
       4.93514711e-01, 4.84340399e-01, 4.75166087e-01, 4.66149953e-01,
       4.57133818e-01, 4.48117684e-01, 4.39101550e-01, 4.30243594e-01,
       4.21385637e-01, 4.12527681e-01, 4.03669725e-01, 3.94969946e-01,
       3.86270168e-01, 3.77570389e-01, 3.68870611e-01, 3.60329010e-01,
       3.51787409e-01, 3.43403986e-01, 3.35020563e-01, 3.26637140e-01,
       3.18253717e-01, 3.10028472e-01, 3.01803227e-01, 2.93736159e-01,
       2.85669092e-01, 2.77760202e-01, 2.69851313e-01, 2.61942423e-01,
       2.54033534e-01, 2.46282822e-01, 2.38532110e-01, 2.30939576e-01,
       2.23347042e-01, 2.15912686e-01, 2.08478330e-01, 2.01202151e-01,
       1.93925973e-01, 1.86807972e-01, 1.79689972e-01, 1.72730149e-01,
       1.65770326e-01, 1.58968681e-01, 1.52167036e-01, 1.45523568e-01,
       1.38880101e-01, 1.32394812e-01, 1.25909522e-01, 1.19582411e-01,
       1.13255299e-01, 1.07244543e-01, 1.01233787e-01, 9.53812085e-02,
       8.95286302e-02, 8.38342297e-02, 7.81398292e-02, 7.27617842e-02,
       6.73837393e-02, 6.21638722e-02, 5.69440051e-02, 5.20404935e-02,
       4.71369820e-02, 4.25498260e-02, 3.79626700e-02, 3.36918697e-02,
       2.94210693e-02, 2.54666245e-02, 2.15121797e-02, 1.80322683e-02,
       1.45523568e-02, 1.15469788e-02, 8.54160076e-03, 6.01075609e-03,
```

(continues on next page)

(continued from previous page)

```
        3.47991142e-03, 1.73995571e-03, 5.95103509e-17, 4.58025254e-17])),
 (array([   0.        ,    7.87401575,   15.7480315 ,   23.62204724,
          31.49606299,   39.37007874,   47.24409449,   55.11811024,
          62.99212598,   70.86614173,   78.74015748,   86.61417323,
          94.48818898,  102.36220472,  110.23622047,  118.11023622,
         125.98425197,  133.85826772,  141.73228346,  149.60629921,
         157.48031496,  165.35433071,  173.22834646,  181.1023622 ,
         188.97637795,  196.8503937 ,  204.72440945,  212.5984252 ,
         220.47244094,  228.34645669,  236.22047244,  244.09448819,
         251.96850394,  259.84251969,  267.71653543,  275.59055118,
         283.46456693,  291.33858268,  299.21259843,  307.08661417,
         314.96062992,  322.83464567,  330.70866142,  338.58267717,
         346.45669291,  354.33070866,  362.20472441,  370.07874016,
         377.95275591,  385.82677165,  393.7007874 ,  401.57480315,
         409.4488189 ,  417.32283465,  425.19685039,  433.07086614,
         440.94488189,  448.81889764,  456.69291339,  464.56692913,
         472.44094488,  480.31496063,  488.18897638,  496.06299213,
         503.93700787,  511.81102362,  519.68503937,  527.55905512,
         535.43307087,  543.30708661,  551.18110236,  559.05511811,
         566.92913386,  574.80314961,  582.67716535,  590.5511811 ,
         598.42519685,  606.2992126 ,  614.17322835,  622.04724409,
         629.92125984,  637.79527559,  645.66929134,  653.54330709,
         661.41732283,  669.29133858,  677.16535433,  685.03937008,
         692.91338583,  700.78740157,  708.66141732,  716.53543307,
         724.40944882,  732.28346457,  740.15748031,  748.03149606,
         755.90551181,  763.77952756,  771.65354331,  779.52755906,
         787.4015748 ,  795.27559055,  803.1496063 ,  811.02362205,
         818.8976378 ,  826.77165354,  834.64566929,  842.51968504,
         850.39370079,  858.26771654,  866.14173228,  874.01574803,
         881.88976378,  889.76377953,  897.63779528,  905.51181102,
         913.38582677,  921.25984252,  929.13385827,  937.00787402,
         944.88188976,  952.75590551,  960.62992126,  968.50393701,
         976.37795276,  984.2519685 ,  992.12598425, 1000.        ]),
  array([1.00000000e+00, 9.90034799e-01, 9.80069598e-01, 9.70104397e-01,
        9.60139196e-01, 9.50173996e-01, 9.40208795e-01, 9.30243594e-01,
        9.20278393e-01, 9.10313192e-01, 9.00347991e-01, 8.90382790e-01,
        8.80417589e-01, 8.70452388e-01, 8.60487188e-01, 8.50521987e-01,
        8.40556786e-01, 8.30591585e-01, 8.20626384e-01, 8.10661183e-01,
        8.00695982e-01, 7.90730781e-01, 7.80765581e-01, 7.70958557e-01,
        7.61151534e-01, 7.51344511e-01, 7.41537488e-01, 7.31730465e-01,
        7.21923442e-01, 7.12116419e-01, 7.02309396e-01, 6.92502373e-01,
        6.82695350e-01, 6.73046504e-01, 6.63397659e-01, 6.53748814e-01,
        6.44099968e-01, 6.34451123e-01, 6.24802278e-01, 6.15311610e-01,
        6.05820943e-01, 5.96330275e-01, 5.86839608e-01, 5.77348940e-01,
        5.67858273e-01, 5.58525783e-01, 5.49193293e-01, 5.39860804e-01,
        5.30528314e-01, 5.21195824e-01, 5.11863334e-01, 5.02689022e-01,
        4.93514711e-01, 4.84340399e-01, 4.75166087e-01, 4.66149953e-01,
        4.57133818e-01, 4.48117684e-01, 4.39101550e-01, 4.30243594e-01,
        4.21385637e-01, 4.12527681e-01, 4.03669725e-01, 3.94969946e-01,
        3.86270168e-01, 3.77570389e-01, 3.68870611e-01, 3.60329010e-01,
        3.51787409e-01, 3.43403986e-01, 3.35020563e-01, 3.26637140e-01,
        3.18253717e-01, 3.10028472e-01, 3.01803227e-01, 2.93736159e-01,
        2.85669092e-01, 2.77760202e-01, 2.69851313e-01, 2.61942423e-01,
        2.54033534e-01, 2.46282822e-01, 2.38532110e-01, 2.30939576e-01,
        2.23347042e-01, 2.15912686e-01, 2.08478330e-01, 2.01202151e-01,
        1.93925973e-01, 1.86807972e-01, 1.79689972e-01, 1.72730149e-01,
        1.65770326e-01, 1.58968681e-01, 1.52167036e-01, 1.45523568e-01,
```

(continues on next page)

(continued from previous page)

```
         1.38880101e-01, 1.32394812e-01, 1.25909522e-01, 1.19582411e-01,
         1.13255299e-01, 1.07244543e-01, 1.01233787e-01, 9.53812085e-02,
         8.95286302e-02, 8.38342297e-02, 7.81398292e-02, 7.27617842e-02,
         6.73837393e-02, 6.21638722e-02, 5.69440051e-02, 5.20404935e-02,
         4.71369820e-02, 4.25498260e-02, 3.79626700e-02, 3.36918697e-02,
         2.94210693e-02, 2.54666245e-02, 2.15121797e-02, 1.80322683e-02,
         1.45523568e-02, 1.15469788e-02, 8.54160076e-03, 6.01075609e-03,
         3.47991142e-03, 1.73995571e-03, 1.09720259e-17, 6.67479670e-17])))
```

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: `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. `exact_xy` follows the same semantics, but with Cartesian coordinates instead of polar. `exact_tan` and `exact_sag` both take a single argument of freq, which may be an int, float, or ndarray.

Finally, MTFs may be plotted:

```
[5]: mt.plot_tan_sag(max_freq=1000, fig=None, ax=None, labels=('Tangential', 'Sagittal'))
     mt.plot2d(max_freq=1000, power=2, fig=None, ax=None)
```

```
[5]: (<Figure size 640x480 with 2 Axes>,
      <matplotlib.axes._subplots.AxesSubplot at 0x7ff19a92bc50>)
```

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

This example should be familiar as the diffraction limited MTF of a circular aperture.

```
[ ]:
```

3.1.5 PSFs

PSFs in prysm have a very simple constructor,

```
[1]: 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.

```
[2]: from prysm import Pupil
     ps = PSF.from_pupil(Pupil(dia=10), efl=20)  # F/2
```

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

```
[3]: ps.encircled_energy(0.1), ps.encircled_energy([0.1, 0.2, 0.3])
```

```
[3]: (0.020147646693404343, array([0.02014765, 0.07817419, 0.16723156]))
```

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 a nonlinear optimization routine, provided the wavelength and F/# are known to support the initial guess.

```
[4]: ps.ee_radius(0.838)
```

```
[4]: 1.4138420278249217
```

Baliga's method is relatively slow for large arrays, so a dictionary is kept of all computed encircled energies at `ps._ee`.

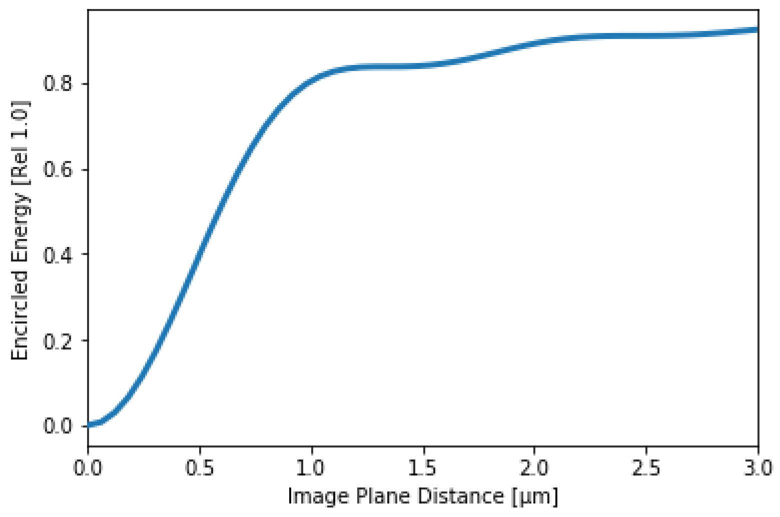
The encircled energy can be plotted. An axis limit must be provided if no encircled energy values have been computed. If some have, by default prysm will plot the computed values if no axis limit is given

```
[5]: ps.plot_encircled_energy()
```

```
[5]: (<Figure size 640x480 with 1 Axes>,
      <matplotlib.axes._subplots.AxesSubplot at 0x7f8b30a08240>)
```

```
[6]: ps.plot_encircled_energy(axlim=3, npts=50)
```

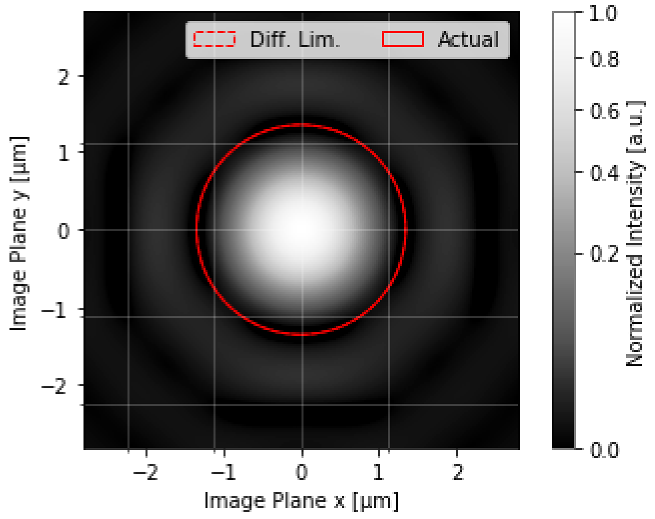
```
[6]: (<Figure size 432x288 with 1 Axes>,
      <matplotlib.axes._subplots.AxesSubplot at 0x7f8b30942518>)
```



The PSF can be plotted in 2D,

```
[7]: # ~0.838, exact value of energy contained in first airy zero
     from prysm.psf import FIRST_AIRY_ENCIRCLED
     ps.plot2d(axlim=5*1.22*ps.wavelength*FIRST_AIRY_ENCIRCLED,
               power=2,
               interp_method='lanczos',
               pix_grid=1.12,
               show_axlabels=True,
               show_colorbar=True,
               circle_ee=FIRST_AIRY_ENCIRCLED)
```

```
[7]: (<Figure size 432x288 with 2 Axes>,
      <matplotlib.axes._subplots.AxesSubplot at 0x7f8b30895d30>)
```



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.

```
[ ]:
```

3.1.6 Pupils

ost 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,

```
[1]: %matplotlib inline
     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 a `dia` argument which controls their diameter in mm, as well as `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,

