poppy Documentation

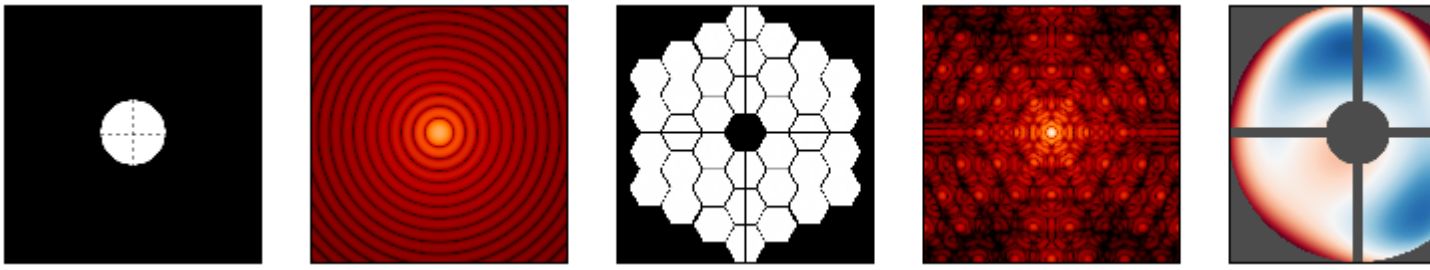
Release 0.8.1dev143.dev143

Marshall Perrin

Mar 20, 2019

Contents

POPPY (Physical Optics Propagation in PYthon) simulates physical optical propagation including diffraction. It implements a flexible framework for modeling Fraunhofer and Fresnel diffraction and point spread function formation, particularly in the context of astronomical telescopes. POPPY was developed as part of a simulation package for JWST, but is broadly applicable to many kinds of imaging simulations.



Part I

Summary

What this software does:

- Allows users to define an optical system consisting of multiple planes, such as pupils and images.
- Provides flexible and extensible optical element classes, including a wide variety of stops, masks, lenses and more.
- Propagates wavefronts between planes using either the *Fraunhofer* or *Fresnel* approximations of scalar electromagnetic theory.
- Computes monochromatic and polychromatic point spread functions through those optics.
- Provides an extensible framework for defining models of optical instruments, including selection of broad- and narrow-band filters, selectable optical components such as pupil stops, models of optical aberrations defined by Zernike polynomials, etc.

What this software does not do:

- Full Rayleigh-Sommerfeld electromagnetic propagation.
- Vector electromagnetic field propagation such as would be needed for modeling polarization effects.
- Modelling of any kind of detector noise or imperfections.

Quickstart IPython Notebook

This documentation is complemented by an IPython Notebook quickstart tutorial.

Downloading and running that notebook is a great way to get started using POPPY. The documentation following here provides greater details on the algorithms and API.

What's New in the latest version?

Part II

Contents

CHAPTER 1

Installation

POPPY may be installed from PyPI in the usual manner for Python packages:

```
% pip install poppy --upgrade
```

The source code is hosted in this repository on GitHub. It is possible to directly install the latest development version from git:

```
% git clone https://github.com/spacetelescope/poppy.git
% cd poppy
% pip install -e .
```

Note: Users at STScI may also access POPPY through the standard SSB software distributions.

1.1 Requirements

- Python 3.5, or more recent. Earlier versions of Python are no longer supported.
- The standard Python scientific stack: `numpy`, `scipy`, `matplotlib`
- POPPY relies upon the astropy community-developed core library for astronomy. astropy, version 1.3 or more recent, is needed.

The following are *optional*. The first, `pysynphot`, is recommended for most users. The other optional installs are only worth adding for speed improvements if you are spending substantial time running calculations.

- pysynphot enables the simulation of PSFs with proper spectral response to realistic source spectra. Without this, PSF fidelity is reduced. See below for *installation instructions for pysynphot*.
- psutil enables slightly better automatic selection of numbers of processes for multiprocess calculations.
- pyFFTW. The FFTW library can speed up the FFTs used in multi-plane optical simulations such as coronagraphiy or slit spectroscopy. Since direct imaging simulations use a discrete matrix FFT instead, direct imaging

simulation speed is unchanged. pyFFTW is recommended if you expect to perform many coronagraphic calculations, particularly for MIRI. (Note: POPPY previously made use of the PyFFTW3 package, which is *different* from pyFFTW. The latter is more actively maintained and supported today, hence the switch. Note also that some users have reported intermittent stability issues with pyFFTW for reasons that are not yet clear.) *At this time we recommend most users should skip installing pyFFTW while getting started with poppy*.

- Anaconda accelerate and numexpr. These optionally can provide improved performance particularly in the Fresnel code.

1.2 Installing or updating pysynphot

Pysynphot is an optional dependency, but is highly recommended. See the pysynphot installation docs here to install `pysynphot` and (at least some of) its CDBS data files.

The minimum needed to have stellar spectral models available for use when creating PSFs is pysynphot itself plus just one of the CDBS data files: the Castelli & Kurucz stellar atlas, file synphot3.tar.gz (18 MB). Feel free to ignore the rest of the synphot CDBS files unless you know you want a larger set of input spectra or need the reference files for other purposes.

1.3 Testing your installation of poppy

Poppy includes a suite of unit tests that exercise its functionality and verify outputs match expectations. You can optionally run this test suite to verify that your installation is working properly:

```
>>> import poppy
>>> poppy.test()
============================= test session starts ==================================
Python 3.6.5, pytest-3.6.1, py-1.5.3, pluggy-0.6.0
Running tests with Astropy version 3.0.3.
... [etc] ...
================= 126 passed, 1 skipped, 1 xfailed in 524.68 seconds ==================
```

Some tests may be automatically skipped depending on whether certain optional packaged are installed, and other tests in development may be marked "expected to fail" (`xfail`), but as long as no tests actually fail then your installation is working as expected.

CHAPTER 2

Release Notes

For a list of contributors, see *About POPPY*.

2.1 0.8.0

2018 December 15

Py2.7 support and deprecated function names removed

As previously announced, support for Python 2 has been removed in this release, as have the deprecated non-PEP8-compliant function names.

New Functionality:

- The `zernike` submodule has gained better support for dealing with wavefront error defined over segmented apertures. The `Segment_Piston_Basis` and `Segment_PTT_Basis` classes implement basis functions for piston-only or piston/tip/tilt motions of arbitrary numbers of hexagonal segments. The `opd_expand_segments` function implements a version of the `opd_expand_orthonormal` algorithm that has been updated to correctly handle disjoint (non-overlapping support) basis functions defined on individual segments. (mperrin)
- Add new `KnifeEdge` optic class representing a sharp opaque half-plane, and a `CircularPhaseMask` representing a circular region with constant optical path difference. (#273, @mperrin)
- Fresnel propagation can now automatically resample wavefronts onto the right pixel scales at Detector objects, same as Fraunhofer propagation. (#242, #264, @mperrin)
- The `display_psf` function now can also handle datacubes produced by `calc_datacube` (#265, @mperrin)

Documentation:

- Various documentation improvements and additions, in particular including a new "Available Optics" page showing visual examples of all the available optical element classes.

Bug Fixes and Software Infrastructure Updates:

- Removal of Python 2 compatibility code, Python 2 test cases on Travis, and similar (#239, @mperrin)
- Removal of deprecated non-PEP8 function names (@mperrin)
- Fix for output PSF formatting to better handle variable numbers of extensions (#219, @shanosborne)
- Fix for FITSOpticalElement opd_index parameter for selecting slices in datacubes (@mperrin)
- Fix inconsistent sign of rotations for FITSOpticalElements vs. other optics (#275, @mperrin)
- Cleaned up the logic for auto-choosing input wavefront array sizes (#274, @mperrin)
- Updates to Travis doc build setup (#270, @mperrin, robelgeda)
- Update package organization and documentation theme for consistency with current STScI package template (#267, #268, #278, @robelgeda)
- More comprehensive unit tests for Fresnel propagation. (#191, #251, #264, @mperrin)
- Update astropy-helpers to current version, and install bootstrap script too (@mperrin, @jhunkeler)
- Minor: doc string correction in FresnelWavefront (@sdwill), fix typo in some error messages (#255, @douglase), update some deprecated logging function calls (@mperrin).

2.2 0.7.0

2018 May 30

Python version support: Future releases will require Python 3.

Please note, this is the *final* release to support Python 2.7. All future releases will require Python 3.5+. See here for more information on migrating to Python 3.

Deprecated function names will go away in next release.

This is also the *final* release to support the older, deprecated function names with mixed case that are not compatible with the Python PEP8 style guide (e.g. `calcPSF` instead of `calc_psf`, etc). Future versions will require the use of the newer syntax.

Performance Improvements:

- Major addition of GPU-accelerated calculations for FFTs and related operations in many propagation calculations. GPU support is provided for both CUDA (NVidia GPUs) and OpenCL (AMD GPUs); the CUDA implementation currently accelerates a slightly wider range of operations. Obtaining optimal performance, and understanding tradeoffs between numpy, FFTW, and CUDA/OpenCL, will in general require tests on your particular hardware. As part of this, much of the FFT infrastructure has been refactored out of the Wavefront classes and into utility functions in `accel_math.py`. This functionality and the resulting gains in performance are described more in Douglas & Perrin, Proc. SPIE 2018. (#239, @douglase), (#250, @mperrin and @douglase).
- Additional performance improvements to other aspects of calculations using the `numexpr` package. Numexpr is now a *highly recommended* optional installation. It may well become a requirement in a future release. (#239, #245, @douglase)
- More efficient display of AnalyticOptics, avoiding unnecessary repetition of optics sampling. (@mperrin)
- Single-precision floating point mode added, for cases that do not require the default double precision floating point and can benefit from the increased speed. (Experimental / beta; some intermediate calculations may still be done in double precision, thus reducing speed gains).

New Functionality:

- New `PhysicalFresnelWavefront` class that uses physical units for the wavefront (e.g. volts/meter) and intensity (watts). See this notebook for examples and further discussion. (#248, @daphil).
- `calc_psf` gains a new parameter to request returning the complex wavefront (#234,@douglase).
- Improved handling of irregular apertures in WFE basis functions (`zernike_basis`, `hexike_basis`, etc.) and the `opd_expand`/`opd_expand_nonorthonormal` fitting functions (@mperrin).
- Added new function `measure_radius_at_ee` which finds the radius at which a PSF achieves some given amount of encircled energy; in some sense an inverse to `measure_ee`. (#244, @shanosborne)
- Much improved algorithm for `measure_fwhm`: the function now works by fitting a Gaussian rather than interpolating between a radial profile on fixed sampling. This yields much better results on low-sampled or under-sampled PSFs. (@mperrin)
- Add `ArrayOpticalElement` class, providing a cleaner interface for creating arbitrary optics at runtime by generating numpy ndarrays on the fly and packing them into an ArrayOpticalElement. (@mperrin)
- Added new classes for deformable mirrors, including both `ContinuousDeformableMirror` and `HexSegmentedDeformableMirror` (@mperrin).

Bug Fixes and Software Infrastructure Updates:

- The Instrument class methods and related API were updated to PEP8-compliant names. Old names remain for back compatibility, but are deprecated and will be removed in the next release. Related code cleanup for better PEP8 compliance. (@mperrin)
- Substantial update to semi-analytic fast coronagraph propagation to make it more flexible about optical plane setup. Fixes #169 (#169, @mperrin)
- Fix for integer vs floating point division when padding array sizes in some circumstances (#235, @exowanderer, @mperrin)
- Fix for aperture clipping in `zernike.arbitrary_basis` (#241, @kvangorkom)
- Fix / documentation fix for divergence angle in the Fresnel code (#237, @douglase). Note, the `divergence` function now returns the *half angle* rather than the *full angle*.
- Fix for `markcentroid` and `imagecrop` parameters conflicting in some cases in `display_psf` (#231, @mperrin)
- For FITSOpticalElements with both shift and rotation set, apply the rotation first and then the shift for more intuitive UI (@mperrin)
- Misc minor doc and logging fixes (@mperrin)
- Increment minimal required astropy version to 1.3, and minimal required numpy version to 1.10; and various related Travis CI setup updates. Also added numexpr test case to Travis. (@mperrin)
- Improved unit test for Fresnel model of Hubble Space Telescope, to reduce memory usage and avoid CI hangs on Travis.
- Update `astropy-helpers` submodule to current version; necessary for compatibility with recent Sphinx releases. (@mperrin)

2.3 0.6.1

2017 August 11

- Update `ah_bootstrap.py` to avoid an issue where POPPY would not successfully install when pulled in as a dependency by another package (@josephoenix)

2.4 0.6.0

2017 August 10

- WavefrontError and subclasses now handle tilts and shifts correctly (#229, @mperrin) Thanks @corcoted for reporting!
- Fix the `test_zernikes_rms` test case to correctly take the absolute value of the RMS error, support `outside=` for `hexike_basis`, enforce which arguments are required for `zernike()`. (#223, @mperrin) Thanks to @kvangorkom for reporting!
- Bug fix for stricter Quantity behavior (`UnitTypeError`) in Astropy 2.0 (@mperrin)
- Added an optional parameter "mergemode" to CompoundAnalyticOptic which provides two ways to combine AnalyticOptics: `mergemode="and"` is the previous behavior (and new default), `mergemode="or"` adds the transmissions of the optics, correcting for any overlap. (#227, @corcoted)
- Add HexagonFieldStop optic (useful for making hexagon image masks for JWST WFSC, among other misc tasks.) (@mperrin)
- Fix behavior where `zernike.arbitrary_basis` would sometimes clip apertures (#222, @kvangorkom)
- Fix `propagate_direct` in fresnel wavefront as described in issue #216 <https://github.com/spacetelescope/poppy/issues/216>_ (#218, @maciekgroch)
- `display_ee()` was not passing the `ext=` argument through to `radial_profile()`, but now it does. (#220, @josephoenix)
- Fix displaying planes where `what='amplitude'` (#217, @maciekgroch)
- Fix handling of FITSOpticalElement big-endian arrays to match recent changes in SciPy (@mperrin) Thanks to @douglase for reporting!
- `radial_profile` now handles `nan` values in radial standard deviations (#214, @douglase)
- The FITS header keywords that are meaningful to POPPY are now documented in fitsheaders and a new `PIXUNIT` keyword encodes "units of the pixels in the header, typically either *arcsecond* or *meter*" (#205, @douglase)
- A typo in the handling of the `markcentroid` argument to `display_psf` is now fixed (so the argument can be set `True`) (#211, @josephoenix)
- `radial_profile` now accepts an optional `pa_range=` argument to specify the [min, max] position angles to be included in the radial profile. (@mperrin)
- Fixes in POPPY to account for the fact that NumPy 1.12+ raises an `IndexError` when non-integers are used to index an array (#203, @kmdouglass)
- POPPY demonstration notebooks have been refreshed by @douglase to match output of the current code

2.5 0.5.1

2016 October 28

- Fix ConfigParser import (see astropy/package-template#172)
- Fixes to formatting of `astropy.units.Quantity` values (#171, #174, #179; @josephoenix, @neilzim)
- Fixes to `fftw_save_wisdom` and `fftw_load_wisdom` (#177, #178; @mmecthley)
- Add `calc_datacube` method to `poppy.Instrument` (#182; @mperrin)
- Test for Apple Accelerate more narrowly (#176; @mperrin)

- `Wavefront.display()` correctly handles `vmin` and `vmax` args (#183; @neilzim)
- Changes to Travis-CI configuration (#197; @etollerud)
- Warn on requested field-of-view too large for pupil sampling (#180; reported by @mmechtley, addressed by @mperrin)
- Bugfix for `add_detector` in `FresnelOpticalSystem` (#193; @maciekgroch)
- Fixes to unit handling and short-distance propagation in `FresnelOpticalSystem` (#194; @maciekgroch, @douglase, @mperrin)
- PEP8 renaming for `poppy.fresnel` for consistency with the rest of POPPY: `propagateTo` becomes `propagate_to`, `addPupil` and `addImage` become `add_pupil` and `add_image`, `inputWavefront` becomes `input_wavefront`, `calcPSF` becomes `calc_psf` (@mperrin)
- Fix `display_psf(..., markcentroid=True)` (#175, @josephoenix)