```
[2]: p = Pupil(samples=123, dia=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 (`p.phase`), wavefunction (`p.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).

```
[3]: import numpy as np
     x, y, phase = np.linspace(-1,1,128), np.linspace(-1,1,128), np.random.rand(128,128)
     p = Pupil(ux=x, uy=y, phase=phase, opd_unit='um')

     # below examples will want something nicer...
     p = Pupil()
```

```
[4]: # returns tuple of unit, slice_of_phase
     p.slice_x # p.slice_y
```

```
[4]: (array([-0.5       , -0.49212598, -0.48425197, -0.47637795, -0.46850394,
             -0.46062992, -0.45275591, -0.44488189, -0.43700787, -0.42913386,
             -0.42125984, -0.41338583, -0.40551181, -0.3976378 , -0.38976378,
             -0.38188976, -0.37401575, -0.36614173, -0.35826772, -0.3503937 ,
             -0.34251969, -0.33464567, -0.32677165, -0.31889764, -0.31102362,
             -0.30314961, -0.29527559, -0.28740157, -0.27952756, -0.27165354,
             -0.26377953, -0.25590551, -0.2480315 , -0.24015748, -0.23228346,
             -0.22440945, -0.21653543, -0.20866142, -0.2007874 , -0.19291339,
             -0.18503937, -0.17716535, -0.16929134, -0.16141732, -0.15354331,
             -0.14566929, -0.13779528, -0.12992126, -0.12204724, -0.11417323,
             -0.10629921, -0.0984252 , -0.09055118, -0.08267717, -0.07480315,
             -0.06692913, -0.05905512, -0.0511811 , -0.04330709, -0.03543307,
             -0.02755906, -0.01968504, -0.01181102, -0.00393701,  0.00393701,
              0.01181102,  0.01968504,  0.02755906,  0.03543307,  0.04330709,
              0.0511811 ,  0.05905512,  0.06692913,  0.07480315,  0.08267717,
              0.09055118,  0.0984252 ,  0.10629921,  0.11417323,  0.12204724,
              0.12992126,  0.13779528,  0.14566929,  0.15354331,  0.16141732,
              0.16929134,  0.17716535,  0.18503937,  0.19291339,  0.2007874 ,
              0.20866142,  0.21653543,  0.22440945,  0.23228346,  0.24015748,
              0.2480315 ,  0.25590551,  0.26377953,  0.27165354,  0.27952756,
              0.28740157,  0.29527559,  0.30314961,  0.31102362,  0.31889764,
              0.32677165,  0.33464567,  0.34251969,  0.3503937 ,  0.35826772,
              0.36614173,  0.37401575,  0.38188976,  0.38976378,  0.3976378 ,
              0.40551181,  0.41338583,  0.42125984,  0.42913386,  0.43700787,
              0.44488189,  0.45275591,  0.46062992,  0.46850394,  0.47637795,
              0.48425197,  0.49212598,  0.5       ]),
      array([nan,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
              0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
              0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
              0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
              0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
              0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
              0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
              0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
              0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,
              0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0., nan]))
```

or evaluate the wavefront,

```
[5]: p.pv, p.rms # in units of opd_unit
```

```
[5]: (0.0, 0.0)
```

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

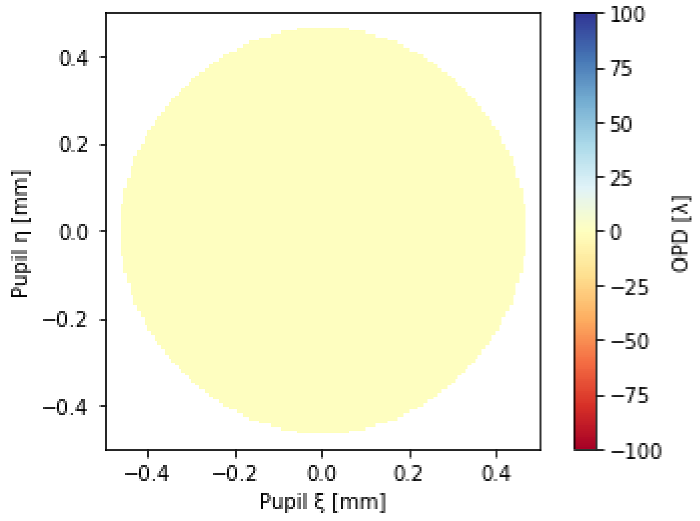
```
[6]: p.strehl  #  [0,1]
```

```
[6]: 1.0
```

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

```
[7]: p.plot2d(cmap='RdYlBu', clim=(-100,100), interp_method='sinc')
```

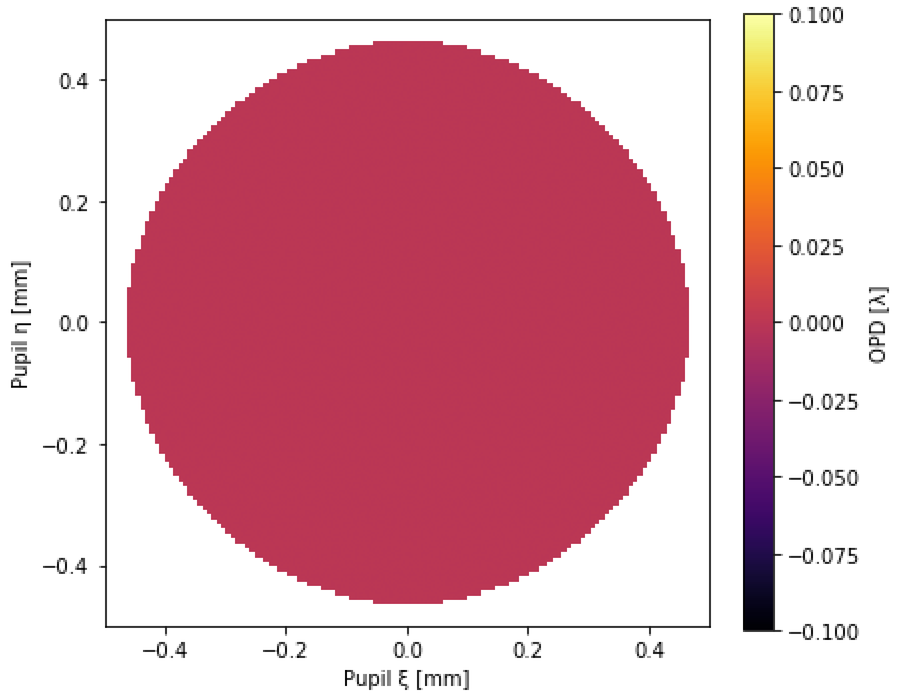
```
[7]: (<Figure size 432x288 with 2 Axes>,
      <matplotlib.axes._subplots.AxesSubplot at 0x7f5b548b5630>)
```



`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

```
[8]: from matplotlib import pyplot as plt
     fig, ax = plt.subplots(figsize=(6,6))
     p.plot2d(fig=fig, ax=ax)
```

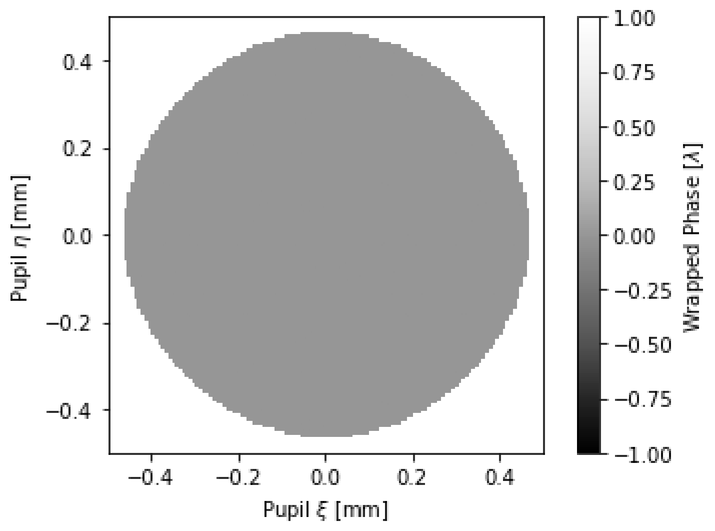
```
[8]: (<Figure size 432x432 with 2 Axes>,
      <matplotlib.axes._subplots.AxesSubplot at 0x7f5b5420c860>)
```



A synthetic interferogram may be generated,

```
[9]: # this one is empty because there is no phase error.
     p.interferogram(passes=2, visibility=0.5)
```

```
[9]: (<Figure size 432x288 with 2 Axes>,
      <matplotlib.axes._subplots.AxesSubplot at 0x7f5b540b4588>)
```



Pupils also support addition and subtraction with other pupil objects,

```
[10]: p2 = p + p - p
```

3.1.7 Zernikes

Prysm supports two flavors of Zernike polynomials; the Fringe set up to the 49th term, and the Noll set up to the 36th 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__`,

```
[1]: %matplotlib inline
     from prysm import FringeZernike, NollZernike
     p = FringeZernike(samples=123, dia=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 variance via the `norm` keyword argument. Both notations also have nice print statements,

```
[ ]: p2 = FringeZernike(Z4=1, Z9=1, Z48=1, base=0, norm=True)
     p2
```

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 constructor, the value would converge to the analytical one.

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

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

Zernike 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)
```

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

```
[ ]: fz4.magnitudes
```

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.

Zernike functions are cached by prysm's internal routines,

[ ]:
```
from prysm.zernike import zcache
# 8 is the index into prysm.zernike.zernikes, which loosely follows the Fringe␣
↪ordering
zcache(8, norm=True, samples=128)
zcache.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.zernike import ZCache
my_zcache = ZCache()
```

See *Pupils* for information about general pupil functions.

CHAPTER 4

Examples

4.1 Examples

4.1.1 Analysis of Interferometric Wavefront Data

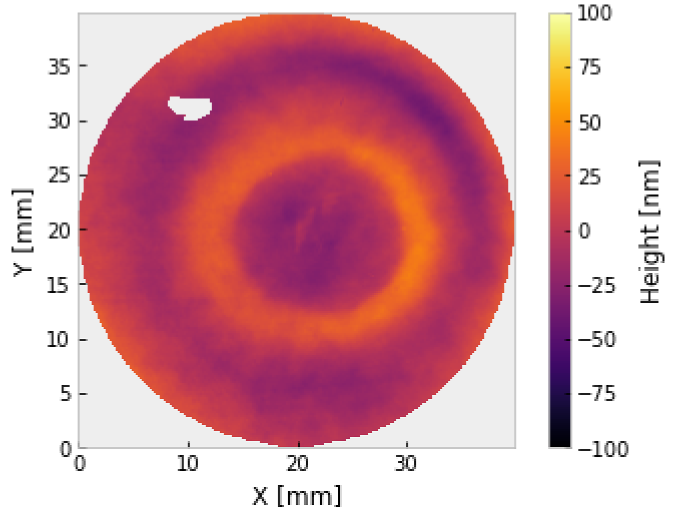
In this example, we will see how to use prysm to almost entirely supplant the software that comes with a commerical interferometer to analyze the wavefront of an optic. We begin by importing the relevant classes and setting some aesthetics for matplotlib.

```
[1]: from prysm import Interferogram, FringeZernike, sample_files, zernikefit

     from matplotlib import pyplot as plt
     plt.style.use('bmh')
```

We point prysm to the file, create a new interferogram, mask it to a circular region 100 mm across, subtract piston, tip/tilt and power, and evalute the PV and RMS wavefront error. We also plot the wavefront to make sure all has gone well

```
[2]: p = sample_files('dat')  # sample Zygo .dat file, will be downloaded on demand and␣
     ↪saved locally
     i = Interferogram.from_zygo_dat(p)
     i.crop().mask('circle', 40).crop()
     i.remove_piston_tiptilt_power()
     print(i.pv, i.rms)
     i.plot2d(clim=100, interp_method='bilinear')  # +/- 100 nm
     plt.grid(False)
```

```
83.33634959318245 15.245987053549815
```



The interferogram is cropped twice – once to enclose the valid data, then again to apply a mask centered on that region. For relatively conventional interferometry, you may want to stop here. If you want to use a different unit, that is easy enough,

```
[3]: i.change_phase_unit('waves')
     1/i.pv, 1/i.rms  # print reciprocal -- "one over xxx waves"
```

```
[3]: (18.962651752261532, 103.65207382701372)
```

There is no need to crop again since the outer bound has not changed. Perhaps you wish to evaluated the RMS within the 1 - 10 mm spatial periods,

```
[4]: i.change_phase_unit('nm')
     i.fill()
     i.bandlimited_rms(1,10)
```

```
[4]: 9.51382202472974
```

This value is derived from the PSD, so you must call fill first. Do not worry about the corners of the array containing data - it will be windowed out. If you do this on a part which has a central obscuration, the result will be incorrect. Otherwise, the value tends to agree to within +/- 5% of Zygo's Mx ® software, though the authors of prysm do not believe they are calculated at all the same way.

If you wish to decompose the wavefront into Zernike polynomials, that is easy enough.

```
[5]: # do this on data which has not been filled to avoid errors introduced by the fill␣
     ↪value.
     coefficients = zernikefit(i.phase, terms=36, norm=True, map_='fringe')
     fz = FringeZernike(coefficients, dia=i.diameter, opd_unit=i.phase_unit, norm=True)
     fz
```

```
[5]: rms normalized Fringe Zernike description with:
         -1.195 Z1 - Piston
         -0.271 Z2 - Tilt Y
         +0.484 Z3 - Tilt X
         -2.172 Z4 - Defocus
         -1.351 Z5 - Primary Astigmatism 0°
         +0.324 Z6 - Primary Astigmatism 45°
         -0.217 Z7 - Primary Coma Y
```

(continues on next page)

(continued from previous page)

```
    -1.655 Z8 - Primary Coma X
    -0.241 Z9 - Primary Spherical
    +3.036 Z10 - Primary Trefoil Y
    -1.244 Z11 - Primary Trefoil X
    +1.381 Z12 - Secondary Astigmatism 0°
    -0.049 Z13 - Secondary Astigmatism 45°
    +2.384 Z14 - Secondary Coma Y
    -1.735 Z15 - Secondary Coma X
    +6.297 Z16 - Secondary Spherical
    +1.349 Z17 - Primary Tetrafoil Y
    -0.166 Z18 - Primary Tetrafoil X
    -0.842 Z19 - Secondary Trefoil Y
    +0.229 Z20 - Secondary Trefoil X
    +0.423 Z21 - Tertiary Astigmatism 0°
    -0.040 Z22 - Tertiary Astigmatism 45°
    -1.042 Z23 - Tertiary Coma Y
    +1.362 Z24 - Tertiary Coma X
    -2.779 Z25 - Tertiary Spherical
    +0.171 Z26 - Primary Pentafoil Y
    +0.020 Z27 - Primary Pentafoil X
    -0.235 Z28 - Secondary Tetrafoil Y
    +0.046 Z29 - Secondary Tetrafoil X
    -0.005 Z30 - Tertiary Trefoil Y
    -0.005 Z31 - Tertiary Trefoil X
    -0.489 Z32 - Quaternary Astigmatism 0°
    +0.025 Z33 - Quaternary Astigmatism 45°
    +0.106 Z34 - Quaternary Coma Y
    -0.192 Z35 - Quaternary Coma X
    +0.307 Z36 - Quaternary Spherical
    45.208 PV, 9.334 RMS [nm]
```

This print might be a bit daunting, one may prefer to see the top few terms by magnitude,

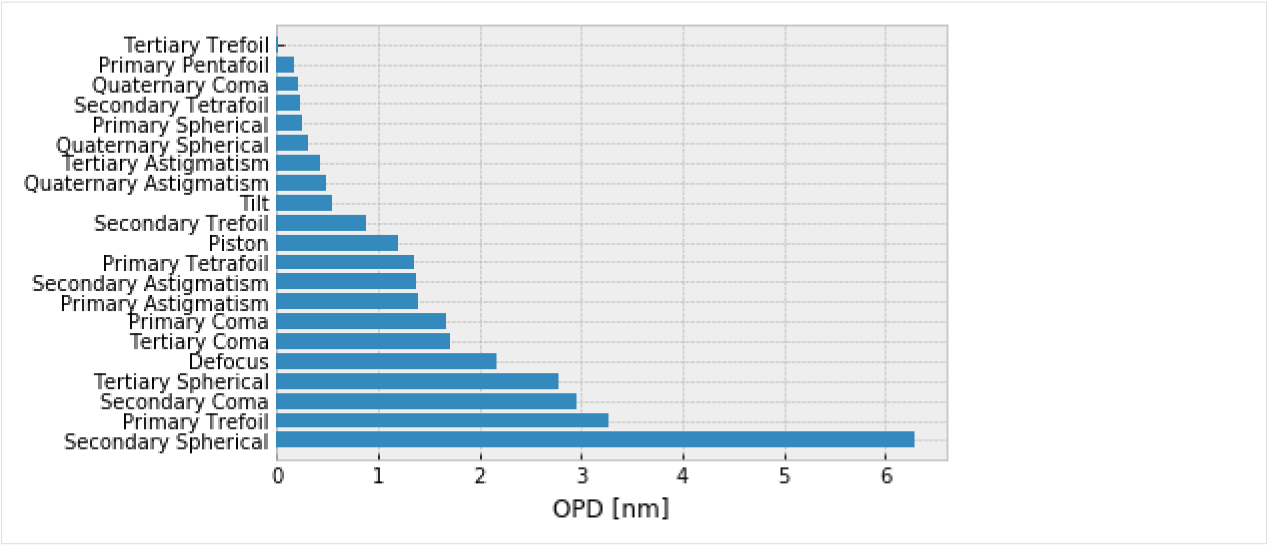
```
[6]: fz.top_n(5)
```

```
[6]: [(6.296543, 16, 'Secondary Spherical'),
      (3.035946, 10, 'Primary Trefoil Y'),
      (-2.779492, 25, 'Tertiary Spherical'),
      (2.38442, 14, 'Secondary Coma Y'),
      (-2.171752, 4, 'Defocus')]
```

or a barplot of all terms,

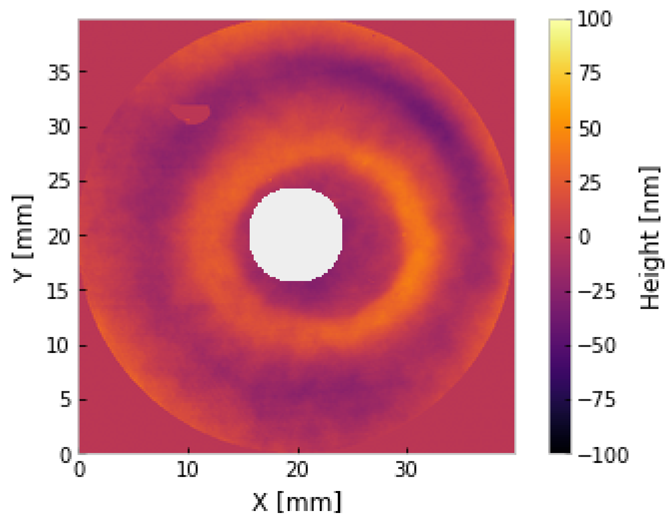
```
[7]: fz.barplot_magnitudes(orientation='v', sort=True)
```

```
[7]: (<Figure size 432x288 with 1 Axes>,
      <matplotlib.axes._subplots.AxesSubplot at 0x7f3caff27cf8>)
```



The sample data has a ciruclar clear aperture, but if it had a central obscuration (such as transmitted wavefront data for a telescope) that would be easy to mask too. PSD, and by extension bandlimited RMS values for data with annular support will be nonsensical.

```
[8]: i.mask('invertedcircle', 7)
     i.plot2d(clim=100)   # +/- 100 nm
     plt.grid(False)
```



```
[ ]:
```

4.1.2 Diffraction Limited PSF and MTF

In this example, we will show how to calculate the diffraction limited PSF and MTF for a circular aperture. We will also compare the numerically derived results from `prysm` with the analytical forms.

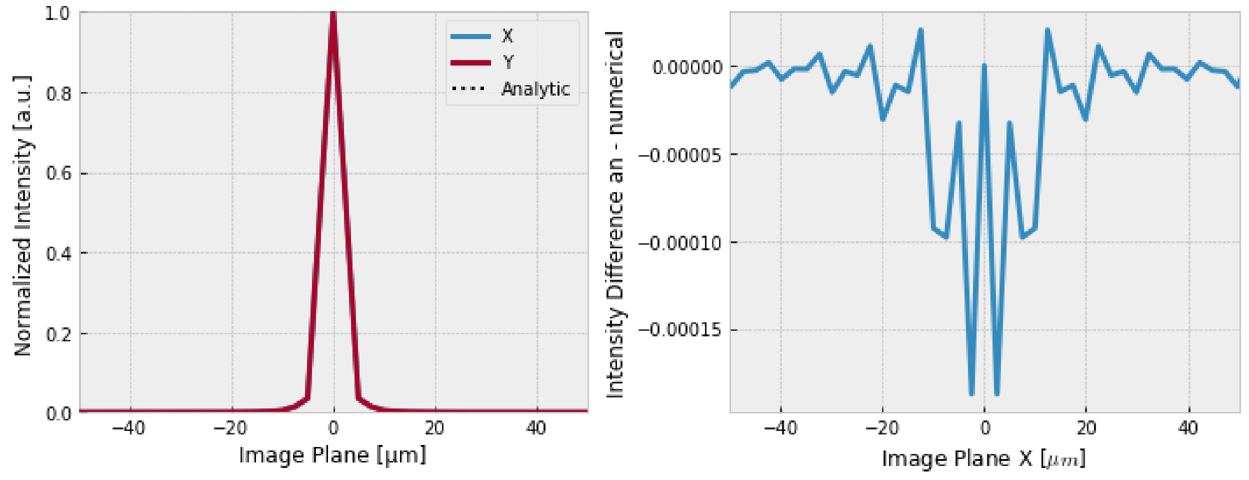
```
[1]: import numpy as np
```

(continues on next page)

(continued from previous page)

```
from prysm import Pupil, PSF, MTF
from prysm.psf import airydisk
from prysm.otf import diffraction_limited_mtf

from matplotlib import pyplot as plt
%matplotlib inline
plt.style.use('bmh')
```

[2]:
```
# system parameters
epd = 1
efl = 10
fno = efl / epd
wavelength = 0.5

# mask_target is just a performance optimization
# more samples makes for more accuracy.
# The below would get you amost the same answer:
# p = Pupil(wavelength=wavelength, dia=epd)

p = Pupil(wavelength=wavelength, dia=epd, mask_target='fcn', samples=256)
psf = PSF.from_pupil(p, efl)
u, sx = psf.slice_x

# calculate the analytical version, and the difference between the two
analytical_psf = airydisk(u, fno, wavelength)
psferr = (analytical_psf - sx)

fig, (ax1, ax2) = plt.subplots(ncols=2, figsize=(10,4))
psf.plot_slice_xy(fig=fig, ax=ax1, axlim=50)
ax1.plot(u, analytical_psf, ls=':', c='k', label='Analytic')
ax1.legend()
ax2.plot(u, psferr, lw=3)
ax2.set(xlim=(-50,50), xlabel=r'Image Plane X [$\mu m$]', ylabel='Intensity␣
↪Difference an - numerical')
fig.tight_layout()
```



One can see that the differences manifest below the fourth decimal place. It might be interesting to see the RMS error,

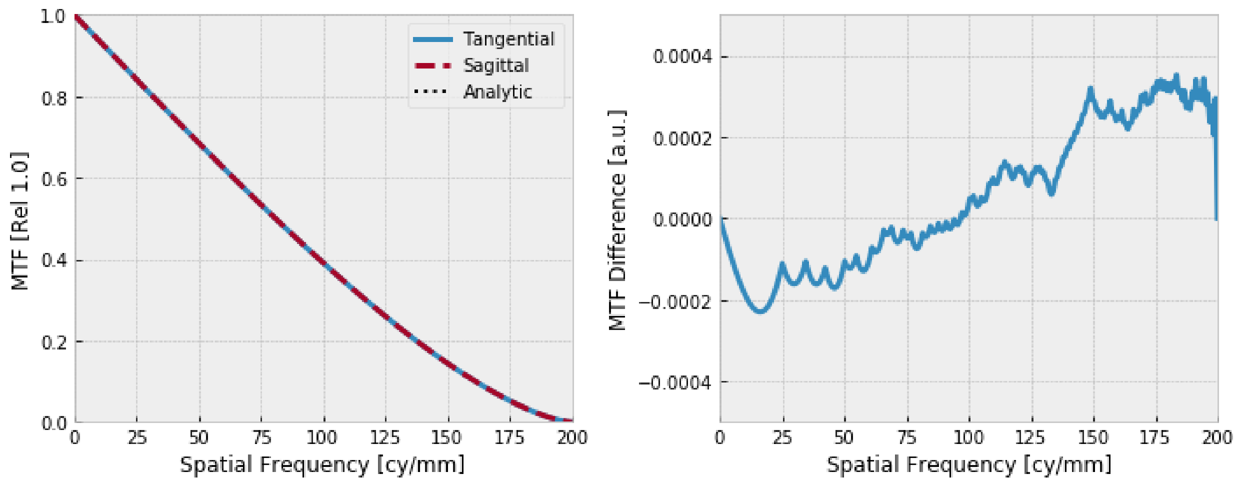
```
[3]: from prysm.util import rms

     rms(psferr)
```

```
[3]: 1.491584437862731e-05
```

This error is over a 1D slice. If calculated over the whole image plane it would be much smaller.