2.6 0.5.0

2016 June 10

Several moderately large enhancements, involving lots of under-the-hood updates to the code. (*While we have tested this code extensively, it is possible that there may be some lingering bugs. As always, please let us know of any issues encountered via 'the github issues page <https://github.com/spacetelescope/poppy/issues/>'_.*)

- Increased use of `astropy.units` to put physical units on quantities, in particular wavelengths, pixel scales, etc. Instead of wavelengths always being implicitly in meters, you can now explicitly say e.g. `wavelength=1*u.micron`, `wavelength=500*u.nm`, etc. You can also generally use Quantities for arguments to OpticalElement classes, e.g. `radius=2*u.cm`. This is *optional*; the API still accepts bare floating-point numbers which are treated as implicitly in meters. (#145, #165; @mperrin, douglase)
- The `getPhasor` function for all OpticalElements has been refactored to split it into 3 functions: `get_transmission` (for electric field amplitude transmission), `get_opd` (for the optical path difference affectig the phase), and `get_phasor` (which combines transmission and OPD into the complex phasor). This division simplifies and makes more flexible the subclassing of optics, since in many cases (such as aperture stops) one only cares about setting either the transmission or the OPD. Again, there are back compatibility hooks to allow existing code calling the deprecated `getPhasor` function to continue working. (#162; @mperrin, josephoenix)
- Improved capabilities for handling complex coordinate systems:
  - Added new `CoordinateInversion` class to represent a change in orientation of axes, for instance the flipping "upside down" of a pupil image after passage through an intermediate image plane.
  - `OpticalSystem.input_wavefront()` became smart enough to check for `CoordinateInversion` and `Rotation` planes, and, if the user has requested a source offset, adjust the input tilts such that the source will move as requested in the final focal plane regardless of intervening coordinate transformations.
  - `FITSOpticalElement` gets new options `flip_x` and `flip_y` to flip orientations of the file data.
- Update many function names for PEP8 style guide compliance. For instance `calc_psf` replaces `calcPSF`. This was done with back compatible aliases to ensure that existing code continues to run with no changes required at this time, but *at some future point* (but not soon!) the older names will go away, so users are encouranged to migrate to the new names. (@mperrin, josephoenix)

And some smaller enhancements and fixes:

- New functions for synthesis of OPDs from Zernike coefficients, iterative Zernike expansion on obscured apertures for which Zernikes aren't orthonormal, 2x faster optimized computation of Zernike basis sets, and compu-

tation of hexike basis sets using the alternate ordering of hexikes used by the JWST Wavefront Analysis System software. (@mperrin)

- New function for orthonormal Zernike-like basis on arbitrary aperture (#166; Arthur Vigan)
- Flip the sign of defocus applied via the `ThinLens` class, such that positive defocus means a converging lens and negative defocus means diverging. (#164; @mperrin)
- New `wavefront_display_hint` optional attribute on OpticalElements in an OpticalSystem allows customization of whether phase or intensity is displayed for wavefronts at that plane. Applies to `calc_psf` calls with `display_intermediates=True`. (@mperrin)
- When displaying wavefront phases, mask out and don't show the phase for any region with intensity less than 1/100th of the mean intensity of the wavefront. This is to make the display less visually cluttered with near-meaningless noise, especially in cases where a Rotation has sprayed numerical interpolation noise outside of the true beam. The underlying Wavefront values aren't affected at all, this just pre-filters a copy of the phase before sending it to matplotlib.imshow. (@mperrin)
- remove deprecated parameters in some function calls (#148; @mperrin)

2.7 0.4.1

2016 Apr 4:

Mostly minor bug fixes:

- Fix inconsistency between older deprecated `angle` parameter to some optic classes versus new `rotation` parameter for any AnalyticOpticalElement (#140; @kvangorkom, @josephoenix, @mperrin)
- Update to newer API for `psutil` (#139; Anand Sivaramakrishnan, @mperrin)
- "measure_strehl" function moved to `webbpsf` instead of `poppy`. (#138; Kathryn St.Laurent, @josephoenix, @mperrin)
- Add special case to handle zero radius pixel in circular BandLimitedOcculter. (#137; @kvangorkom, @mperrin)
- The output FITS header of an `AnalyticOpticalElement`'s `toFITS()` function is now compatible with the input expected by `FITSOpticalElement`.
- Better saving and reloading of FFTW wisdom.
- Misc minor code cleanup and PEP8 compliance. (#149; @mperrin)

And a few more significant enhancements:

- Added `MatrixFTCoronagraph` subclass for fast optimized propagation of coronagraphs with finite fields of view. This is a related variant of the approach used in the `SemiAnalyticCoronagraph` class, suited for coronagraphs with a focal plane field mask limiting their field of view, for instance those under development for NASA's WFIRST mission. (#128; #147; @neilzim)
- The `OpticalSystem` class now has `npix` and `pupil_diameter` parameters, consistent with the `FresnelOpticalSystem`. (#141; @mperrin)
- Added `SineWaveWFE` class to represent a periodic phase ripple.

2.8 0.4.0

2015 November 20

- **Major enhancement: the addition of Fresnel propagation** (#95, #100, #103, #106, #107, #108, #113, #114, #115, #100, #100; @douglase, @mperrin, @josephoenix) *Many thanks to @douglase for the initiative and code contributions that made this happen.*
- Improvements to Zernike aberration models (#99, #110, #121, #125; @josephoenix)
- Consistent framework for applying arbitrary shifts and rotations to any AnalyticOpticalElement (#7, @mperrin)
- When reading FITS files, OPD units are now selected based on BUNIT header keyword instead of always being "microns" by default, allowing the units of files to be set properly based on the FITS header.
- Added infrastructure for including field-dependent aberrations at an optical plane after the entrance pupil (#105, @josephoenix)
- Improved loading and saving of FFTW wisdom (#116, #120, #122, @josephoenix)
- Allow configurable colormaps and make image origin position consistent (#117, @josephoenix)
- Wavefront.tilt calls are now recorded in FITS header HISTORY lines (#123; @josephoenix)
- Various improvements to unit tests and test infrastructure (#111, #124, #126, #127; @josephoenix, @mperrin)

2.9 0.3.5

2015 June 19

- Now compatible with Python 3.4 in addition to 2.7! (#83, @josephoenix)
- Updated version numbers for dependencies (@josephoenix)
- Update to most recent astropy package template (@josephoenix)
- `AsymmetricSecondaryObscuration` enhanced to allow secondary mirror supports offset from the center of the optical system. (@mperrin)
- New optic `AnnularFieldStop` that defines a circular field stop with an (optional) opaque circular center region (@mperrin)
- display() functions now return Matplotlib.Axes instances to the calling functions.
- `FITSOpticalElement` will now determine if you are initializing a pupil plane optic or image plane optic based on the presence of a `PUPLSCAL` or `PIXSCALE` header keyword in the supplied transmission or OPD files (with the transmission file header taking precedence). (#97, @josephoenix)
- The `poppy.zernike.zernike()` function now actually returns a NumPy masked array when called with `mask_array=True`
- poppy.optics.ZernikeAberration and poppy.optics.ParameterizedAberration have been moved to poppy.wfe and renamed `ZernikeWFE` and `ParameterizedWFE`. Also, ZernikeWFE now takes an iterable of Zernike coefficients instead of (n, m, k) tuples.
- Various small documentation updates
- Bug fixes for:
  - redundant colorbar display (#82)
  - Unnecessary DeprecationWarnings in `poppy.utils.imshow_with_mouseover()` (#53)
  - Error in saving intermediate planes during calculation (#81)
  - Multiprocessing causes Python to hang if used with Apple Accelerate (#23, n.b. the fix depends on Python 3.4)

  - Copy in-memory FITS HDULists that are passed in to FITSOpticalElement so that in-place modifications don't affect the caller's copy of the data (#89)
  - Error in the `poppy.utils.measure_EE()` function produced values for the edges of the radial bins that were too large, biasing EE values and leading to weird interpolation behavior near r = 0. (#96)

2.10 0.3.4

2015 February 17

- Continued improvement in unit testing (@mperrin, @josephoenix)
- Continued improvement in documentation (@josephoenix, @mperrin)
- Functions such as addImage, addPupil now also return a reference to the added optic, for convenience (@josephoenix)
- Multiprocessing code and semi-analytic coronagraph method can now return intermediate wavefront planes (@josephoenix)
- Display methods for radial profile and encircled energy gain a normalization keyword (@douglase)
- matrixDFT: refactor into unified function for all centering types (@josephoenix)
- matrixDFT bug fix for axes parity flip versus FFT transforms (Anand Sivaramakrishnan, @josephoenix, @mperrin)
- Bug fix: Instrument class can now pass through dict or tuple sources to OpticalSystem calc_psf (@mperrin)
- Bug fix: InverseTransmission class shape property works now. (@mperrin)
- Refactor instrument validateConfig method and calling path (@josephoenix)
- Code cleanup and rebalancing where lines had been blurred between poppy and webbpsf (@josephoenix, @mperrin)
- Misc packaging infrastructure improvements (@embray)
- Updated to Astropy package helpers 0.4.4
- Set up integration with Travis CI for continuous testing. See https://travis-ci.org/mperrin/poppy

2.11 0.3.3

2014 Nov

Bigger team!. This release log now includes github usernames of contributors:

- New classes for wavefront aberrations parameterized by Zernike polynomials (@josephoenix, @mperrin)
- ThinLens class now reworked to require explicitly setting an outer radius over which the wavefront is normalized. *Note this is an API change for this class, and will require minor changes in code using this class*. ThinLens is now a subclass of CircularAperture.
- Implement resizing of phasors to allow use of FITSOpticalElements with Wavefronts that have different spatial sampling. (@douglase)
- Installation improvements and streamlining (@josephoenix, @cslocum)
- Code cleanup and formatting (@josephoenix)
- Improvements in unit testing (@mperrin, @josephoenix, @douglase)

- Added normalize='exit_pupil' option; added documentation for normalization options. (@mperrin)
- Bug fix for "FQPM on an obscured aperture" example. Thanks to Github user qisaiman for the bug report. (@mperrin)
- Bug fix to compound optic display (@mperrin)
- Documentation improvements (team)

2.12 0.3.2

Released 2014 Sept 8

- Bug fix: Correct pupil orientation for inverse transformed pupils using PyFFTW so that it is consistent with the result using numpy FFT.

2.13 0.3.1

Released August 14 2014

- **Astropy compatibility updated to 0.4.**
  - Configuration system reworked to accomodate the astropy.configuration transition.
  - Package infrastructure updated to most recent astropy package-template.
- Several OpticalElements got renamed, for instance `IdealCircularOcculter` became just `CircularOcculter`. (*All* the optics in poppy are fairly idealized and it seemed inconsistent to signpost that for only some of them. The explicit 'Ideal' nametag is kept only for the FQPM to emphasize that one in particular uses a very simplified prescription and neglects refractive index variation vs wavelength.)
- Substantially improved unit test system.
- Some new utility functions added in poppy.misc for calculating analytic PSFs such as Airy functions for comparison (and use in the test system).
- Internal code reorganization, mostly which should not affect end users directly.
- Packaging improvements and installation process streamlining, courtesy of Christine Slocum and Erik Bray
- Documentation improvements, in particular adding an IPython notebook tutorial.

2.14 0.3.0

Released April 7, 2014

- Dependencies updated to use astropy.
- Added documentation and examples for POPPY, separate from the WebbPSF documentation.
- Improved configuration settings system, using astropy.config framework.
  - The astropy.config framework itself is in flux from astropy 0.3 to 0.4; some of the related functionality in poppy may need to change in the future.
- Added support for rectangular subarray calculations. You can invoke these by setting fov_pixels or fov_arcsec with a 2-element iterable:

```
>> nc = webbpsf.NIRCam()
>> nc.calc_psf('F212N', fov_arcsec=[3,6])
>> nc.calc_psf('F187N', fov_pixels=(300,100) )
```

Those two elements give the desired field size as (Y,X) following the usual Python axis order convention.

- Added support for pyFFTW in addition to PyFFTW3.
- pyFFTW will auto save wisdom to disk for more rapid execution on subsequent invocations
- InverseTransmission of an AnalyticElement is now allowed inside a CompoundAnalyticOptic
- Added SecondaryObscuration optic to conveniently model an opaque secondary mirror and adjustible support spiders.
- Added RectangleAperture. Added rotation keywords for RectangleAperture and SquareAperture.
- Added AnalyticOpticalElement.sample() function to sample analytic functions onto a user defined grid. Refactored the display() and toFITS() functions. Improved functionality of display for CompoundAnalyticOptics.

2.15 0.2.8

- First release as a standalone package (previously was integrated as part of webbpsf). See the release notes for WebbPSF for prior verions.
- switched package building to use `setuptools` instead of `distutils`/`stsci_distutils_hack`
- new `Instrument` class in poppy provides much of the functionality previously in JWInstrument, to make it easier to model generic non-JWST instruments using this code.

CHAPTER 3

Overview

The module poppy implements an object-oriented system for modeling physical optics propagation with diffraction, particularly for telescopic and coronagraphic imaging. Right now only image and pupil planes are supported; intermediate planes are a future goal.

Poppy also includes a system for modeling a complete instrument (including optical propagation, synthetic photometry, and pointing jitter), and a variety of useful utility functions for analysing and plotting PSFs, documented below.

Key Concepts:

To model optical propagation, poppy implements an object-oriented system for representing an optical train. There are a variety of `OpticalElement` classes representing both physical elements as apertures, mirrors, and apodizers, and also implicit operations on wavefronts, such as rotations or tilts. Each `OpticalElement` may be defined either via analytic functions (e.g. a simple circular aperture) or by numerical input FITS files (e.g. the complex JWST aperture with appropriate per-segment WFE). A series of such `OpticalElement` objects is chained together in order in an `OpticalSystem` class. That class is capable of generating `Wavefront` instances suitable for propagation through the desired elements (with correct array size and sampling), and onto the final image plane.

There is an even higher level class `Instrument` which adds support for selectable instrument mechanisms (such as filter wheels, pupil stops, etc). In particular it adds support for computing via synthetic photometry the appropriate weights for multiwavelength computations through a spectral bandpass filter, and for PSF blurring due to pointing jitter (neither of which effects are modeled by `OpticalSystem`). Given a specified instrument configuration, an appropriate `OpticalSystem` is generated, the appropriate wavelengths and weights are calculated based on the bandpass filter and target source spectrum, the PSF is calculated, and optionally is then convolved with a blurring kernel due to pointing jitter. For instance, all of the WebbPSF instruments are implemented by subclassing `poppy.Instrument`.

3.1 Fraunhofer domain calculations

poppy's default mode assumes that optical propagation can be modeled using Fraunhofer diffraction (the "far field" approximation), such that the relationship between pupil and image plane optics is given by two-dimensional Fourier transforms. (Fresnel propagation is also available, *with slightly different syntax*.)

Two different algorithmic flavors of Fourier transforms are used in Poppy. The familiar FFT algorithm is used for transformations between pupil and image planes in the general case. This algorithm is relatively fast (*O(N log(N))*)

but imposes strict constraints on the relative sizes and samplings of pupil and image plane arrays. Obtaining fine sampling in the image plane requires very large oversized pupil plane arrays and vice versa, and image plane pixel sampling becomes wavelength dependent. To avoid these constraints, for transforms onto the final `Detector` plane, instead a Matrix Fourier Transform (MFT) algorithm is used (See Soummer et al. 2007 Optics Express). This allows computation of the PSF directly on the desired detector pixel scale or an arbitrarily finely subsampled version therof. For equivalent array sizes *N*, the MFT is slower than the FFT(*O(N^3)*), but in practice the ability to freely choose a more appropriate *N* (and to avoid the need for post-FFT interpolation onto a common pixel scale) more than makes up for this and the MFT is faster.