Next, we consider the MTF:

```
[4]: # calculate the MTF and its error
     mtf = MTF.from_psf(psf)
     nu, m = mtf.tan
     m_analytic = diffraction_limited_mtf(fno, wavelength, frequencies=nu)
     mtferr = (m_analytic - m)

     fig, (ax1, ax2) = plt.subplots(ncols=2, figsize=(10,4))
     mtf.plot_tan_sag(fig=fig, ax=ax1)
     ax1.plot(nu, m_analytic, ls=':', c='k', label='Analytic')
     ax1.legend()
     ax2.plot(nu, mtferr, lw=3)
     ax2.set(xlim=(0,200), ylim=(-0.0005,0.0005), xlabel='Spatial Frequency [cy/mm]',␣
     ↪ylabel='MTF Difference [a.u.]')
     fig.tight_layout()
```



Once again the error is at the fourth decimal place. The RMS may be interesting again,

```
[5]: rms(mtferr)
```

```
[5]: 0.00018149825282448085
```

4.1.3 Image-Based Wavefront Sensing

In this example, we will show how ot implement image-based wavefront sensing based on prysm. We will use the library both to synthesize the truth data, and as the embedded modeling tool used as part of the optimization routine.

We begin by importing a few classes and functions from prysm and other modules and writing some functions to generate data from zernike coefficients,

[1]:
```
from functools import partial

import numpy as np

from scipy.optimize import minimize

from prysm import NollZernike, PSF

from matplotlib import pyplot as plt
%matplotlib inline
plt.style.use('bmh')

def bake_in_defocus(zernikes, defocus_values):
    return [{**zernikes, **dict(Z4=defocus)} for defocus in defocus_values]

def zerns_idxs_to_dict(zernike_coefs, indices):
    return {k:v for k, v in zip(indices, zernike_coefs)}

# a few extra variables are passed in and aren't used,
# but we're aiming for brevity in this example, not pristine code
# so we just collect them with kwargs and don't use them
def zernikes_to_pupil(zerns, epd, wvl, samples, efl=None, Q=None, target='fcn'):
    return NollZernike(**zerns,  # coefficients
                       dia=epd, wavelength=wvl,  # physical parameters
                       norm=True, opd_unit='waves',  # units and normalization
                       mask='circle', mask_target=target,  # geometry
                       samples=samples)  # sampling

def zernikes_to_psfs(sets_of_zernikes, efl, epd, Q, wvl, samples):
    psfs = []
    for zerns in sets_of_zernikes:
        pupil = zernikes_to_pupil(zerns, epd, wvl, samples)
        psf = PSF.from_pupil(pupil, Q=Q, efl=efl)
        psfs.append(psf)
    return psfs
```

`zernikes_to_psfs` is quite terse, but has a lot going on. `zerns` is a dictionary that contains key-value pairs of (Noll) Zernike indexes and their coefficients. Unpacking it allows us use arbitrary combinations of Zernike terms within the algorithm. In `zernikes_to_pupil`, `mask_target` is used to avoid masking the phase during optimization, improving performance. We make explicit that the pupil is circular with `mask`. It will be used both inside the optimizer and to synthesize data. We will treat EFL, EPD, Q, and λ as known.

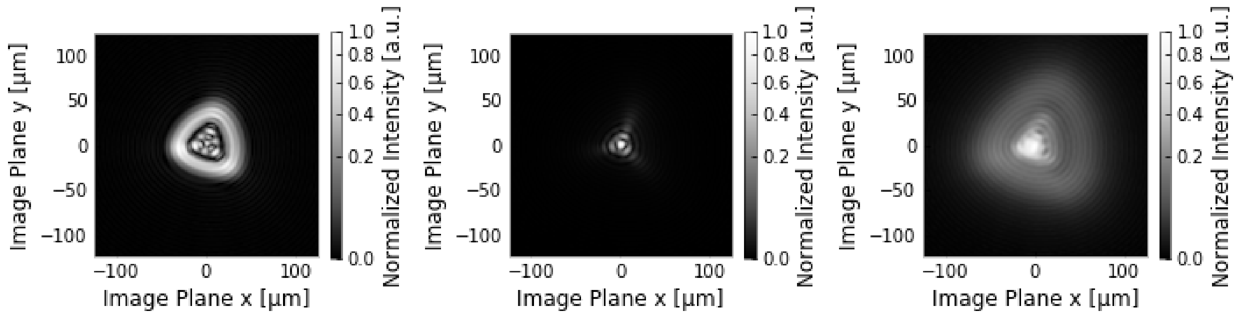
Next, we synthesize the truth data.

[2]:
```
# wavefront data
defocus_values_waves_rms = [-0.65, 0, 0.76]  # noninteger values with nothing special␣
↪about them
reference_zernikes = dict(Z5=0.012, Z6=0.023, Z7=0.034, Z8=0.045, Z9=0.056, Z10=0.067,
↪ Z11=0.12)
wavefront_coefs = bake_in_defocus(reference_zernikes, defocus_values_waves_rms)

# system parameters
efl = 1500
epd = 150  # F/10, e.g. an unobscured telescope with 15 cm aperture
wvl = 0.55 # monochromatic visible system
Q = 2.13   # minorly oversampled
samples = 128 # this is a reasonable number for small OPD and uncomplicated aperture␣
↪geometry
```

(continues on next page)

(continued from previous page)

```
ztp_kwargs = dict(efl=efl, epd=epd, Q=Q, wvl=wvl, samples=samples)
truth_psfs = zernikes_to_psfs(wavefront_coefs, **ztp_kwargs)

def rowplot_psfs(psfs):
    fig, axs = plt.subplots(ncols=3, figsize=(10,4))
    for psf, ax in zip(psfs, axs):
        psf.plot2d(fig=fig, ax=ax, axlim=125, power=2)
        ax.grid(False)

    fig.tight_layout()
    return fig, axs

rowplot_psfs(truth_psfs);
```



The goal of image-based wavefront sensing is to take these PSFs and use them to estimate the wavefront that generated them (above, `reference_zernikes`) without prior knowledge. We do this by constructing a nonlinear optimization problem and asking the computer to determine the wavefront coefficients for us. To that end, we formulate a cost function which calculates the mean square error between the data and the truth,

```
[3]: def optfcn(zernike_coefs, indices, ztp_kwargs, defocuses, truth_psfs):
         base_wavefront_coefs = zerns_idxs_to_dict(zernike_coefs, indices)
         wavefront_coefs = bake_in_defocus(base_wavefront_coefs, defocuses)
         psfs = zernikes_to_psfs(wavefront_coefs, **ztp_kwargs)

         # t = truth, m = model.  sum(^2) = mean square error
         diffs = [((t.data - m.data)**2).sum() for t, m in zip(truth_psfs, psfs)]

         # normalize by N,
         # psf.size = number of pixels
         diff = sum(diffs) / (len(psfs) * psfs[0].size)
         return diff

     coefs = [f'Z{n}' for n in [5, 6, 7, 8, 9, 10, 11]]
     optfcn_used = partial(optfcn,
                           indices=coefs,
                           ztp_kwargs=ztp_kwargs,
                           defocuses=defocus_values_waves_rms,
                           truth_psfs=truth_psfs)
```

This function computes the mean square error between the data in our model PSFs and the truth. Let's check that it returns zero for the truth:

```
[4]: optfcn_used(reference_zernikes.values())
```

```
[4]: 0.0
```

This (probably) means we didn't make a mistake. In your own program, this should be verified more rigorously. To get a sense for execution time, how quickly can we evaluate it?

```
[5]: %timeit optfcn_used(reference_zernikes.values())
```

```
72.7 ms ± 5.09 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

On the computer this document was written on, the time is 5.5ms per run. So we should expect much less than a second per iteration of the optimizer; this problem is not so large that running it requires a supercomputer.

All that is left to do is ask the optimizer kindly for the true wavefront, given a guess. A general guess might be all zeros – a pupil with no wavefront error. This does not assume any prior knowledge. The L-BFGS-B minimizer tends to perform the best for this problem, from experience.

```
[6]: opt_result = minimize(optfcn_used, tuple([0] * len(coefs)), method='L-BFGS-B')
opt_result
```

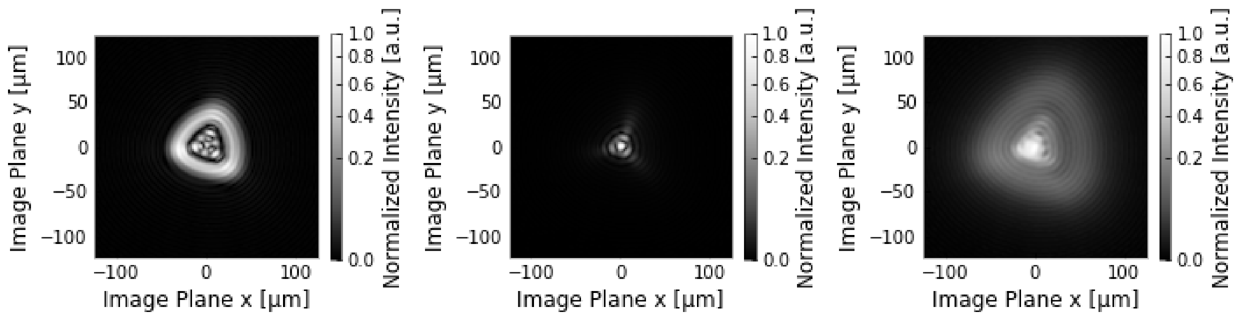
```
[6]:       fun: 7.018049323512015e-10
 hess_inv: <7x7 LbfgsInvHessProduct with dtype=float64>
      jac: array([1.46185426e-06, 2.59488516e-06, 3.67814544e-06, 3.74879589e-06,
        4.55081702e-07, 3.22723710e-06, 1.36784945e-06])
  message: b'CONVERGENCE: NORM_OF_PROJECTED_GRADIENT_<=_PGTOL'
     nfev: 208
      nit: 18
   status: 0
  success: True
        x: array([0.01205372, 0.02314215, 0.03408237, 0.04508817, 0.05602504,
        0.06708379, 0.12002976])
```

Casting the `x0` variable to a tuple makes it immutable, which will help when you try to debug one of these problems, but doesn't matter here.

How did the optimizer do? Well, let's compare the PSFs.

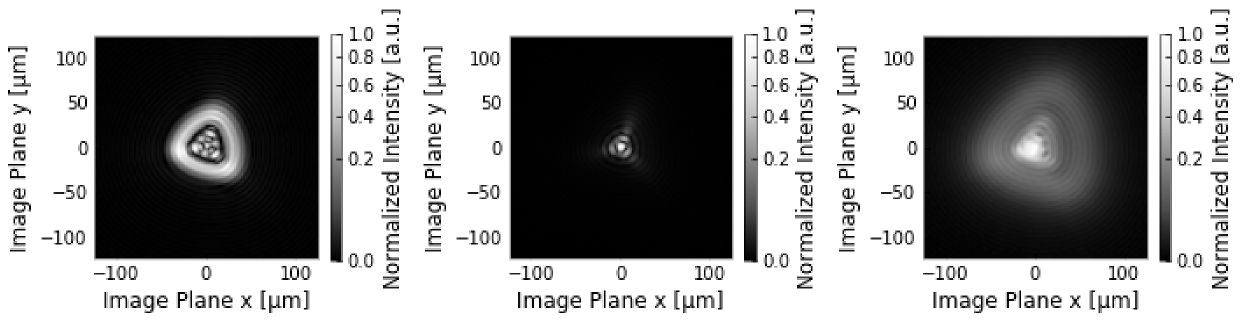
```
[7]: # this is copy pasted from optfcn, but we aren't trying to make the cleanest code␣
↪right now.
base_wavefront_coefs = zerns_idxs_to_dict(opt_result.x, coefs)
wavefront_coefs = bake_in_defocus(base_wavefront_coefs, defocus_values_waves_rms)
retrieved_psfs = zernikes_to_psfs(wavefront_coefs, **ztp_kwargs)

print('true PSFs')
rowplot_psfs(truth_psfs);
```

```
true PSFs
```



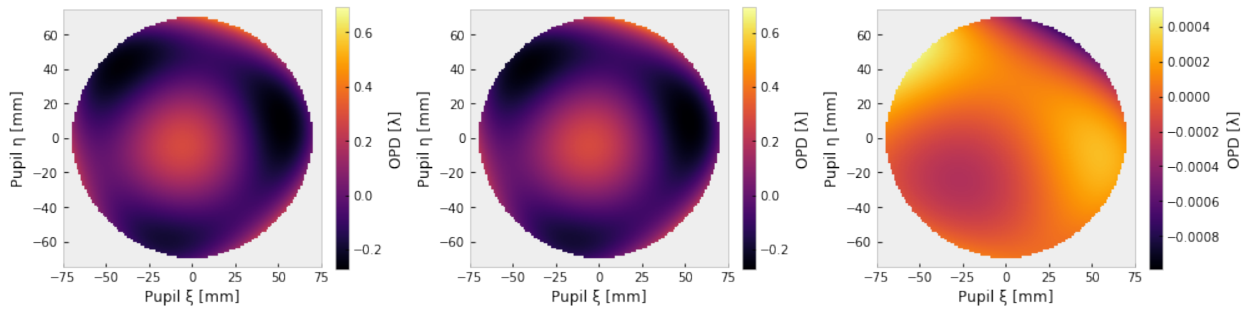
[8]:
```
print('estimated PSFs')
rowplot_psfs(retrieved_psfs);
```

```
estimated PSFs
```



Looks like a good match. How about the wavefronts? We have to change the mask target here to the phase visualizes the way we expect.

[9]:
```
true_wavefront = zernikes_to_pupil(reference_zernikes, **ztp_kwargs, target='both')
retrieved_wavefront = zernikes_to_pupil(wavefront_coefs[1], **ztp_kwargs, target='both
↪')
elementwise_difference = true_wavefront - retrieved_wavefront

fig, axs = plt.subplots(ncols=3, figsize=(14,8))
true_wavefront.plot2d(fig=fig, ax=axs[0])
retrieved_wavefront.plot2d(fig=fig, ax=axs[1])
elementwise_difference.plot2d(fig=fig, ax=axs[2])


for ax in axs:
    ax.grid(False)

fig.tight_layout()
```



The optimization function ideally lets us predict the RMS error of the estimate. Let's compare,

[10]:
```
np.sqrt(opt_result.fun), elementwise_difference.rms
```

[10]:
```
(2.6491601166241378e-05, 0.00021466060172462098)
```

They differ by about a factor of ten, which is unfortunate. The RMS error is 2 thousandths of a wave; a pretty good result. What's the peak error?

[11]:
```
elementwise_difference.pv
```

```
[11]: 0.0015103102405956137
```

about 1 1/2 hundredth of a wave, or ~7 nanometers. Not bad. Competitive with interferometers, anyway.

More advanced tasks are left to the reader, such as:

- handling of noise
- performance optimization
- improved flexibility w.r.t units, pupil shapes, etc
- estimation of Q or other system parameters
- inclusion of focal plane effects
- extension to N PSFs instead of 3
- use of more terms
- use of other types of phase diversity instead of focus
- formulation of a cost function better connected to the error in the wavefront estimate
- tracking of the optimizer to better understand the course of optimization
- use of other data, such as MTF

```
[ ]:
```

4.1.4 Aberration Transfer Functions

One may find reference on the internet to an "Aberration Transfer Function" introduced by Shannon used to model the MTF of an aberrated imaging system as:

$$
\begin{aligned}
DTF(\nu) &= \frac{2}{\pi}\left[\arccos\nu - \nu\sqrt{1-\nu^2}\right] \\
ATF(\nu) &= 1 - \left(\frac{W_{\text{rms}}}{0.18}\right)^2\left(1-4(\nu-0.5)^2\right) \\
MTF(\nu) &= DTF(\nu) \times ATF(\nu)
\end{aligned}
$$

where DTF is the diffraction-limited MTF, ATF is the "Aberration Transfer Function," MTF is the modulation transfer function, and W_{rms} is the RMS wavefront error.

In this example, we will show that this treatment should not be used if accuracy is desired from a model. The example should also highlight the terse nature of examples such as this when calculated using prysm.

We begin by importing some classes and functions from the library, and defining the ATF function as Shannon describes it:

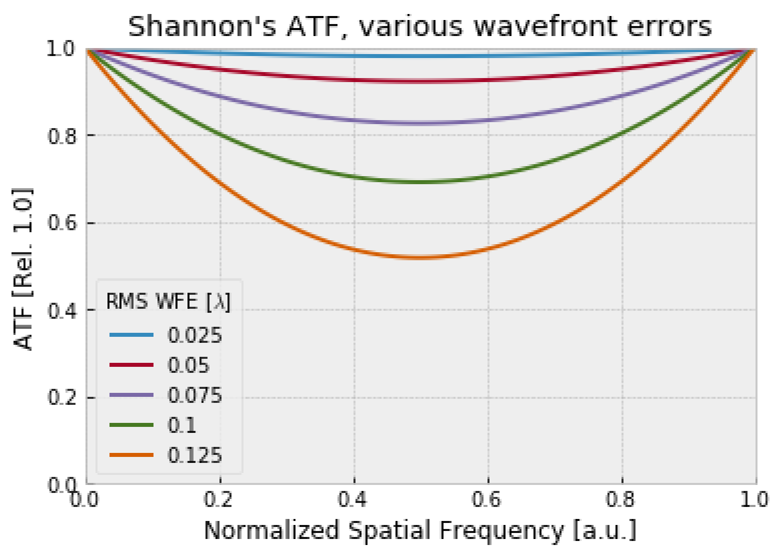
```
[1]: import numpy as np

from prysm.otf import diffraction_limited_mtf
from prysm import FringeZernike, PSF, MTF

from matplotlib import pyplot as plt

def shannon_atf(nu, Wrms):
```

(continues on next page)

(continued from previous page)

```
    return 1 - ((Wrms / 0.18) ** 2 * (1 - 4 * (nu - 0.5) ** 2 ))

%matplotlib inline
plt.style.use('bmh')
```

[2]:
```
Wrms_vals = [0.025, 0.05, 0.075, 0.1, 0.125]
nu = np.linspace(0, 1, 100)
atf_curves = []
for Wrms in Wrms_vals:
    atf_curves.append(shannon_atf(nu=nu, Wrms=Wrms))

fig, ax = plt.subplots()
for (curve, label) in zip(atf_curves, Wrms_vals):
    ax.plot(nu, curve, label=label)

ax.legend(title=r'RMS WFE [$\lambda$]')
ax.set(xlim=(0,1), xlabel='Normalized Spatial Frequency [a.u.]',
       ylim=(0,1), ylabel='ATF [Rel. 1.0]',
       title="Shannon's ATF, various wavefront errors");
```



Now we'll pick a few different Zernike modes and show the numerically derived version, generated by calculating the MTF numerically and dividing by the diffraction limited MTF for a circular aperture, given above as DTF. The accuracy of prysm's MTF calculations is sufficiently high that we can ignore that as a reason for the discrepancy. *The accuracy of prysm's MTF calculations is that we can ignore that as a reason for any discrepancy.*

[3]:
```
# only 20 lines of code, half of which is looping or plotting!
def render_atf_curves_zernike_mode(mode_index, Wrms_vals):
    kwarg = {}

    real_mtfs = []
    for Wrms in Wrms_vals:
        kwarg[f'Z{mode_index}'] = Wrms
        pupil = FringeZernike(**kwarg, norm=True, opd_unit='waves')  # waves is the␣
↪default, not really needed
        psf = PSF.from_pupil(pupil, efl=2)  # normalized frequency makes this choice␣
↪arbitrary
```

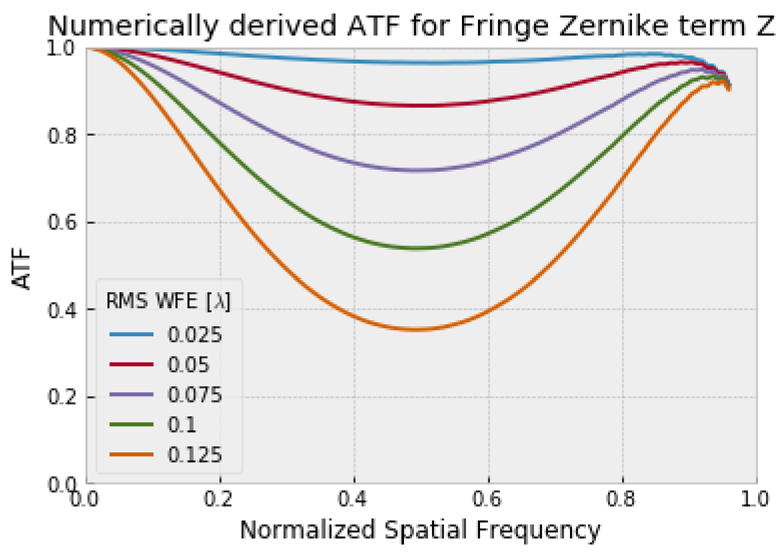
(continues on next page)

(continued from previous page)

```
        mtf = MTF.from_psf(psf)

        u, mtf_ = mtf.tan
        real_mtfs.append(mtf_)

    cutoff = 1 / (psf.wavelength * psf.fno) * 1e3 # 1e3 is cy/um => cy/mm
    normalized_frequencies = u / cutoff
    diffraction_limit = diffraction_limited_mtf(psf.fno, psf.wavelength,␣
↪frequencies=u)

    # don't plot quite all of the curve, division by almost zero is a problem at the␣
↪end
    fig, ax = plt.subplots()
    for (curve, label) in zip(real_mtfs, Wrms_vals):
        atf = curve / diffraction_limit
        ax.plot(normalized_frequencies[:-5], atf[:-5], label=label)

    ax.legend(title=f'RMS WFE [$\lambda$]')
    ax.set(xlim=(0,1), xlabel='Normalized Spatial Frequency',
           ylim=(0,1), ylabel='ATF',
           title=f'Numerically derived ATF for Fringe Zernike term Z{mode_index}')

    return fig, ax
```

```
[4]: # Z4 = defocus, the lowest-order Zernike error to affect imaging (and MTF)
     render_atf_curves_zernike_mode(4, Wrms_vals)
```

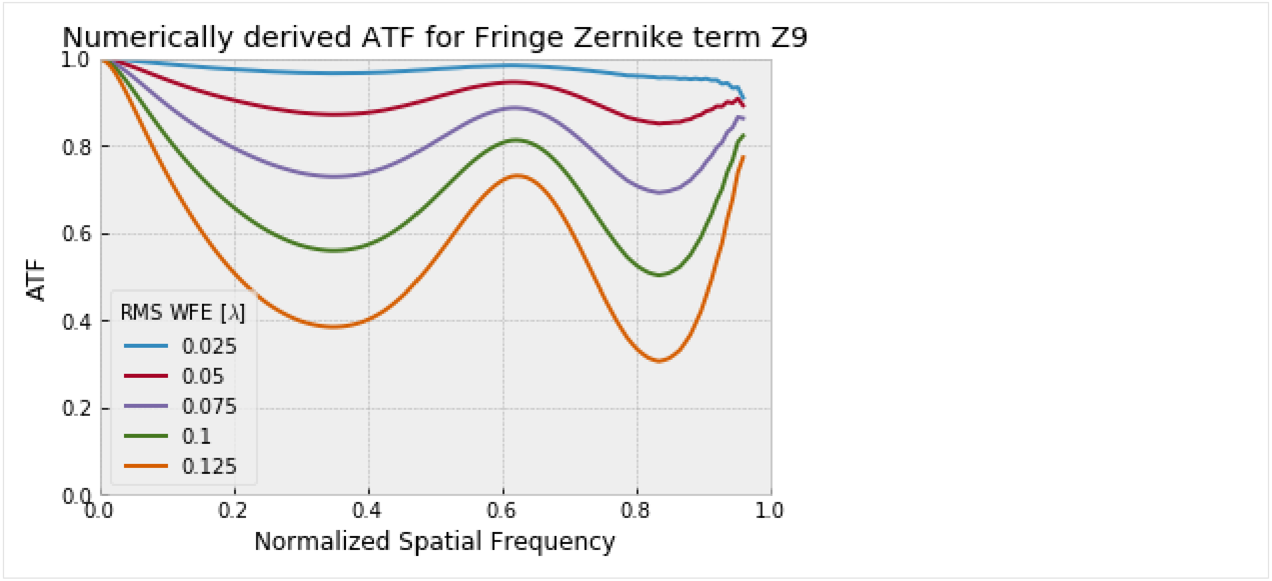
```
[4]: (<Figure size 432x288 with 1 Axes>,
      <matplotlib.axes._subplots.AxesSubplot at 0x7f9d74aa04a8>)
```



The curve looks broadly similar, but the belly reaches down quite a bit further. What about higher order terms?