Note: This code makes use of the python standard module `logging` for output information. Top-level details of the calculation are output at level `logging.INFO`, while details of the propagation through each optical plane are printed at level `logging.DEBUG`. See the Python logging documentation for an explanation of how to redirect the poppy logger to the screen, a textfile, or any other log destination of your choice.

3.2 Working with OpticalElements

OpticalElements can be instantiated from FITS files, or created by one of a large number of analytic function definitions implemented as `AnalyticOpticalElement` subclasses. Typically these classes take some number of arguments to set their properties. Once instantiated, any analytic function can be displayed on screen, sampled onto a numerical grid, and/or saved to disk.:

```
>>> ap = poppy.CircularAperture(radius=2)      # create a simple circular aperture
>>> ap.display(what='both')                    # display both intensity and phase components

>>> values = ap.sample(npix=512)               # evaluate on 512 x 512 grid
>>> ap.to_fits('test_circle.fits', npix=1024)  # write to disk as a FITS file with higher sampling
```

When sampling an `AnalyticOpticalElement`, you may choose to obtain various representations of its action on a complex wavefront, including the amplitude transmission; intensity transmission; or phase delay in waves, radians, or meters. See the `AnalyticOpticalElement` class documentation for detailed arguments to these functions.

`OpticalElement` objects have attributes such as `shape` (For a `FITSOpticalElement` the array shape in usual Python (Y,X) order; None for a `AnalyticOpticalElement`), a descriptive `name` string, and size information such as `pixelscale`. The type of size information present depends on the *plane type*.

3.3 Optical Plane Types

An `OpticalSystem` consists of a series of two or more planes, of various types. The plane type of a given `OpticalElement` is encoded by its `planetype` attribute. The allowed types of planes are:

- **Pupil** planes, which have spatial scale measured in meters. For instance a telescope could have a diameter of 1 meter and be represented inside an array 1024x1024 pixels across with pixel scale 0.002 meters/pixel, so that the aperture is a circle filling half the diameter of the array. Pupil planes typically have a `pupil_diam` attribute which, please note, defines the diameter of the *numerical array* (e.g. 2.048 m in this example), rather than whatever subset of that array has nonzero optical transmission.
- **Image** planes, which have angular sampling measured in arcseconds. The default behavior for an image plane in POPPY is to have the sampling automatically defined by the natural sampling of a Fourier Transform of the previous pupil array. This is generally appropriate for most intermediate optical planes in a system. However there are also:

- **Detector** planes, which are a specialized subset of image plane that has a fixed angular sampling (pixel scale). For instance one could compute the PSF of that telescope over a field of view 10 arcseconds square with a sampling of 0.01 arcseconds per pixel.
- **Rotation** planes, which represent a change of coordinate system rotating by some number of degrees around the optical axis. Note that POPPY always represents an "unfolded", linear optical system; fold mirrors and/or other intermediate powered optics are not represented as such. Rotations can take place after either an image or pupil plane.

POPPY thus is capable of representing a moderate subset of optical imaging systems, though it is not intended as a substitute for a professional optics design package such as Zemax or Code V for design of full optical systems.

3.4 Defining your own custom optics

All `OpticalElement` classes must have methods `get_transmission` and `get_opd` which returns the amplitude transmission and optical path delay representing that optic, sampled appropriately for a given input `Wavefront` and at the appropriate wavelength. These are combined together to calculate the complex phasor which is applied to the wavefront's electric field. To define your own custom OpticalElements, you can:

1. Subclass `AnalyticOpticalElement` and write suitable function(s) to describe the properties of your optic,
2. Combine two or more existing `AnalyticOpticalElement` instances as part of a `CompoundAnalyticOptic`, or
3. Generate suitable transmission and optical path difference arrays using some other tool, save them as FITS files with appropriate keywords, and instantiate them as an `FITSOpticalElement`

FITSOpticalElements have separate attributes for amplitude and phase components, which may be read separately from 2 FITS files:

- `amplitude`, the electric field amplitude transmission of the optic
- opd, the optical path difference of the optic

Defining functions on a AnalyticOpticalElement subclass allows more flexibility for amplitude transmission or OPDs to vary with wavelength or other properties.

See *Extending POPPY by defining your own optics and instruments* for more details and examples.

CHAPTER 4

Examples

Let's dive right in to some example code.

(A runnable notebook version of this examples page is included in the `notebooks` subdirectory of the poppy source, or is available from here.)

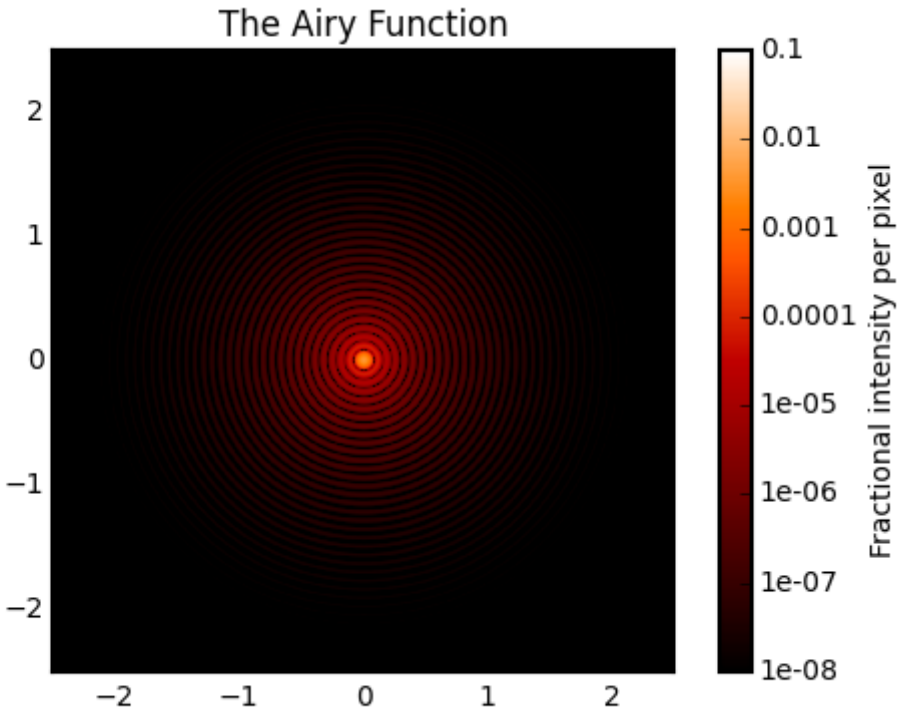
For all of the following examples, you will have more informative text output when running the code if you first enable Python's logging mechanism to display log messages to screen:

```
import logging
logging.basicConfig(level=logging.DEBUG)
```

4.1 A simple circular pupil

This is very simple, as it should be:

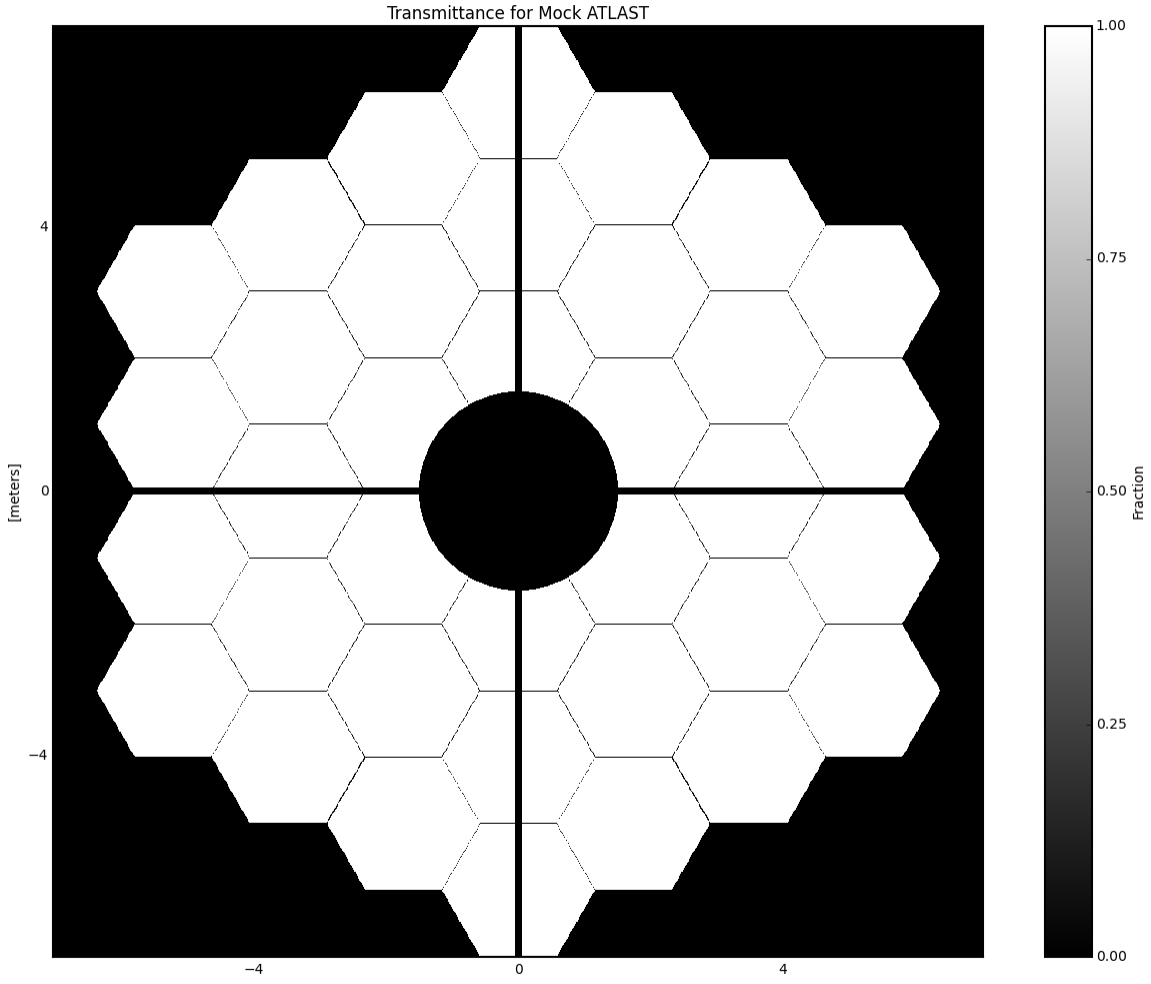
```
osys = poppy.OpticalSystem()
osys.add_pupil( poppy.CircularAperture(radius=3))    # pupil radius in meters
osys.add_detector(pixelscale=0.010, fov_arcsec=5.0)  # image plane coordinates in arcseconds

psf = osys.calc_psf(2e-6)                            # wavelength in microns
poppy.display_psf(psf, title='The Airy Function')
```



4.2 A complex segmented pupil

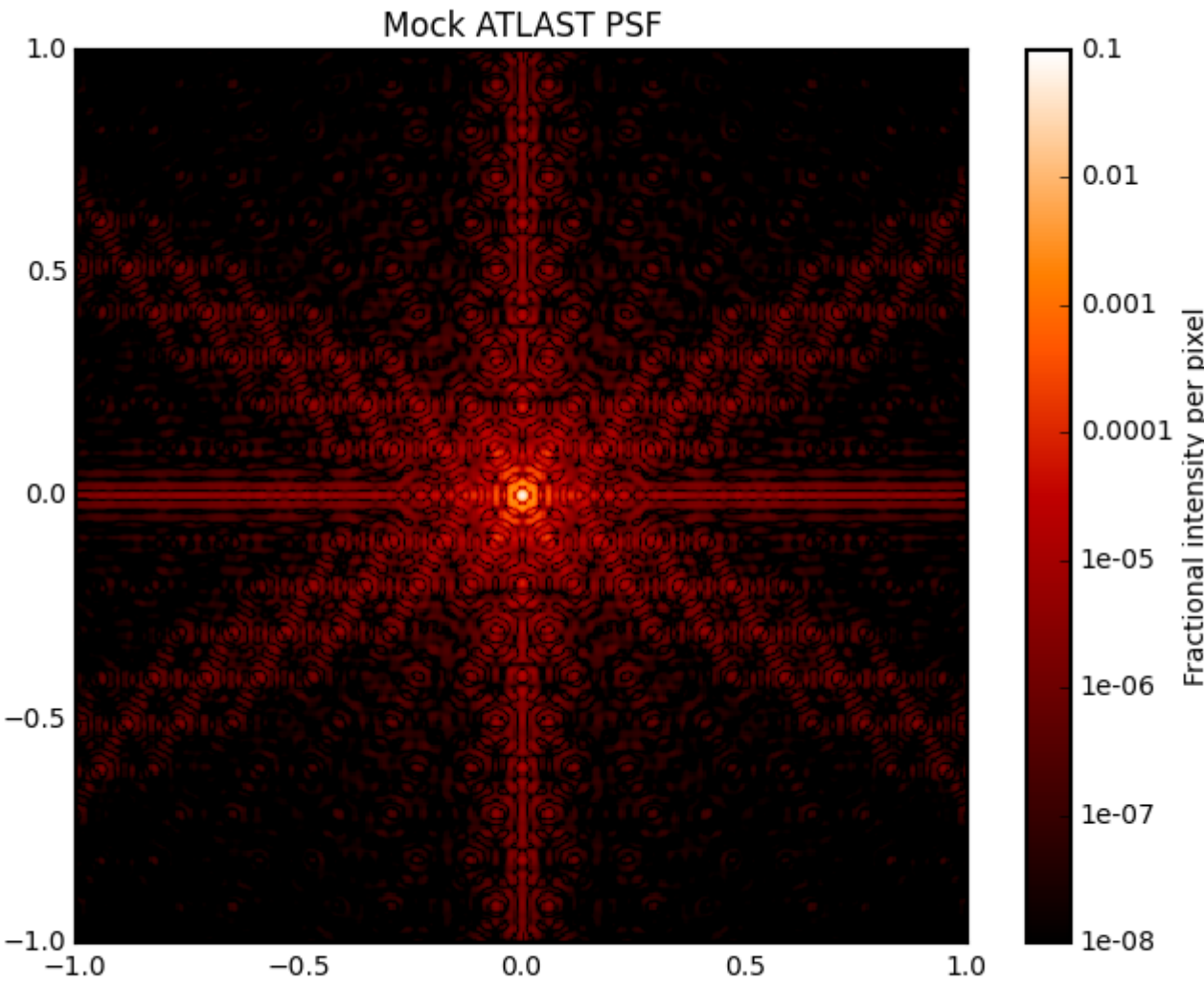
By combining multiple analytic optics together it is possible to create quite complex pupils:

```
ap = poppy.MultiHexagonAperture(rings=3, flattoflat=2)           # 3 rings of 2 m segments yields 14.1␣
↪m circumscribed diameter
sec = poppy.SecondaryObscuration(secondary_radius=1.5, n_supports=4, support_width=0.1)   # secondary␣
↪with spiders
atlast = poppy.CompoundAnalyticOptic( opticslist=[ap, sec], name='Mock ATLAST')           # combine␣
↪into one optic

atlast.display(npix=1024, colorbar_orientation='vertical')
```



And here's the PSF:

```
osys = poppy.OpticalSystem()
osys.add_pupil(atlast)
osys.add_detector(pixelscale=0.010, fov_arcsec=2.0)
psf = osys.calc_psf(1e-6)

poppy.display_psf(psf, title="Mock ATLAST PSF")
```



4.3 Multiple defocused PSFs

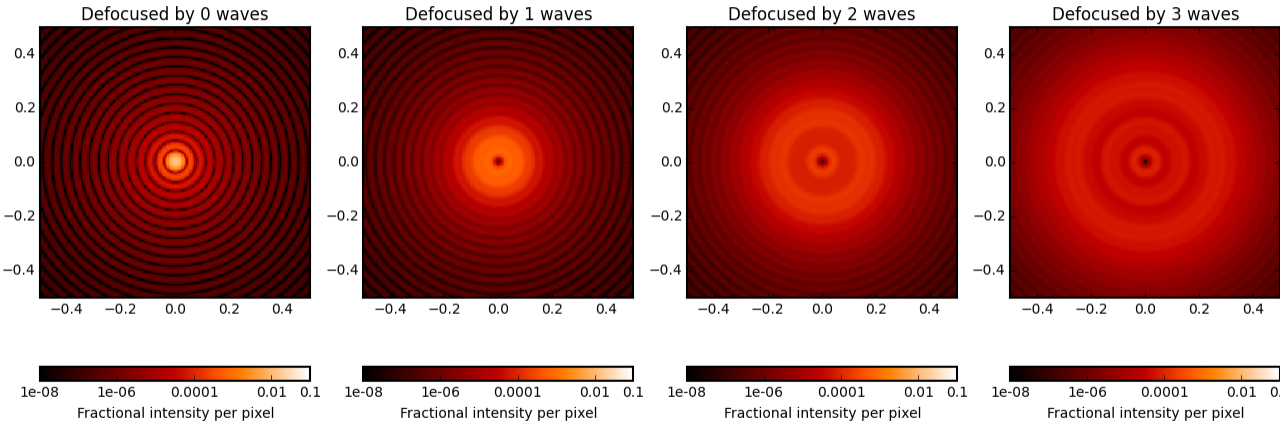
Defocus can be added using a lens:

```
wavelen=1e-6
nsteps = 4
psfs = []
for nwaves in range(nsteps):

    osys = poppy.OpticalSystem("test", oversample=2)
    osys.add_pupil( poppy.CircularAperture(radius=3))    # pupil radius in meters
    osys.add_pupil( poppy.ThinLens(nwaves=nwaves, reference_wavelength=wavelen, radius=3))
    osys.add_detector(pixelscale=0.01, fov_arcsec=4.0)
```

(continues on next page)

(continued from previous page)

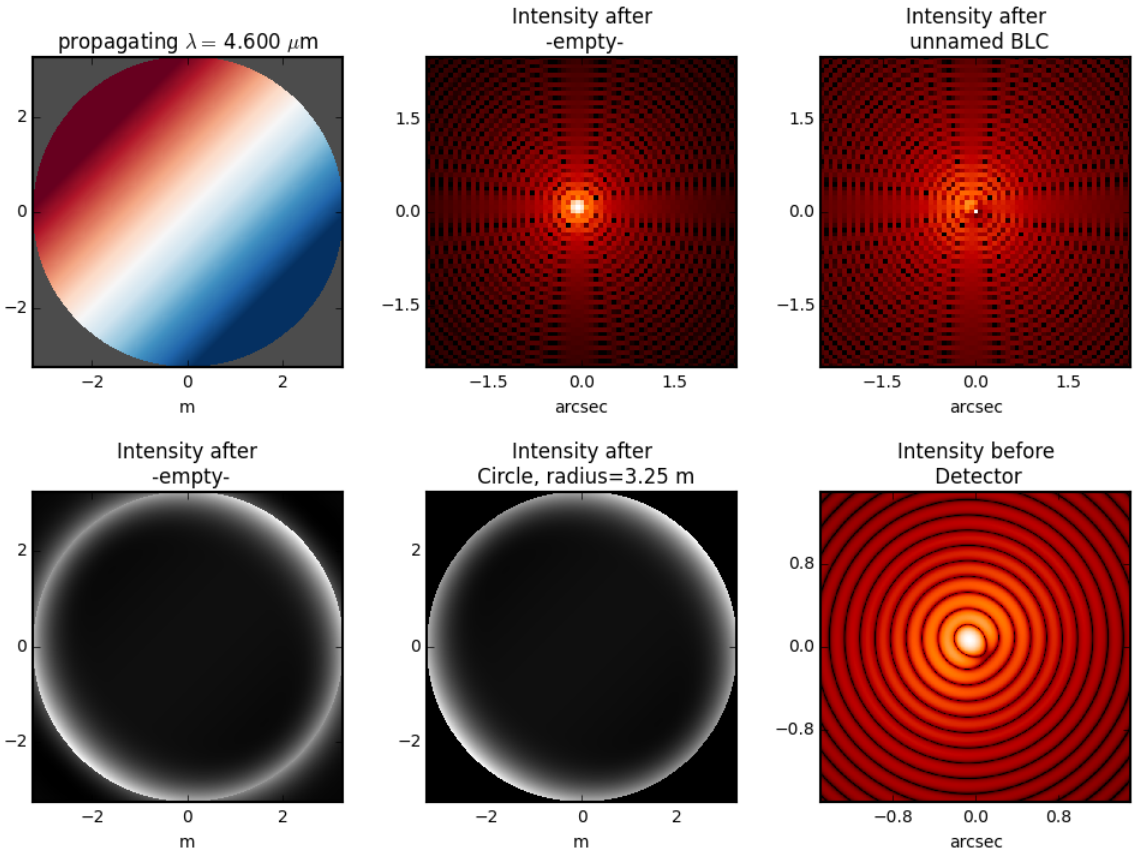
```
    psf = osys.calc_psf(wavelength=wavelen)
    psfs.append(psf)

    plt.subplot(1,nsteps, nwaves+1)
    poppy.display_psf(psf, title='Defocused by {0} waves'.format(nwaves),
        colorbar_orientation='horizontal')
```



4.4 Band Limited Coronagraph with Off-Axis Source

As an example of a more complicated calculation, here's a NIRCam-style band limited coronagraph with the source not precisely centered:

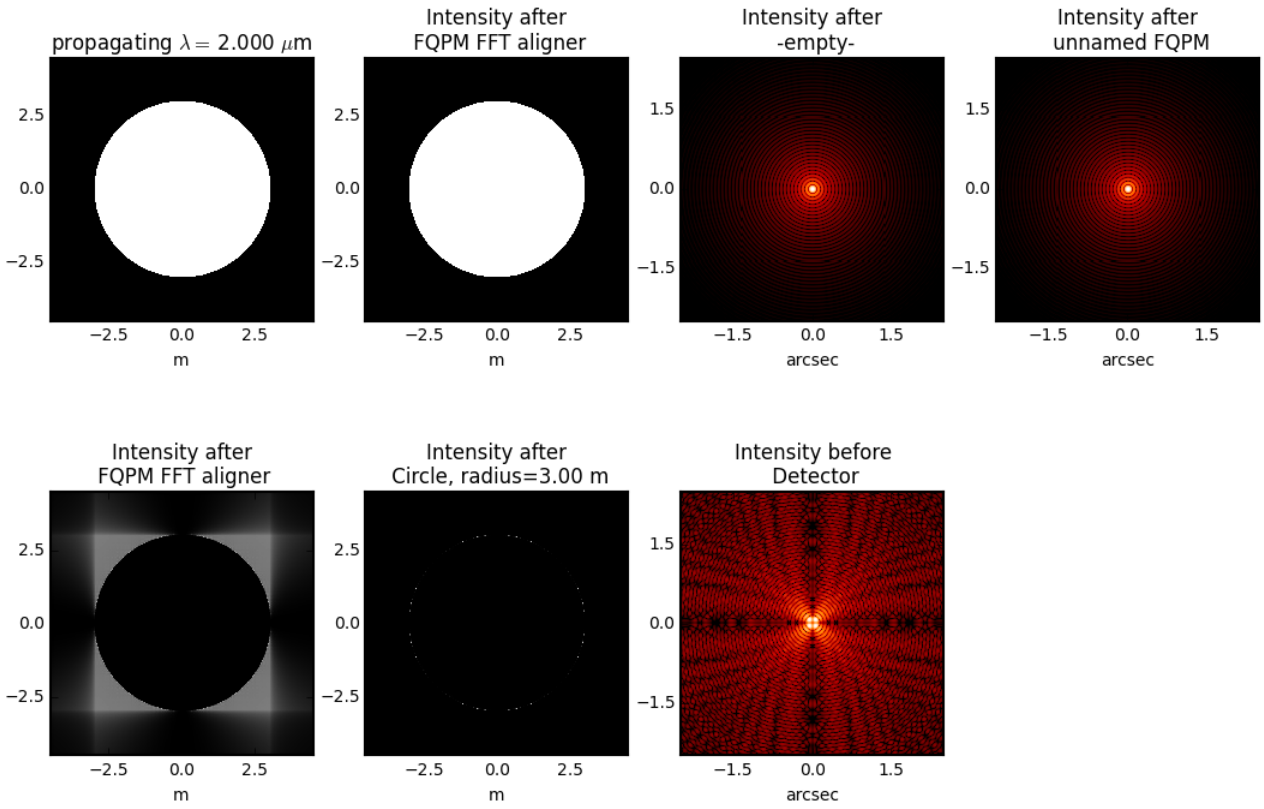
```
oversample=2
pixelscale = 0.010  #arcsec/pixel
wavelength = 4.6e-6

osys = poppy.OpticalSystem("test", oversample=oversample)
osys.add_pupil(poppy.CircularAperture(radius=6.5/2))
osys.add_image()
osys.add_image(poppy.BandLimitedCoron(kind='circular',  sigma=5.0))
osys.add_pupil()
osys.add_pupil(poppy.CircularAperture(radius=6.5/2))
osys.add_detector(pixelscale=pixelscale, fov_arcsec=3.0)

osys.source_offset_theta = 45.
osys.source_offset_r =  0.1  # arcsec
psf = osys.calc_psf(wavelength=wavelength, display_intermediates=True)
```



4.5 FQPM coronagraph

Four quadrant phase mask coronagraphs are a bit more complicated because one needs to ensure proper alignment of the FFT result with the center of the phase mask. This is done using a virtual optic called an 'FQPM FFT aligner' as follows:

```
optsys = poppy.OpticalSystem()
optsys.add_pupil( poppy.CircularAperture( radius=3, pad_factor=1.5)) #pad display area by 50%
optsys.add_pupil( poppy.FQPM_FFT_aligner())   # ensure the PSF is centered on the FQPM cross hairs
optsys.add_image()  # empty image plane for "before the mask"
optsys.add_image( poppy.IdealFQPM(wavelength=2e-6))
optsys.add_pupil( poppy.FQPM_FFT_aligner(direction='backward'))  # undo the alignment tilt after going␣
↪back to the pupil plane
optsys.add_pupil( poppy.CircularAperture( radius=3)) # Lyot mask - change radius if desired
optsys.add_detector(pixelscale=0.01, fov_arcsec=10.0)


psf = optsys.calc_psf(wavelength=2e-6, display_intermediates=True)
```



4.6 FQPM on an Obscured Aperture (demonstrates compound optics)

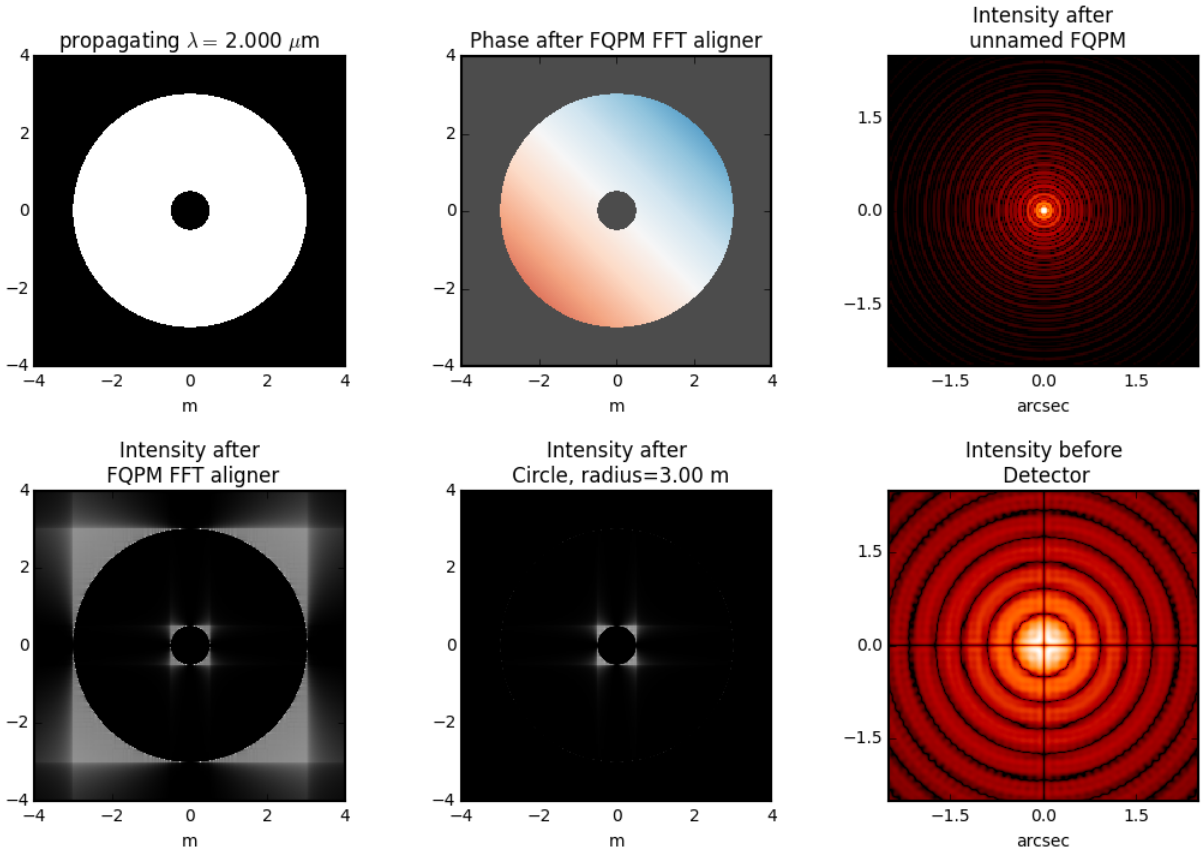
As a variation, we can add a secondary obscuration. This can be done by creating a compound optic consisting of the circular outer aperture plus an opaque circular obscuration. The latter we can make using the InverseTransmission class.

```
primary = poppy.CircularAperture( radius=3)
secondary = poppy.InverseTransmission( poppy.CircularAperture(radius=0.5) )
aperture = poppy.CompoundAnalyticOptic( opticslist = [primary, secondary] )

optsys = poppy.OpticalSystem()
optsys.add_pupil( aperture)
optsys.add_pupil( poppy.FQPM_FFT_aligner())   # ensure the PSF is centered on the FQPM cross hairs
optsys.add_image( poppy.IdealFQPM(wavelength=2e-6))
optsys.add_pupil( poppy.FQPM_FFT_aligner(direction='backward'))  # undo the alignment tilt after going␣
↪back to the pupil plane
optsys.add_pupil( poppy.CircularAperture( radius=3)) # Lyot mask - change radius if desired
optsys.add_detector(pixelscale=0.01, fov_arcsec=10.0)

optsys.display()
```

(continues on next page)

(continued from previous page)

```
psf = optsys.calc_psf(wavelength=2e-6, display_intermediates=True)
```



4.7 Semi-analytic Coronagraph Calculations

In some cases, coronagraphy calculations can be sped up significantly using the semi-analytic algorithm of Soummer et al. This is implemented by first creating an OpticalSystem as usual, and then casting it to a SemiAnalyticCoronagraph class (which has a special customized propagation method implementing the alternate algorithm):