```
[5]: # Z9 = "zernike primary spherical" -- low-order spherical aberration
     render_atf_curves_zernike_mode(9, Wrms_vals)
```

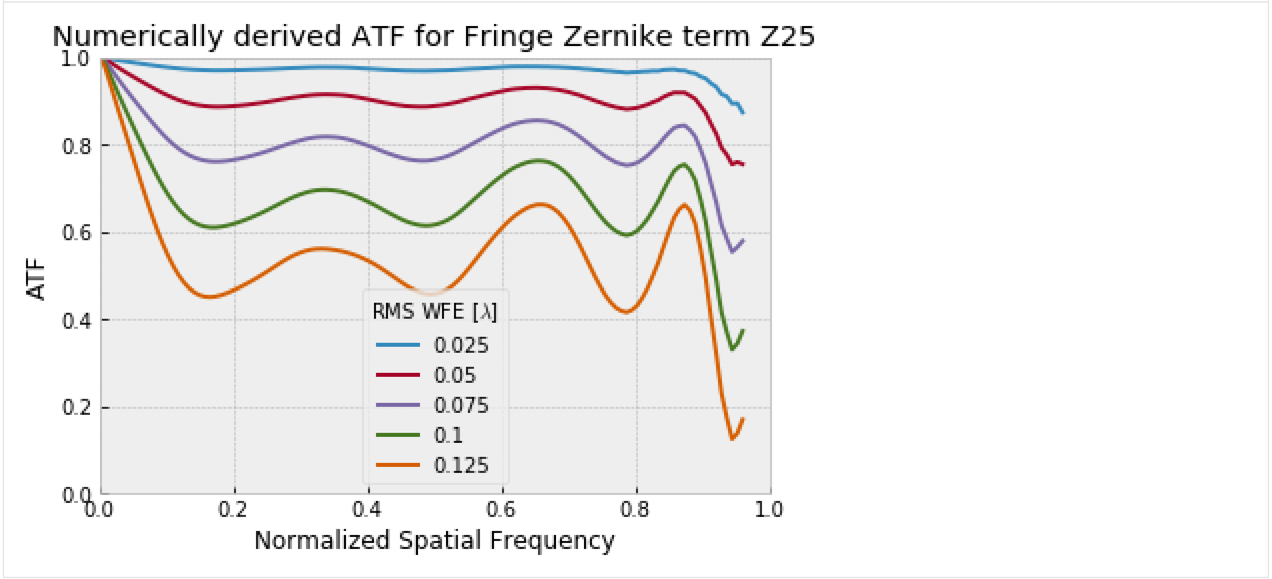
```
[5]: (<Figure size 432x288 with 1 Axes>,
      <matplotlib.axes._subplots.AxesSubplot at 0x7f9d741ed208>)
```



We can see that for *low order* spherical aberration, the curves look very different. What if we had a higher order variant?

```
[6]: # Z 25 = "zernike tertiary spherical" -- 8th order spherical aberration, in Hopkins'␣
     ↪wave aberration expansion
     render_atf_curves_zernike_mode(25, Wrms_vals)
```

```
[6]: (<Figure size 432x288 with 1 Axes>,
      <matplotlib.axes._subplots.AxesSubplot at 0x7f9d74167f98>)
```



Even worse. These are lots of squiggly lines, what if we directly compare a real ATF for a reasonable wavefront vs Shannon's ATF equation?

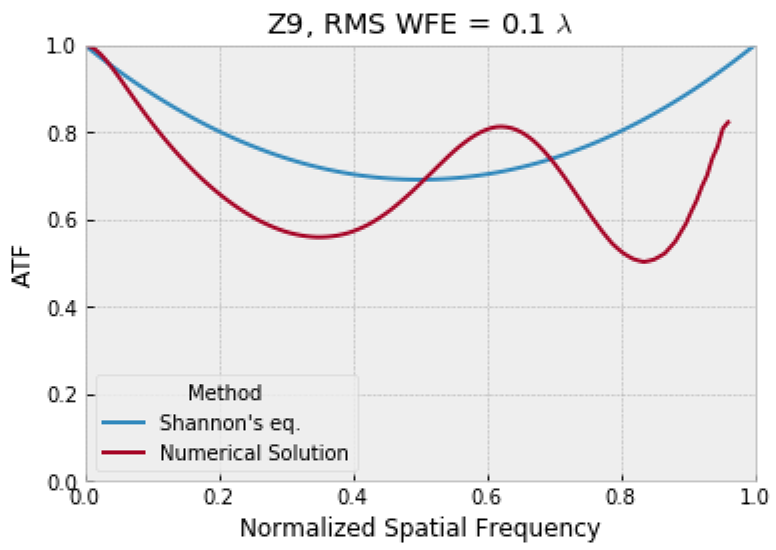
```
[7]: # most of this code is just copy pasted from above
     pupil = FringeZernike(Z9=0.1, norm=True, opd_unit='waves')  # waves is the default,␣
     ↪not really needed
     psf = PSF.from_pupil(pupil, efl=2)  # normalized frequency makes this choice arbitrary
```

(continues on next page)

(continued from previous page)

```
mtf = MTF.from_psf(psf)

u, mtf_ = mtf.tan

diffraction_limit = diffraction_limited_mtf(psf.fno, psf.wavelength, frequencies=u)

real_atf = mtf_ / diffraction_limit
unormalized = u / (1 / (psf.wavelength * psf.fno) * 1e3)

fig, ax = plt.subplots()

ax.plot(nu, shannon_atf(nu, 0.1), label="Shannon's eq.")
ax.plot(unormalized[:-5], real_atf[:-5], label='Numerical Solution')

ax.legend(title='Method')
ax.set(xlim=(0,1), xlabel='Normalized Spatial Frequency',
       ylim=(0,1), ylabel='ATF',
       title=r'Z9, RMS WFE = 0.1 $\lambda$')
```

[7]:
```
[(0, 1),
 Text(0, 0.5, 'ATF'),
 (0, 1),
 Text(0.5, 0, 'Normalized Spatial Frequency'),
 Text(0.5, 1.0, 'Z9, RMS WFE = 0.1 $\\lambda$')]
```



Not a good match. What if we look at the peak error in Shannon's equation as a function of Zernike index and RMS WFE corresponding to the Marechal critera?

[8]:
```
def render_atf_peakerror_vs_zernike(max_zernike=36, rms_wfe= 1 / 14): # 1 / 14 is the␣
↪Marechal criteria
    # a lot of this code is similar to the earlier function
    peak_errors = []

    # calculate one pilot case to get the metadata for the diffraction limited MTF. ␣
↪This is a performance optimization
    pupil = FringeZernike()
    psf = PSF.from_pupil(pupil, efl=2)
```

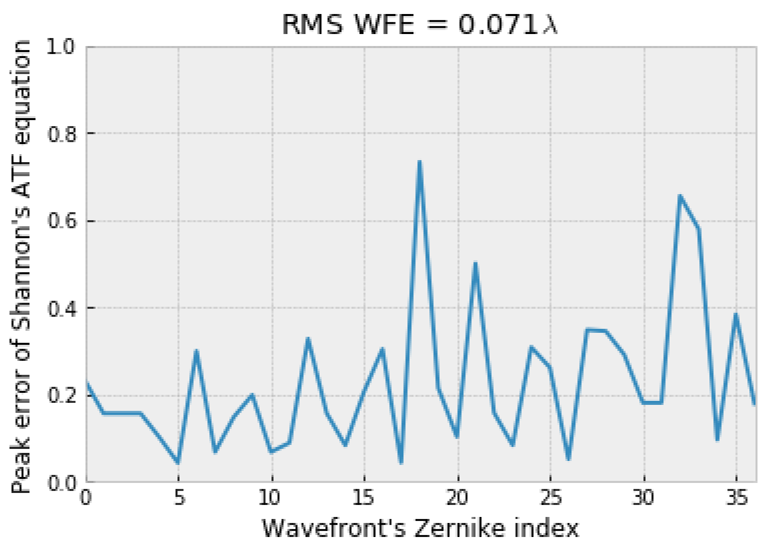
(continues on next page)

(continued from previous page)

```
    mtf = MTF.from_psf(psf)
    u, t = mtf.tan

    diffraction_limit = diffraction_limited_mtf(psf.fno, psf.wavelength,␣
↪frequencies=u)
    cutoff = 1 / (psf.wavelength * psf.fno) * 1e3
    normalized_frequencies = u / cutoff

    shannon = shannon_atf(normalized_frequencies, rms_wfe)

    idxs = list(range(max_zernike+1))
    for i in idxs:
        kwarg = {}
        kwarg[f'Z{i}'] = rms_wfe
        pupil = FringeZernike(**kwarg, norm=True, opd_unit='waves')  # waves is the␣
↪default, not really needed
        psf = PSF.from_pupil(pupil, efl=2)  # normalized frequency makes this choice␣
↪arbitrary
        mtf = MTF.from_psf(psf)

        cutoff = 1 / (psf.wavelength * psf.fno) * 1e3 # 1e3 is cy/um => cy/mm
        u, mtf_ = mtf.tan

        atf = mtf_ / diffraction_limit
        difference = abs(shannon[:-10] - atf[:-10])  # erode a little more of the end␣
↪here for high order cases
        peak_errors.append(difference.max())

    fig, ax = plt.subplots()
    ax.plot(idxs, peak_errors)
    ax.set(xlim=(0,max_zernike), xlabel="Wavefront's Zernike index",
           ylim=(0,1), ylabel="Peak error of Shannon's ATF equation",
           title=f'RMS WFE = {rms_wfe:.3f}' + r'$\lambda$')

    return fig, ax
```

```
[9]: render_atf_peakerror_vs_zernike(36, rms_wfe=1 / 14)
```

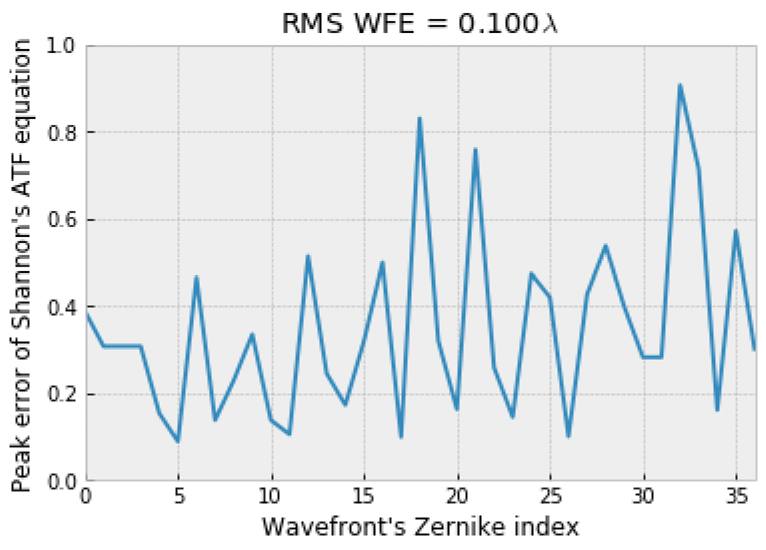
```
[9]: (<Figure size 432x288 with 1 Axes>,
      <matplotlib.axes._subplots.AxesSubplot at 0x7f9d74081240>)
```



For a wavefront at the Marechal criteria, the error can be as high as 0.7. Since MTF must be within the range [0, 1], this means the error is at least 70%. What if we had a tenth wave RMS?

```
[10]: render_atf_peakerror_vs_zernike(36, rms_wfe=1 / 10)
```

```
[10]: (<Figure size 432x288 with 1 Axes>,
       <matplotlib.axes._subplots.AxesSubplot at 0x7f9d6fbc0400>)
```



Now the absolute error can be as high as 0.9, again at least 90% due to the normalization of MTF.

Since these errors are so large, we can conclude that Shannon's ATF function should not be used if accuracy is desired.

4.1.5 "Onion Ring Bokeh"

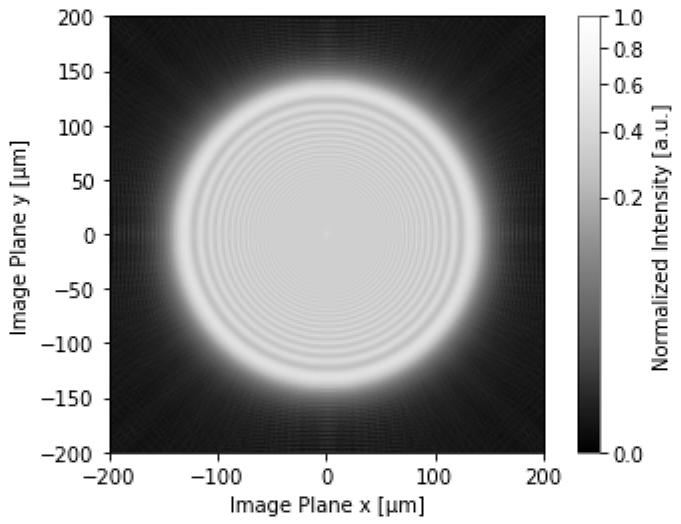
In the photography community, defects in the out-of-focus highlights of lenses are widely discussed. These result from mid-particular spatial frequency errors on the optical surfaces. Here, we will show an example of using prysm in a relatively extended fashion to model this. We begin, as usual, by importing some classes and other libraries.

```
[1]: import numpy as np

     from prysm import NollZernike, PSF

     from matplotlib import pyplot as plt
     %matplotlib inline
```

We begin by using the `NollZernike` class to make a pupil with 25 waves fo defocus to give us a relatively uniform disk for the diffraction limited out of focus image.

```
[2]: pupil = NollZernike(Z4=25/1.5, samples=384, dia=25)  # 100 waves PV of defocus, 25mm␣
     ↪for F/2
     pupil._gengrid()  # generate the (normalized coordinate) grid, stored in pupil.rho,␣
     ↪pupil.phi

     ps = PSF.from_pupil(pupil, efl=50, Q=1.25)
     ps.plot2d(axlim=200, power=3)
```

```
[2]: (<Figure size 432x288 with 2 Axes>,
      <matplotlib.axes._subplots.AxesSubplot at 0x7fab16813240>)
```



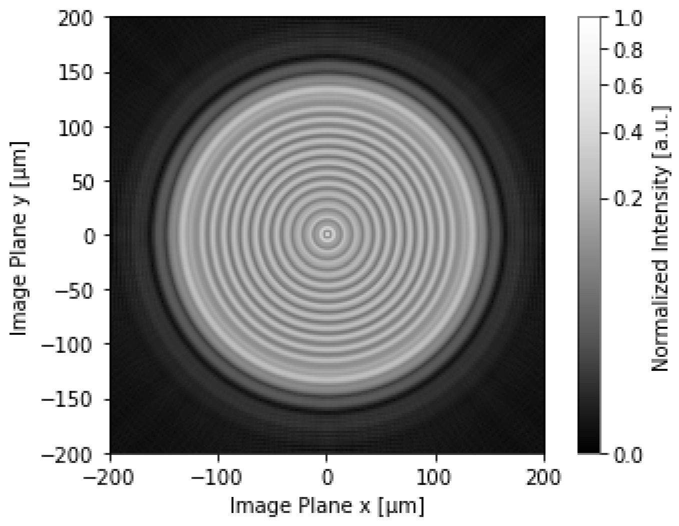
The ripples are Fresnel rings and are a consequence interference very similar to the Gibbs phenomenon. They disappear the farther out of focus you go, and can be wiped away by coarser sampling with an image sensor, or significantly polychromatic light.

We used a value of Q below 2 (worse than Nyquist sampled) because the image is so out of focus, it largely lacks high frequency detail to be aliased, so the choice of Q has minimal consequence.

What if the pupil had some ripples in it from structured errors on optical surfaces in the system?

```
[3]: pupil2 = pupil.copy()

     # 15 cycles per aperture, lambda/25 RMS amplitude
     const = 2 * np.pi * 15
     phase_mod = np.sin(pupil.rho * const) * (np.sqrt(2) / 25)
     pupil2.phase += phase_mod

     ps = PSF.from_pupil(pupil2, efl=50, Q=1.25)
     ps.plot2d(axlim=200, power=3)
```

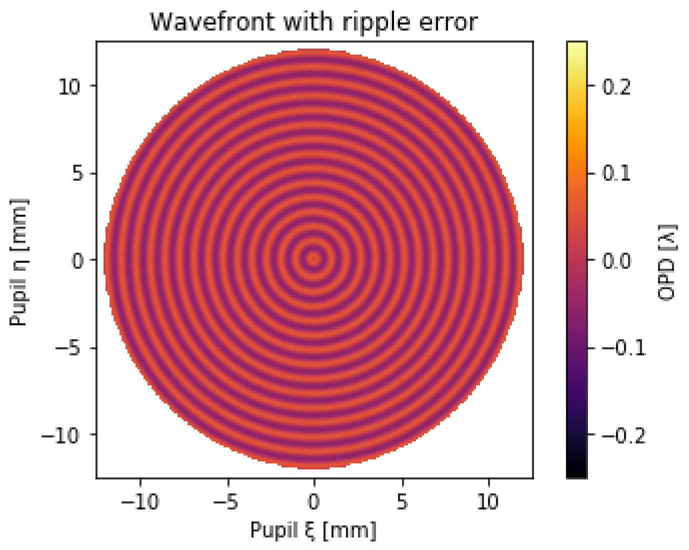
```
[3]: (<Figure size 432x288 with 2 Axes>,
      <matplotlib.axes._subplots.AxesSubplot at 0x7fab158c45f8>)
```

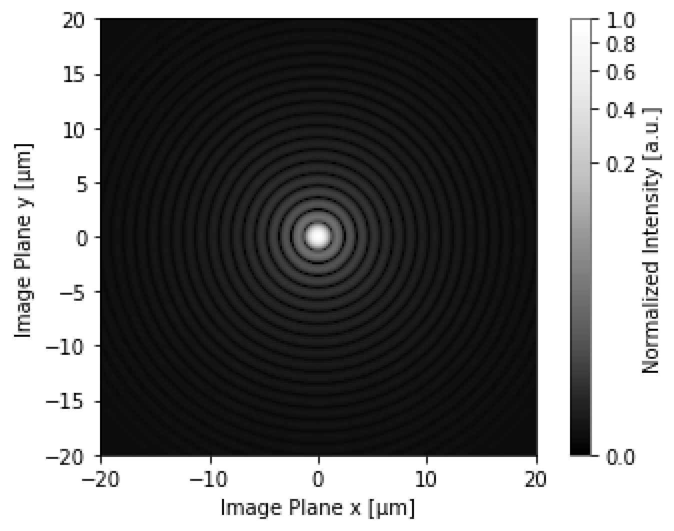


The ripples print through into the in-focus image. What if the image was in focus?

```
[4]: pupil3 = NollZernike(samples=384, dia=25)   # same parameters as before, but no error
     pupil3.phase += phase_mod

     pupil3.plot2d(clim=0.25)   # +/- 138 nm
     plt.title('Wavefront with ripple error')

     ps = PSF.from_pupil(pupil3, efl=50, Q=3) # Q=2 now, we need high resolution
     ps.plot2d(axlim=20, power=4)
```

```
[4]: (<Figure size 432x288 with 2 Axes>,
      <matplotlib.axes._subplots.AxesSubplot at 0x7fab0ffbcba8>)
```



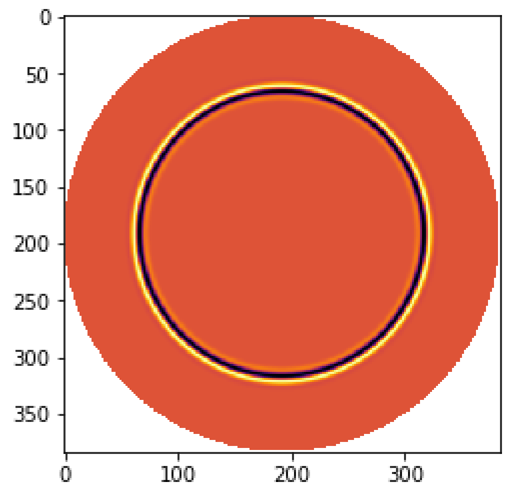


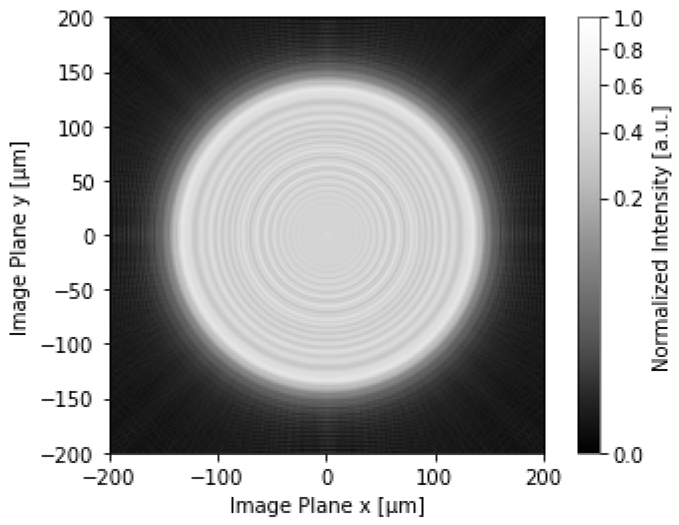
This looks just like an Airy disk; the ripples have low RMS amplitude, so they do little damage to the in-focus image.

What if the ripples were more localized, occuring in just one band within the clear aperture?

[5]:
```
def gauss_r(r, center, sigma=0.1, amplitude=1):
    numerator = (r-center) ** 2
    denominator = 2 * sigma ** 2
    return amplitude * np.exp(-numerator / denominator)

phase_mod2 = phase_mod * gauss_r(pupil.rho, center=0.66, sigma=0.02)
phase_mod_vis = phase_mod2.copy()
phase_mod_vis[pupil._mask == 0] = np.nan
plt.imshow(phase_mod_vis, cmap='inferno')

pupil4 = pupil.copy()
pupil4.phase += phase_mod2

ps = PSF.from_pupil(pupil4, efl=50, Q=1.25) # Q=2 now, we need high resolution
ps.plot2d(axlim=200, power=3)
```

[5]:
```
(<Figure size 432x288 with 2 Axes>,
 <matplotlib.axes._subplots.AxesSubplot at 0x7fab0fe489b0>)
```





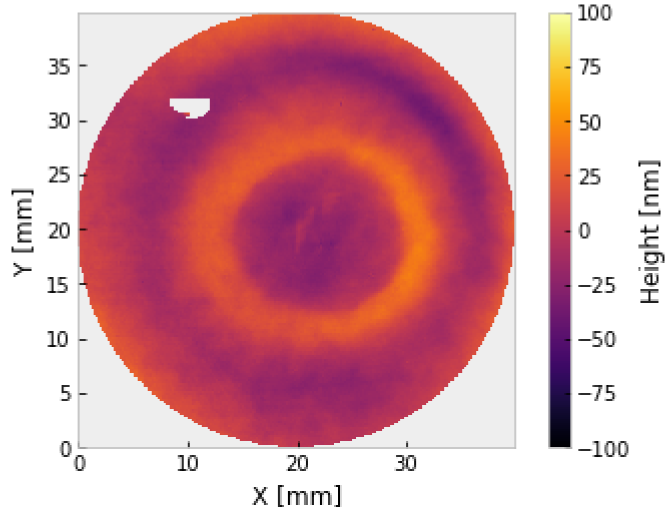
Now there is an inner ring in addition to the Fresnel rings. The formation of this is related to the angular spectrum of the aperture. An ensemble of rays from the perturbed annulus of the aperture propagate at a different angle to the rest; the interference begins at the radius they appear at within the clear aperture, crosses through the center at focus where they overlap with the natural interference from the support of the aperture, then dissipate to an infinite distance as the observation plane moves further behind focus.

4.1.6 Split Transform Method

In this example we will demonstrate the split transform method for analyzing the imaging performance of a mirror. We begin as usual by importing the relevant classes.

```
[1]: from prysm import Interferogram, Pupil, PSF, sample_files

     from matplotlib import pyplot as plt
     %matplotlib inline
     plt.style.use('bmh')
```

```
[2]: p = sample_files('dat')  # sample Zygo .dat file, will be downloaded on demand and␣
     ↪saved locally
     i = Interferogram.from_zygo_dat(p)
     i.crop().mask('circle', 40).crop()
     i.remove_piston_tiptilt_power()
     i.plot2d(clim=100, interp_method=None)  # verify the phase looks OK
     plt.grid(False)
```



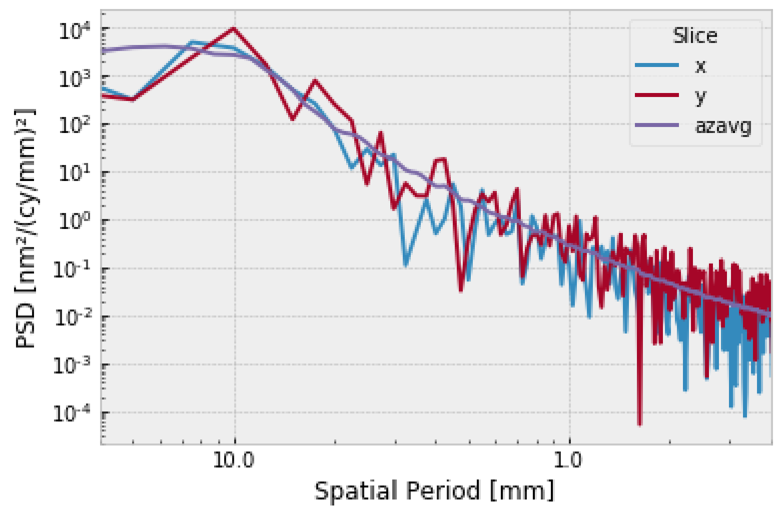
If you are dissatisfied with the masking prowess of prysm, it is recommended to use the software that came with your interferometer. The order of operations from here is very important. Because prysm modifies these classes in-place, we should propagate the PSF before filling the interferogram for PSF analysis.

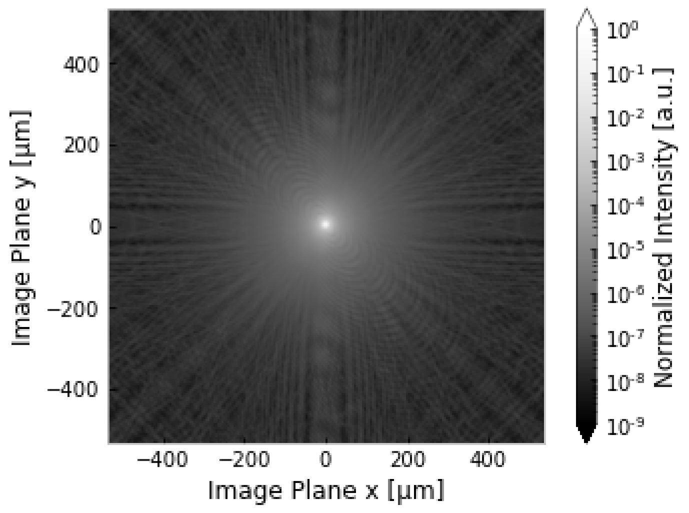
```
[3]: pu = Pupil.from_interferogram(i)
     psf = PSF.from_pupil(pu, efl=200, Q=2)  # F/2
     i.fill()
```