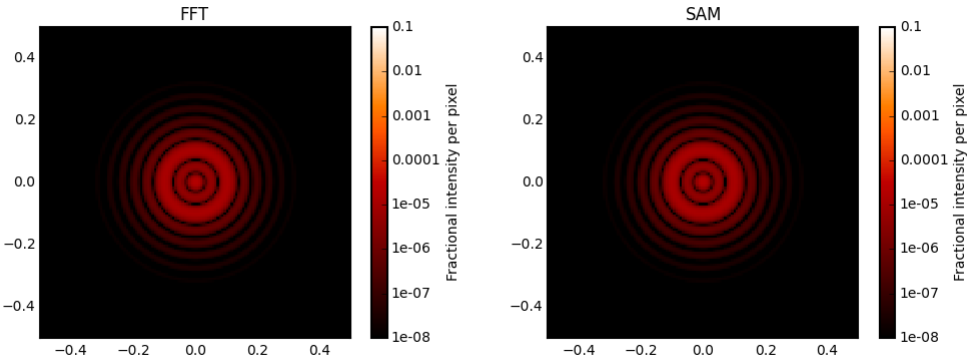
The following code performs the same calculation both ways and compares their speeds:

```
radius = 6.5/2
lyot_radius = 6.5/2.5
pixelscale = 0.060

osys = poppy.OpticalSystem("test", oversample=8)
osys.add_pupil( poppy.CircularAperture(radius=radius), name='Entrance Pupil')
osys.add_image( poppy.CircularOcculter(radius = 0.1) )
osys.add_pupil( poppy.CircularAperture(radius=lyot_radius), name='Lyot Pupil')
osys.add_detector(pixelscale=pixelscale, fov_arcsec=5.0)
```

(continues on next page)

(continued from previous page)

```
plt.figure(1)
sam_osys = poppy.SemiAnalyticCoronagraph(osys, oversample=8, occulter_box=0.15)

import time
t0s = time.time()
psf_sam = sam_osys.calc_psf(display_intermediates=True)
t1s = time.time()

plt.figure(2)
t0f = time.time()
psf_fft = osys.calc_psf(display_intermediates=True)
t1f = time.time()

plt.figure(3)
plt.clf()
plt.subplot(121)
poppy.utils.display_psf(psf_fft, title="FFT")
plt.subplot(122)
poppy.utils.display_psf(psf_sam, title="SAM")

print "Elapsed time, FFT:  %.3s" % (t1f-t0f)
print "Elapsed time, SAM:  %.3s" % (t1s-t0s)
```



On my circa-2010 Mac Pro, the results are dramatic:

```
Elapsed time, FFT:  62.
Elapsed time, SAM:  4.1
```

4.8 Shifting and rotating optics

All OpticalElements support arbitrary shifts and rotations of the optic. Set the `shift_x`, `shift_y` or `rotation` attributes. The shifts are given in meters for pupil plane optics, or arcseconds for image plane optics. Rotations are given in degrees counterclockwise around the optical axis.

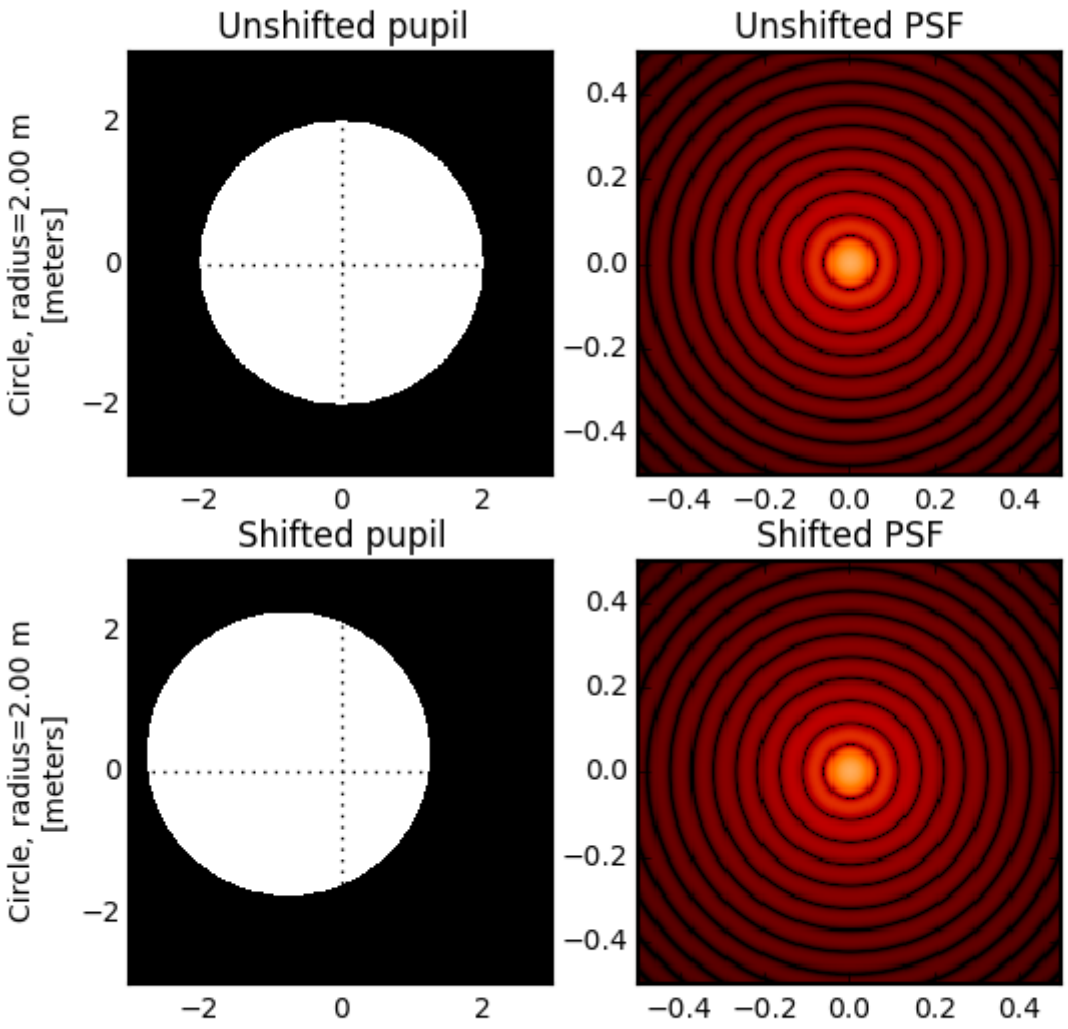
As an example, we can demonstrate the invariance of PSFs when an aperture is shifted:

```
ap_regular = poppy.CircularAperture(radius=2, pad_factor=1.5)  # pad_factor is important here - without␣
↪it you will
ap_shifted = poppy.CircularAperture(radius=2, pad_factor=1.5)  # crop off part of the circle outside␣
↪the array.
```

(continues on next page)

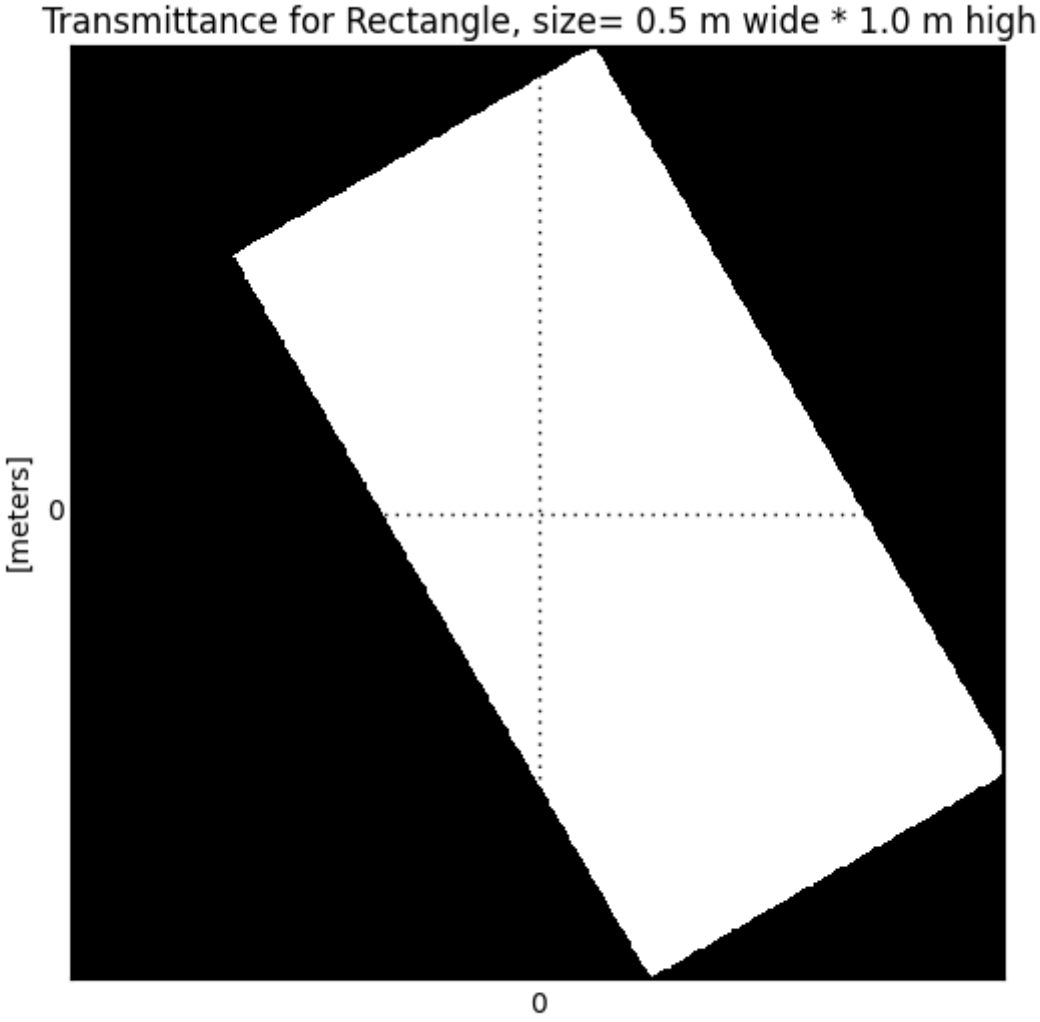
(continued from previous page)

```
ap_shifted.shift_x =-0.75
ap_shifted.shift_y = 0.25

plt.figure(figsize=(6,6))

for optic, title, i in [(ap_regular, 'Unshifted', 1), (ap_shifted, 'Shifted', 3)]:

    sys = poppy.OpticalSystem()
    sys.add_pupil(optic)
    sys.add_detector(0.010, fov_pixels=100)
    psf = sys.calc_psf()

    ax1 = plt.subplot(2,2,i)
    optic.display(nrows=2, colorbar=False, ax=ax1)
    ax1.set_title(title+' pupil')
    ax2 = plt.subplot(2,2,i+1)
    poppy.display_psf(psf,ax=ax2, colorbar=False)
    ax2.set_title(title+' PSF')
```



In addition to setting the attributes as shown in the above example, these options can be set directly in the initialization of such elements:

```
ap = poppy.RectangleAperture(rotation=30, shift_x=0.1)
ap.display(colorbar=False)
```


Transmittance for Rectangle, size= 0.5 m wide * 1.0 m high
[meters]
0
0

CHAPTER 5

Available Optical Element Classes

There are many available predefined types of optical elements in poppy. In addition you can easily specify your own custom optics.

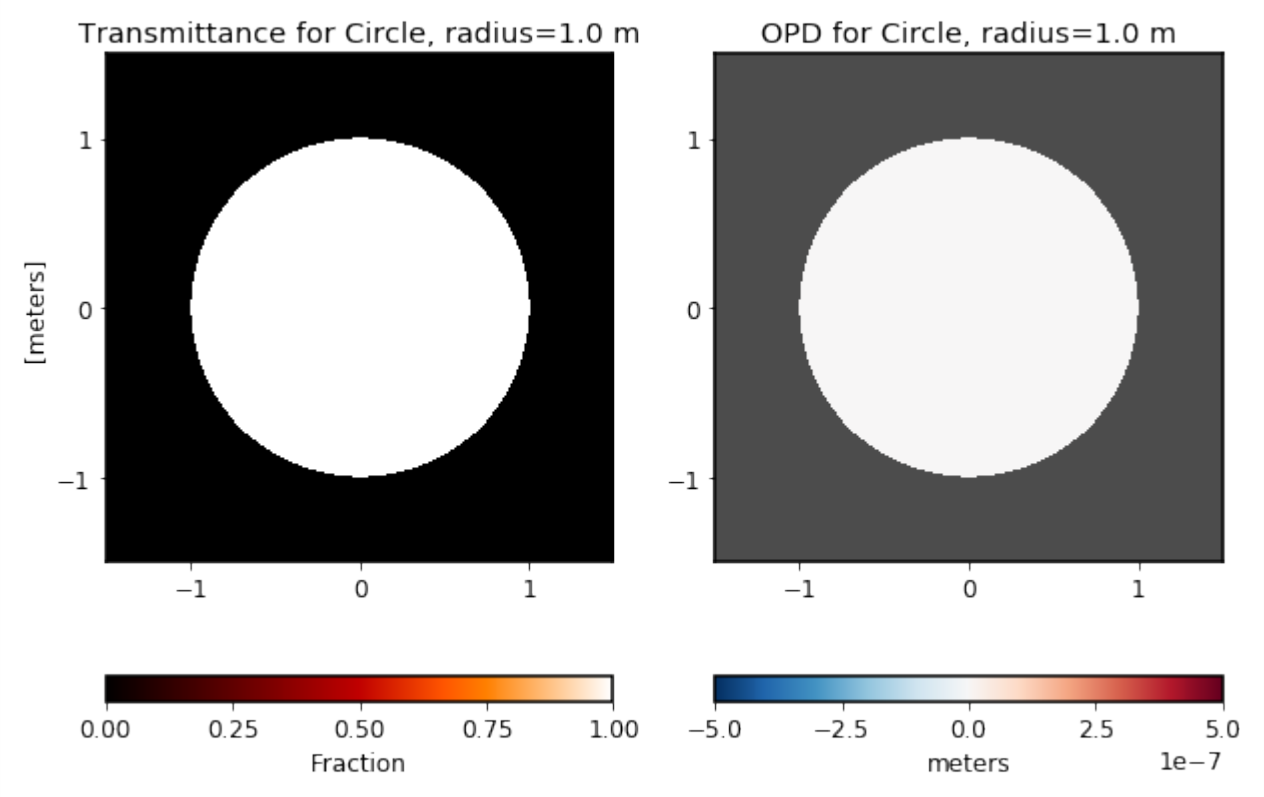
5.1 Pupils and Aperture Stops

These optics have dimensions specified in meters, or other units of length. They can be used in the pupil planes of a Fraunhofer optical system, or any plane of a Fresnel optical system. All of these may be translated or rotated around in the plane by setting the `rotation`, `shift_x` or `shift_y` parameters.

5.1.1 CircularAperture

A basic aperture stop.