```
[3]: <prysm.interferogram.Interferogram at 0x7f6a5508dc50>
```

We can then plot,

```
[4]: fig, ax = plt.subplots()
     i.plot_psd_slices(fig=fig, ax=ax, mode='period', xlim=(25, 0.25), lw=2)
     fig, ax = plt.subplots()
     # bicubic is highly recommended when the view is small with many pixels
     psf.plot2d(axlim=psf.support / 2,
                clim=(1e-9,1e0),
                interp_method='bicubic',
                fig=fig, ax=ax)
     plt.grid(False)
```





4.1.7 System Modeling

In this example we will see how to model an end-to-end optical system using prysm. Our system will have both an objective lens or telescope as well as a sensor with an optical low-pass filter. We begin by importing the relevant classes and setting some visual styles:

[1]:
```
from prysm import FringeZernike, PSF, MTF, PixelAperture, OLPF
from matplotlib import pyplot as plt
%matplotlib inline
plt.style.use('bmh')
```

Next we model the PSF of the objective, given its aperture, focal length, and Zernike coefficients for its wavefront, such as from a Shack-Hartmann sensor or interferometer:

[2]:
```
# data from a wavefront sensor, optical design program, etc...
coefficients = [0, 0, 0, 0, 0.1, 0.1, -0.025, -0.025, 0.1]

# a circular aperture inscribed in a square 10mm on a side with 50mm EFL
```

(continues on next page)

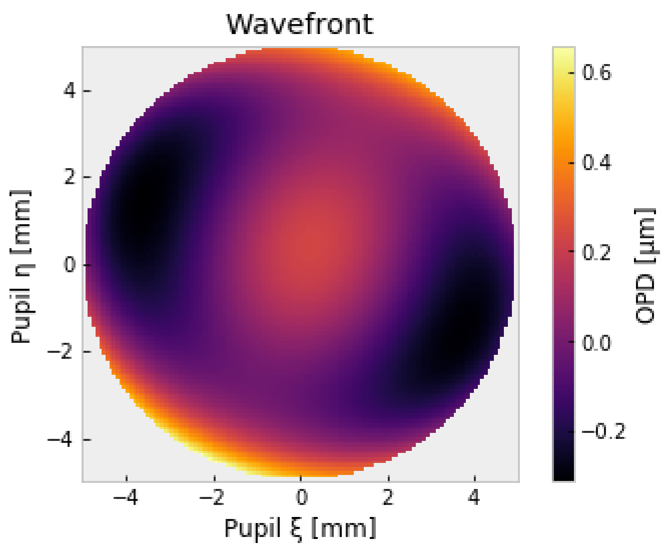
(continued from previous page)

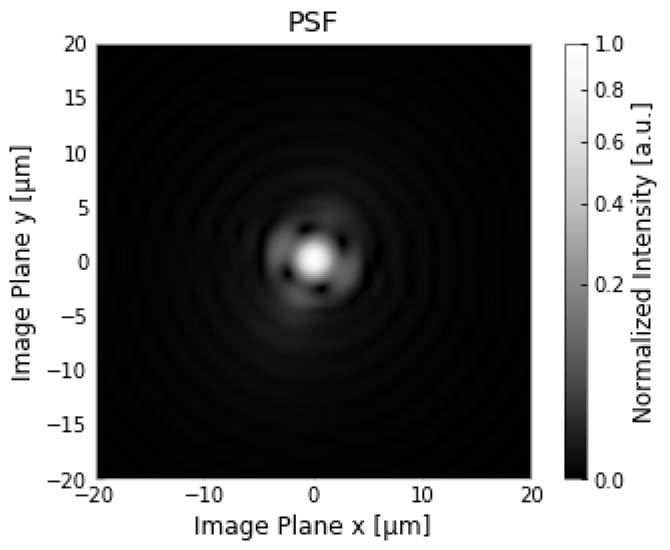
```
# note the default mask is a circle, so the kwarg is somewhat redundant here.
pupil = FringeZernike(coefficients, dia=10, mask='circle', opd_unit='um', norm=True)
psf = PSF.from_pupil(pupil, efl=40)  # F/2
```

Here we have implicitly accepted the default wavelength of 0.5 microns, and Q factor of 2 (Nyquist sampling) which are usually sane defaults. The pupil is circular and is sufficiently described by a Zernike expansion up to Z9.

We can plot the wavefront or PSF of the objective. The wavefront will appear to not quite fill the array, but this is just an artifact of the default lanczos interpolation and relatively few samples.

[3]:
```
fig, ax = pupil.plot2d(interp_method='nearest')
ax.grid(False)
ax.set_title('Wavefront')

fig, ax = psf.plot2d(axlim=20, power=2)  # 1/2 stretch, colorbar scales as well.
ax.grid(False)
ax.set_title('PSF');
```

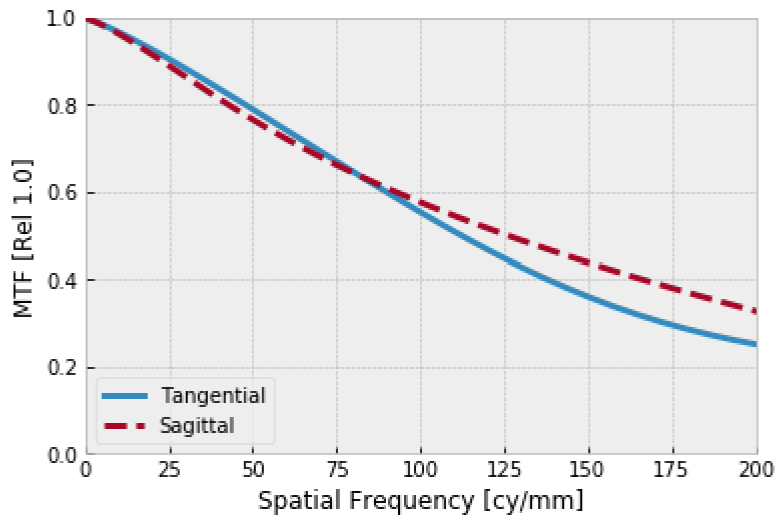




or compute its MTF. Note that “tan” and “sag” here accept the assumption of optical design code that we are looking at an object extended in Y, with no extent in X. For example, this means we could be at an (x,y) field point of (0, 1) degrees. On-axis, tan and sag are simply misgnomers for the “x” and “y” MTFs.

[4]:
```
mtf = MTF.from_psf(psf)
mtf.plot_tan_sag(max_freq=200)
```

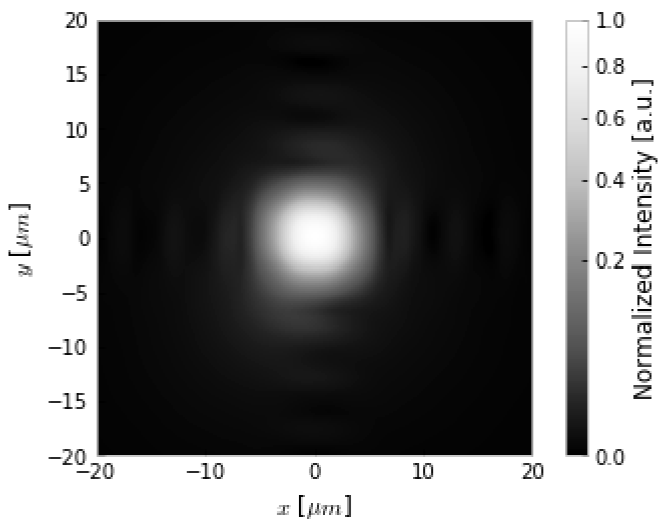
[4]:
```
(<Figure size 432x288 with 1 Axes>,
 <matplotlib.axes._subplots.AxesSubplot at 0x7f68fe901eb8>)
```



[5]:
```
pixel_pitch = 5  # 5 micron diameter pixels
aa_filter = OLPF(pixel_pitch*0.66)
pixel = PixelAperture(pixel_pitch)
sys_psf = psf.conv(aa_filter).conv(pixel).renorm()  # renorm so max=1
```

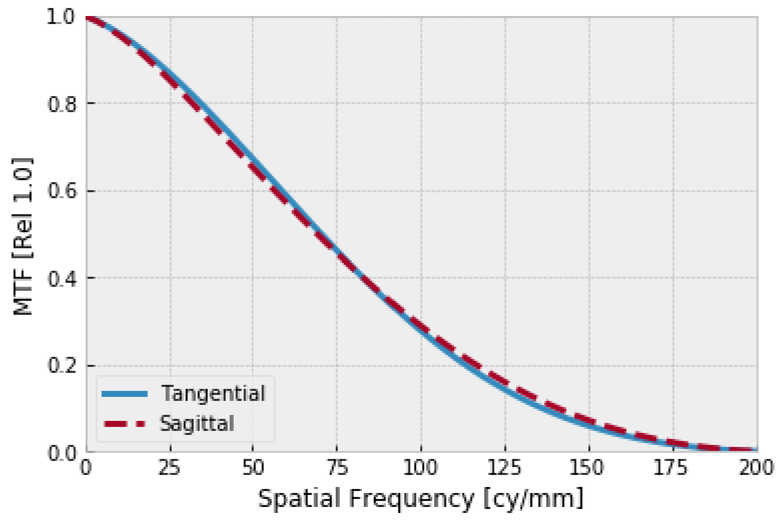
We can plot the system PSF, which is abstract since it includes the pixel aperture. You would not normally look at this, but prysm doesn’t stop you from doing that.

[6]:
```
sys_psf.show(xlim=20, interp_method='lanczos', power=2)  # sys_psf is a Convolvable,␣
↪not a PSF.
plt.grid(False)
```



```
[7]: sys_mtf = MTF.from_psf(sys_psf)
     sys_mtf.plot_tan_sag(max_freq=200)
```

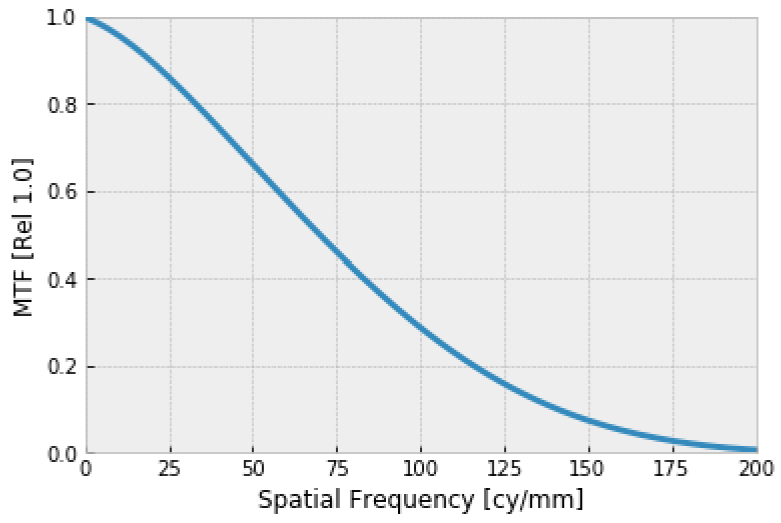
```
[7]: (<Figure size 432x288 with 1 Axes>,
      <matplotlib.axes._subplots.AxesSubplot at 0x7f68fe843470>)
```



We see the system MTF reach zero at 200 cy/mm, as would be expected for a 5 micron pixel. We also see the PSF is significantly squared off, since the pixel aperture contribution is larger than that of the optical system. One might find the azimuthally averaged MTF to be more informative:

```
[8]: sys_mtf.plot_azimuthal_average(max_freq=200)
```

```
[8]: (<Figure size 432x288 with 1 Axes>,
      <matplotlib.axes._subplots.AxesSubplot at 0x7f68fe855be0>)
```



For more information on the classes used, see *Zernikes*, *PSFs*, *MTFs*, and *PixelApertures, OLPFs, and convolutions*