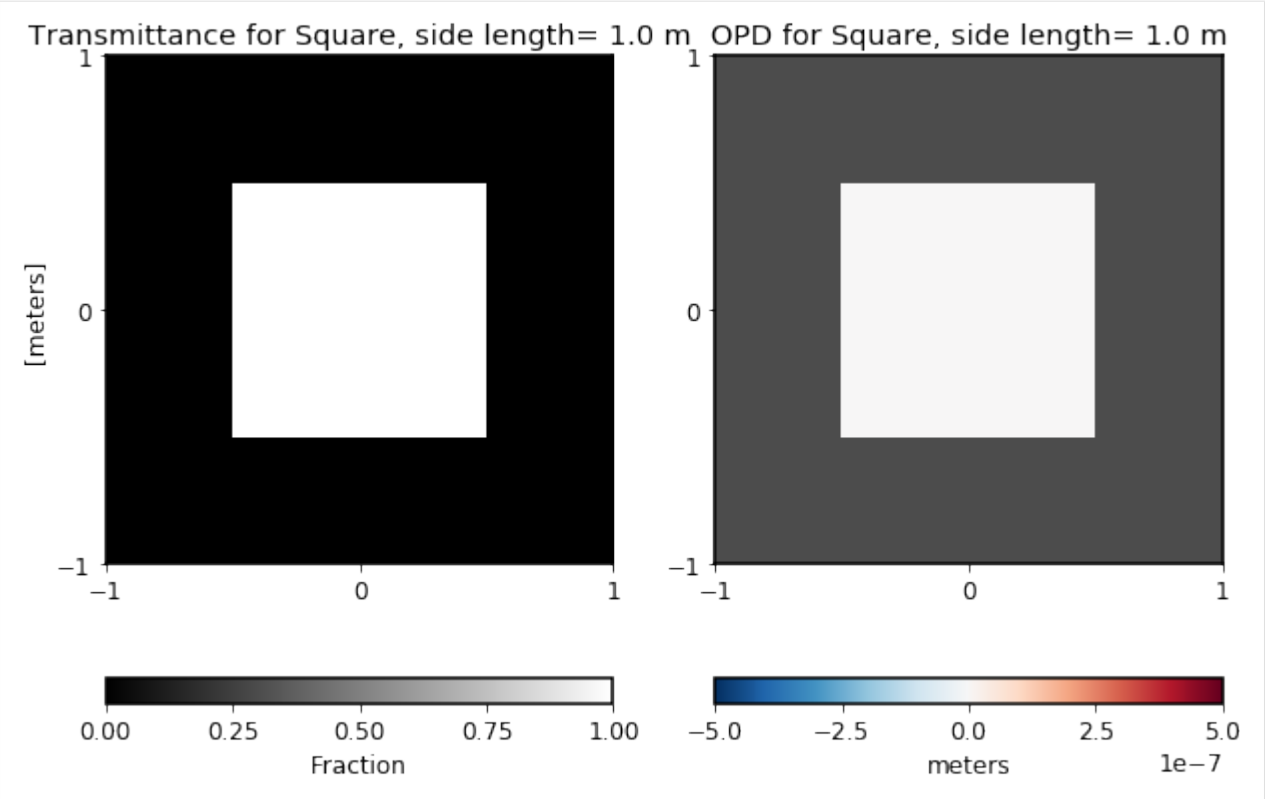
[3]:
```
optic = poppy.CircularAperture(radius=1)
optic.display(what='both');
```



5.1.2 SquareAperture

A square stop. The specified size is the length across any one side.

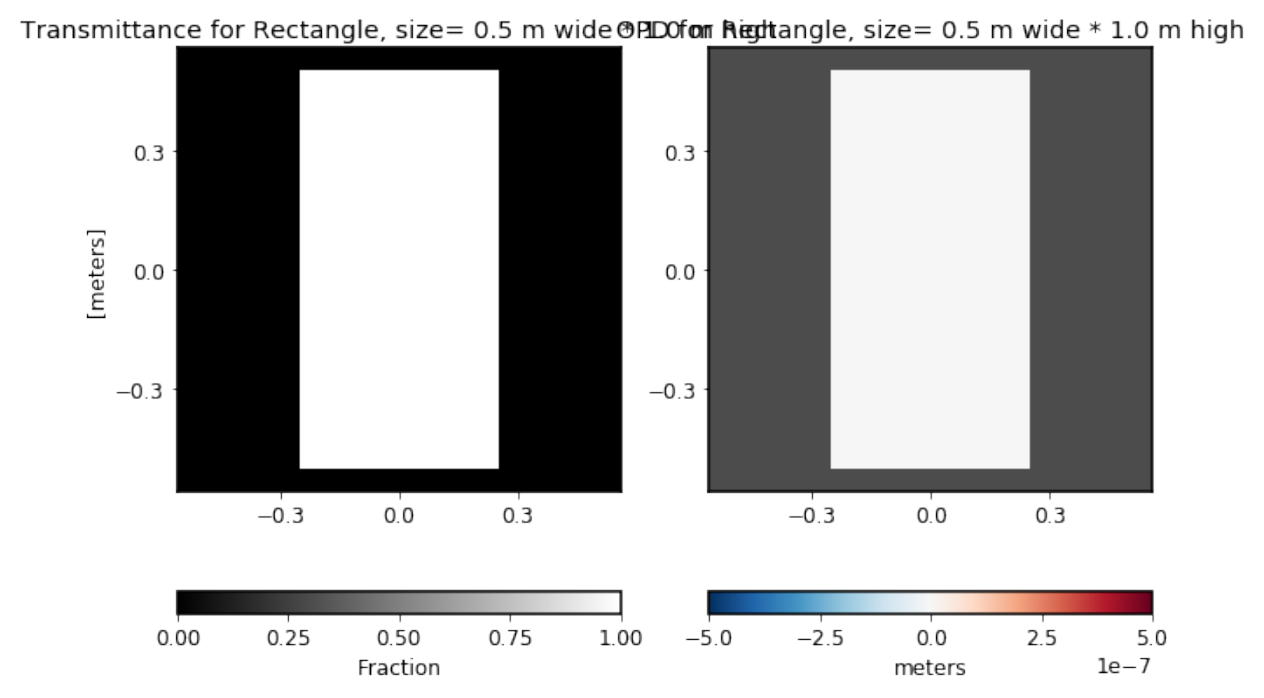
[4]:
```
optic = poppy.SquareAperture(size=1.0)
optic.display(what='both');
```



5.1.3 RectangularAperture

Specify the width and height to define a rectangle.

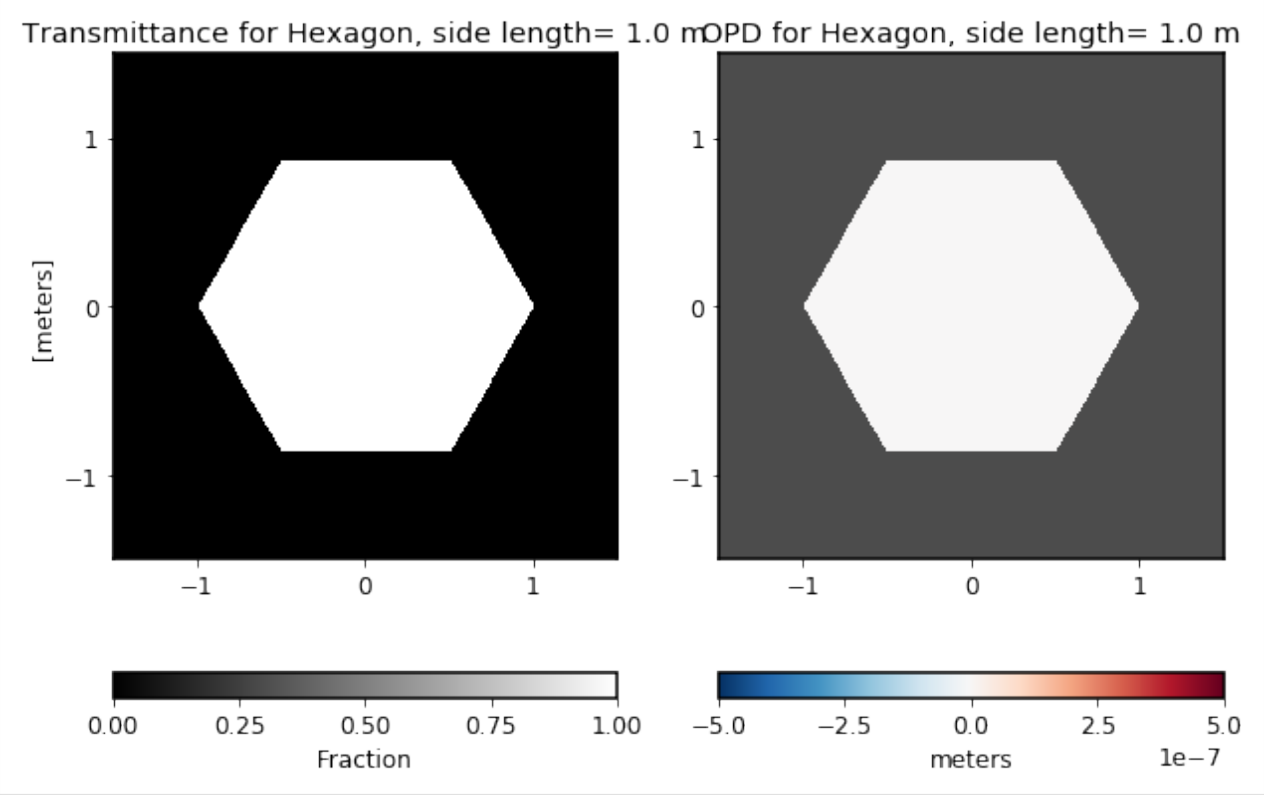
[5]:
```
optic = poppy.RectangleAperture(width=0.5*u.m, height=1.0*u.m)
optic.display(what='both');
```



5.1.4 HexagonAperture

For instance, one segment of a segmented mirror.

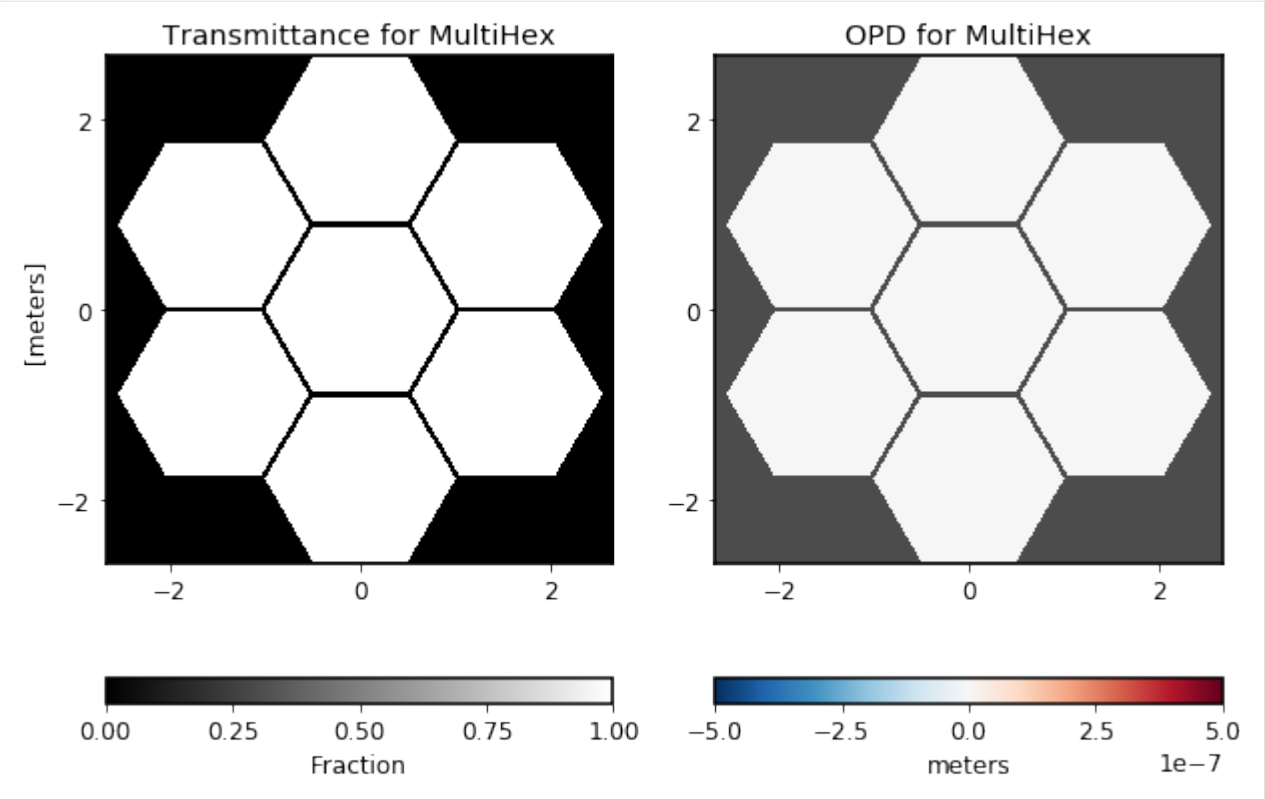
[6]:
```
optic = poppy.HexagonAperture(side=1.0)
optic.display(what='both');
```



5.1.5 MultiHexagonAperture

Arbitrarily many hexagons, in rings. You can adjust the size of each hex, the gap width between them, and whether any hexes are missing (in particular the center one).

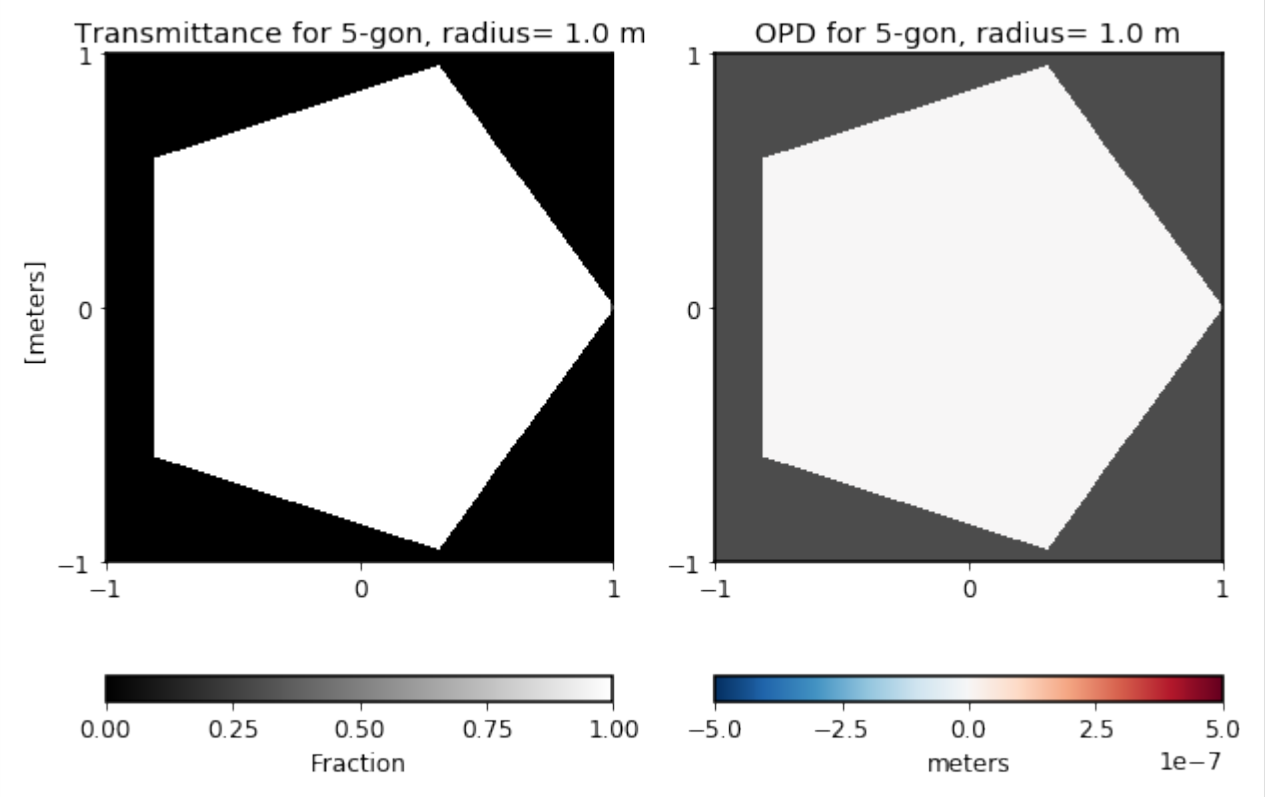
[7]:
```
optic = poppy.MultiHexagonAperture(side=1, rings=1, gap=0.05, center=True)
optic.display(what='both');
```



5.1.6 NgonAperture

Triangular apertures or other regular polygons besides hexagons are uncommon, but a generalized N-gon aperture allows modeling them if needed.

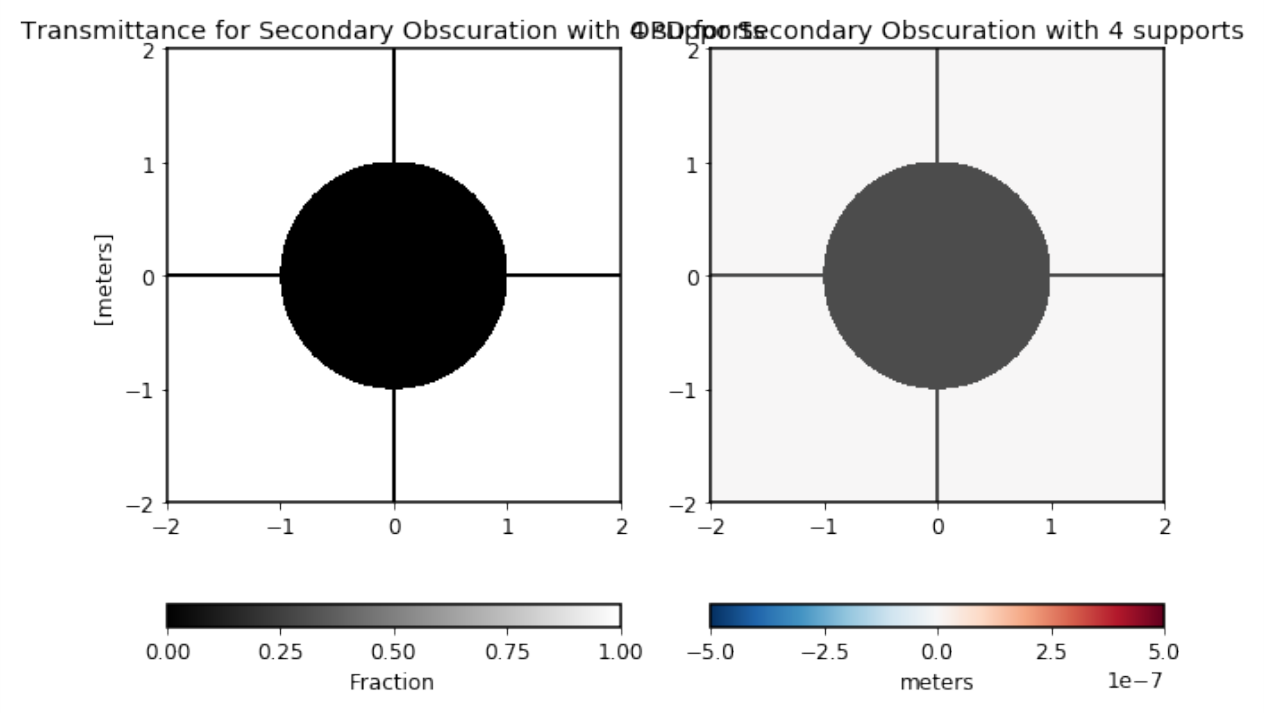
[8]:
```
optic = poppy.NgonAperture(nsides=5)
optic.display(what='both');
```



5.1.7 SecondaryObscuration

This class adds an obstruction which is supported by a regular evenly-spaced grid of identical struts, or set `n_supports=0` for a free-standing obscuration.

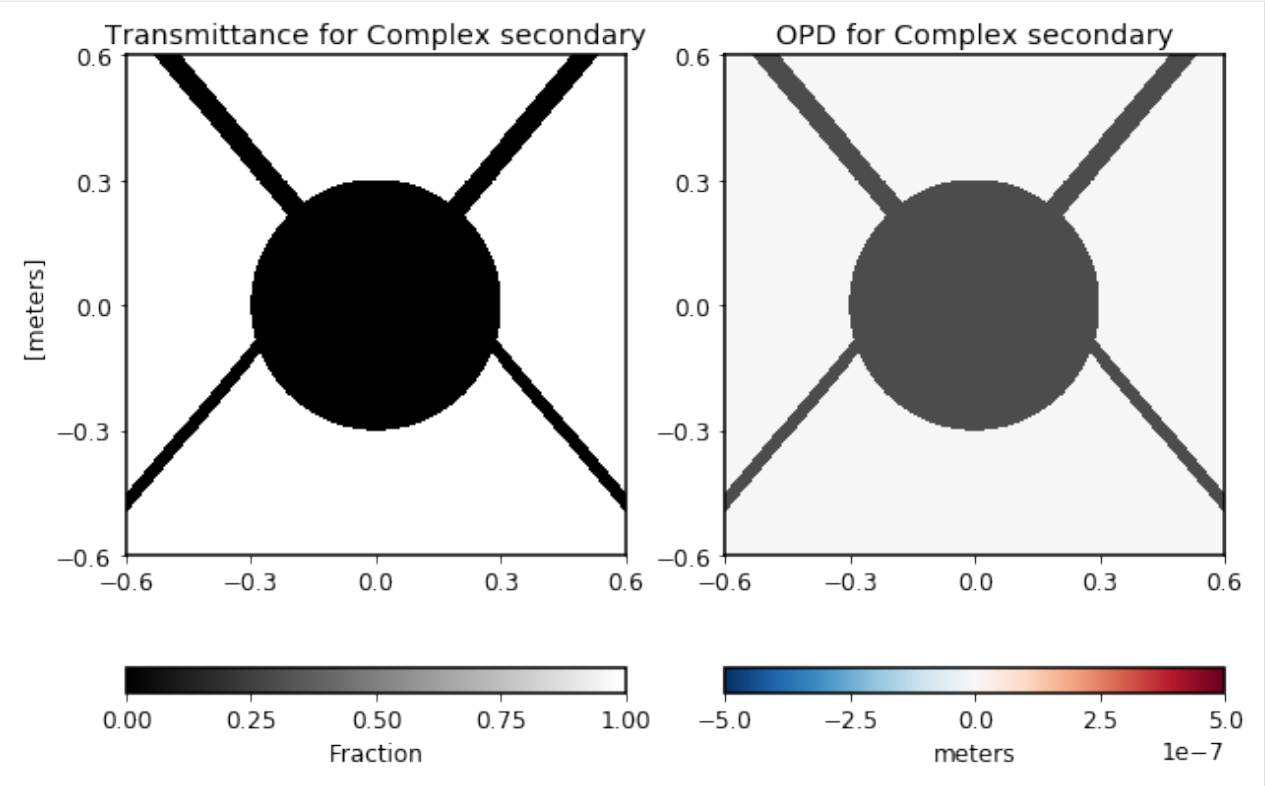
[9]:
```
optic = poppy.SecondaryObscuration(secondary_radius=1.0,
                                   n_supports=4,
                                   support_width=5*u.cm)
optic.display(what='both');
```



5.1.8 AsymmetricSecondaryObscuration

This class allows making more complex "spider" support patterns than the SecondaryObscuration class. Each strut may individually be adjusted in angle, width, and offset in x and y from the center of the aperture. Angles are given in a convention such that the +Y axis is 0 degrees, and increase counterclockwise (based on the typical astronomical convention that north is the origin for position angles, increasing towards east.)

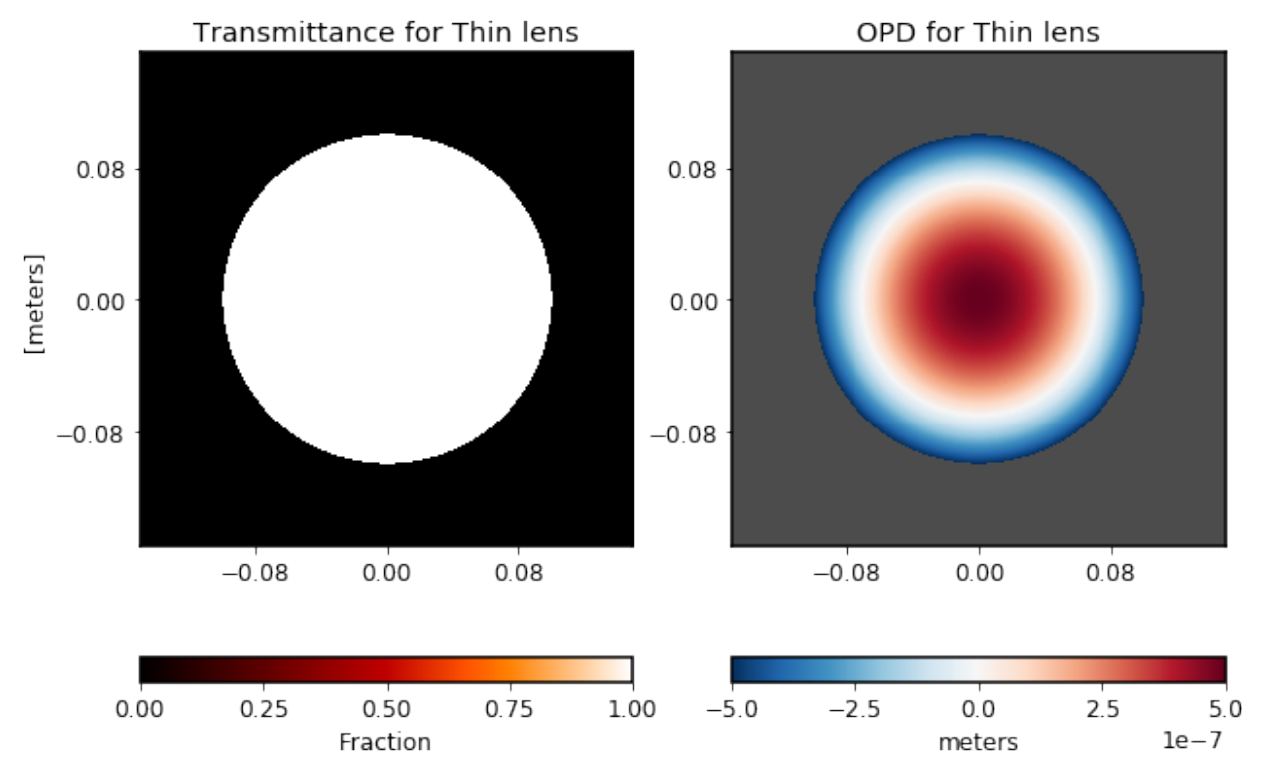
[10]:
```
optic = poppy.AsymmetricSecondaryObscuration(secondary_radius=0.3*u.m,
                                             support_angle=(40, 140, 220, 320),
                                             support_width=[0.05, 0.03, 0.03, 0.05],
                                             support_offset_x=[0, -0.2, 0.2, 0],
                                             name='Complex secondary')
optic.display(what='both');
```



5.1.9 ThinLens

This models a lens or powered mirror in the thin-lens approximation, with the retardance specified in terms of a number of waves at a given wavelength. The lens is perfectly achromatic.

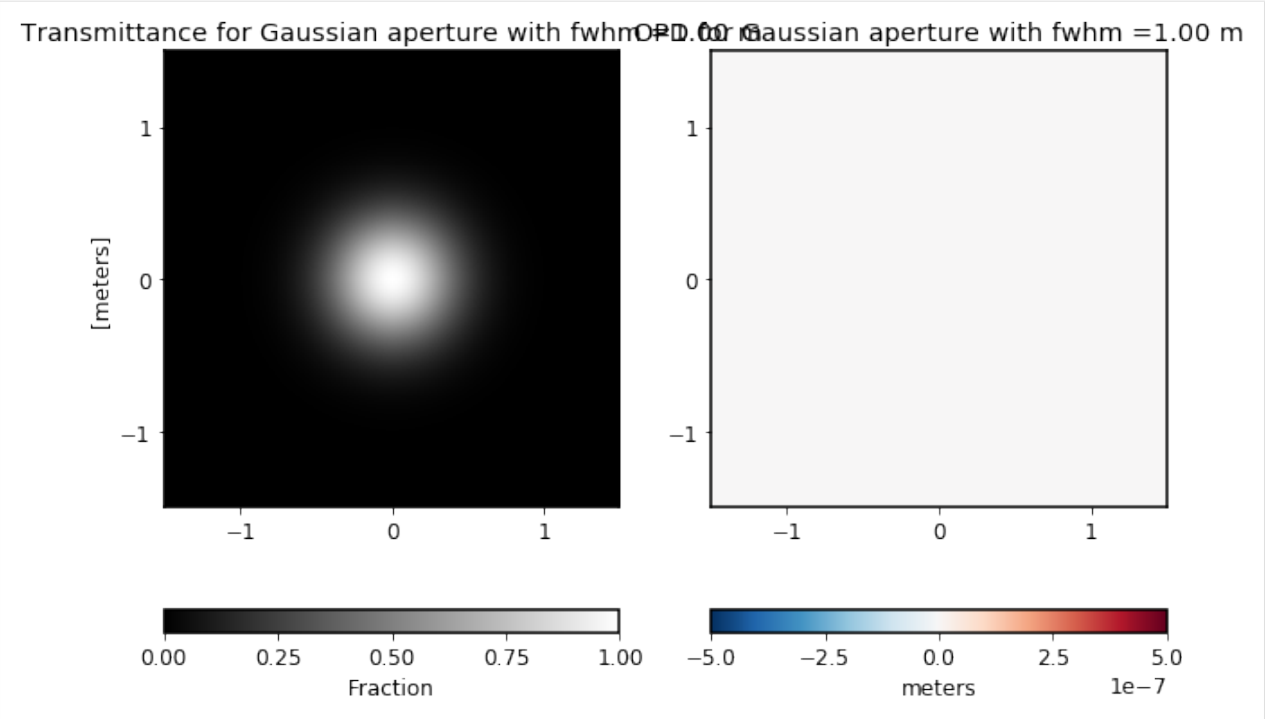
[11]:
```
optic = poppy.ThinLens(nwaves=1, reference_wavelength=1e-6*u.m, radius=10*u.cm)
optic.display(what='both');
```



5.1.10 GaussianAperture

This Gaussian profile can be used for instance to model an apodizer in the pupil plane, or to model a beam launched from a fiber optic.

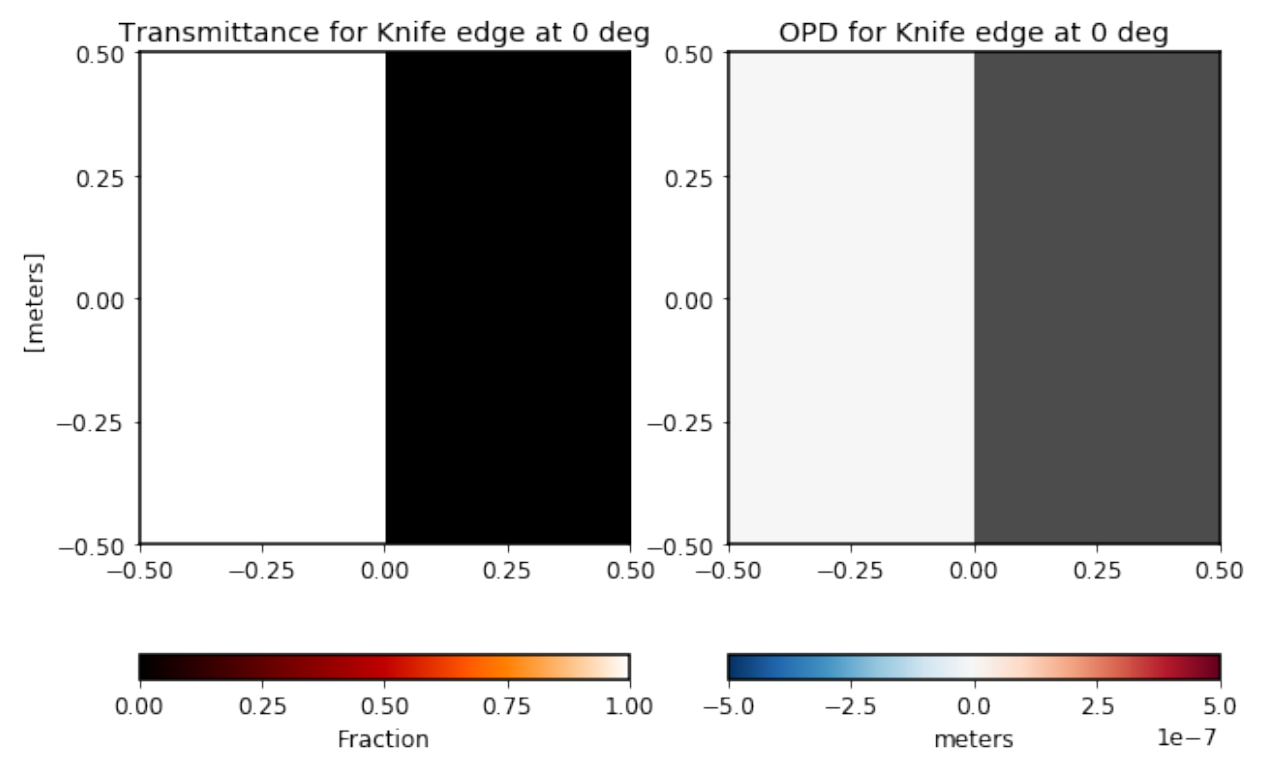
[12]:
```
optic = poppy.GaussianAperture(fwhm=1*u.m)
optic.display(what='both');
```



5.1.11 KnifeEdge

A knife edge is an infinite opaque half-plane.

[13]:
```
optic = poppy.optics.KnifeEdge(rotation=0)
optic.display(what='both');
```

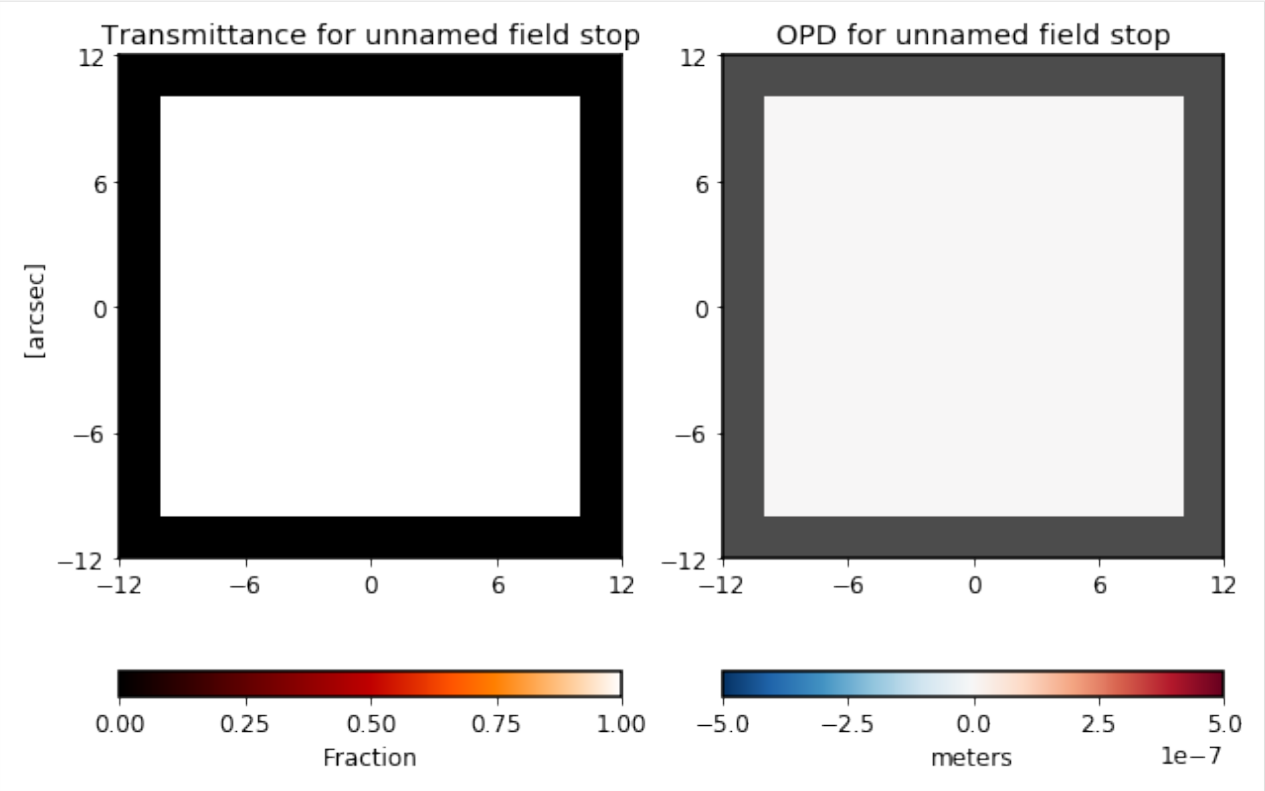


5.2 Image Plane Elements

Image plane classes have dimensions specified in units of arcseconds projected onto the sky. These classes can be placed in image planes in either Fraunhofer or Fresnel optical systems.

5.2.1 SquareFieldStop

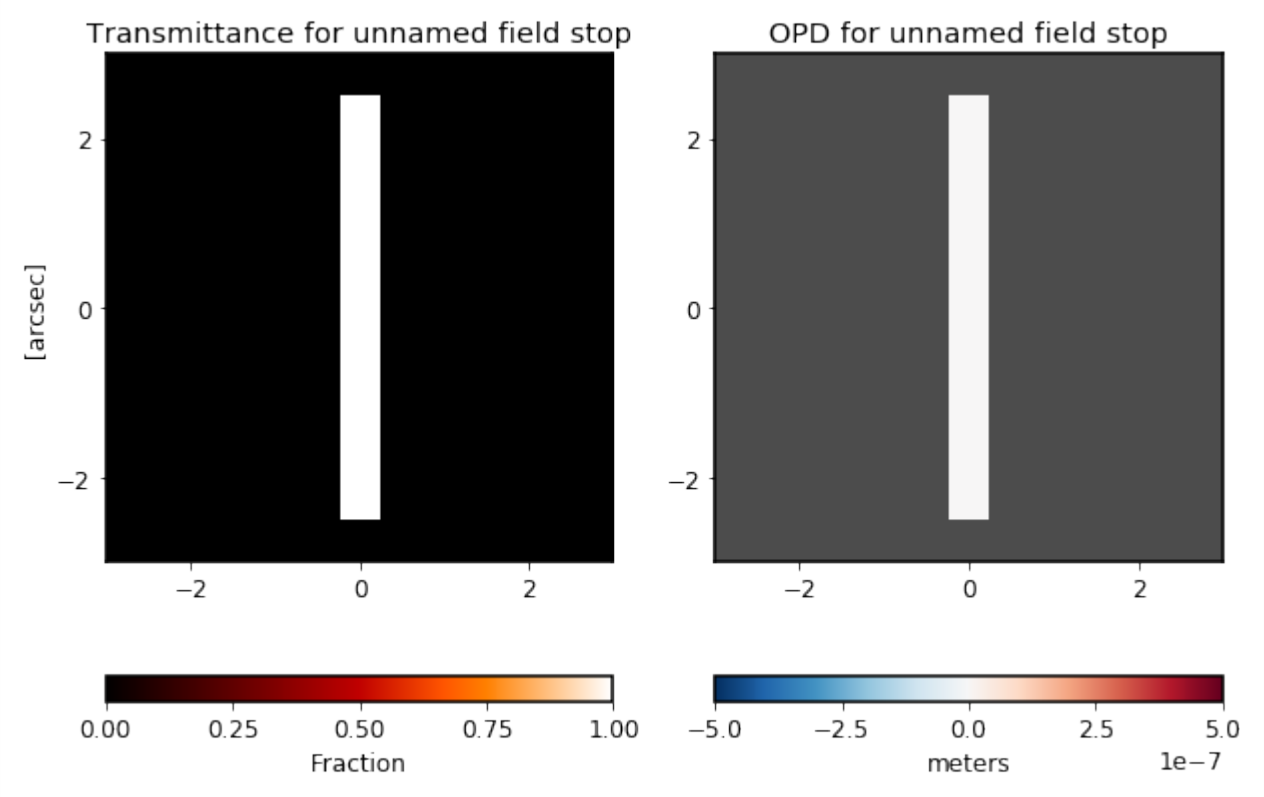
[14]:
```
optic = poppy.SquareFieldStop()
optic.display(what='both');
```



5.2.2 RectangularFieldStop

This class can be used to implement a spectrograph slit, for instance.

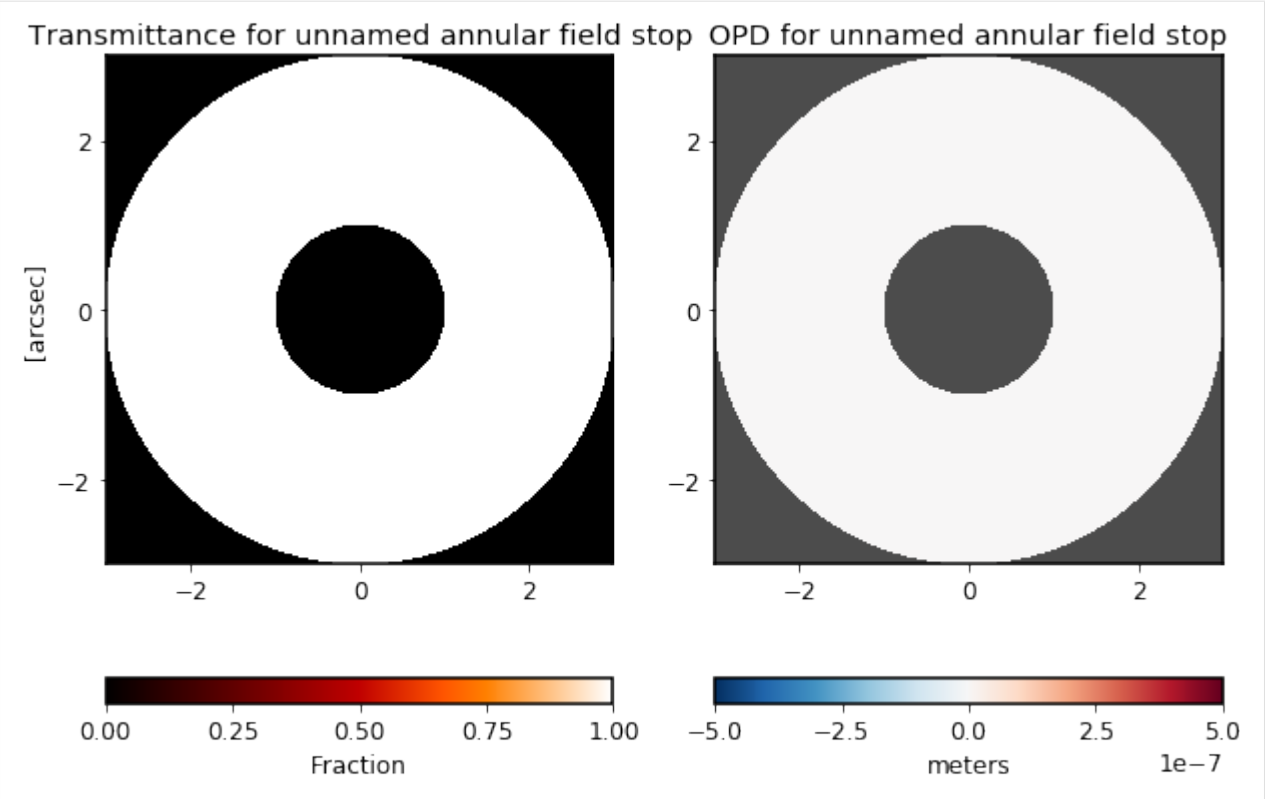
[15]:
```
optic = poppy.RectangularFieldStop()
optic.display(what='both');
```



5.2.3 AnnularFieldStop

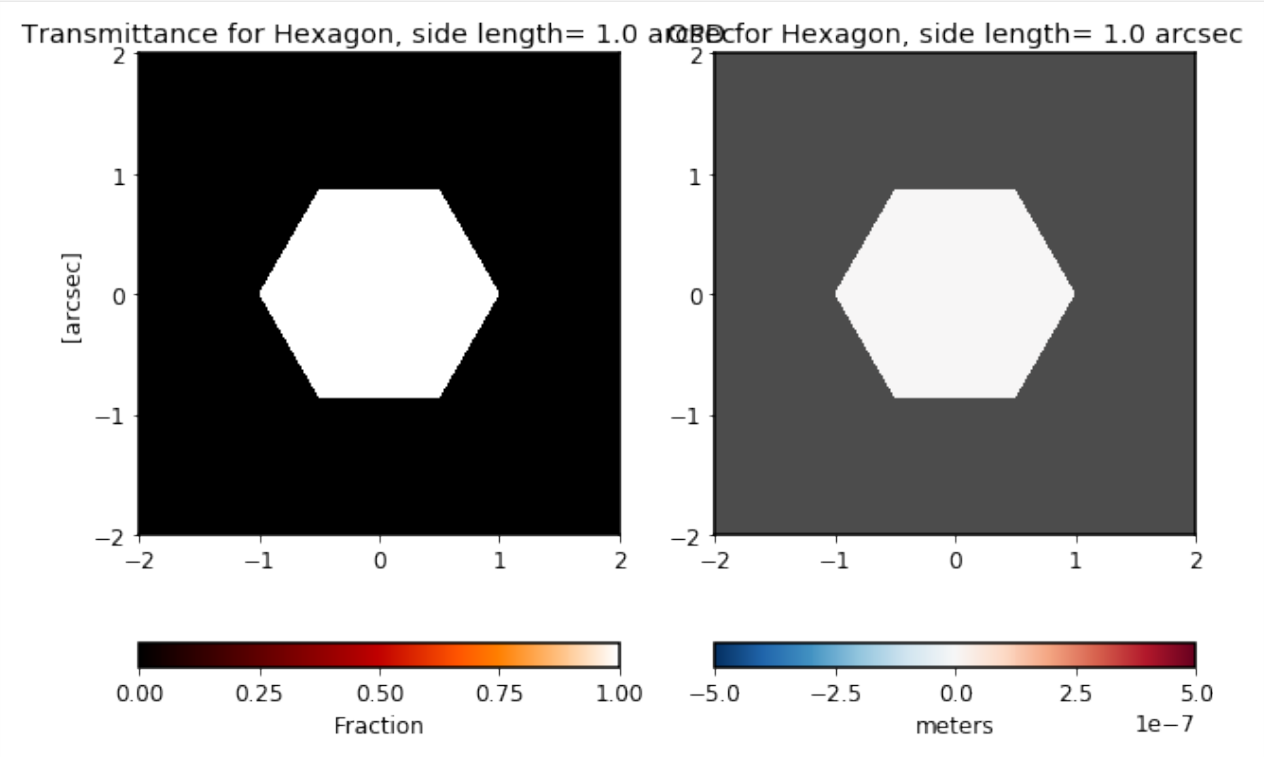
You can also use this as a circular field stop by setting `radius_inner=0`.

[16]:
```
optic = poppy.optics.AnnularFieldStop(radius_inner=1, radius_outer=3)
optic.display(what='both');
```



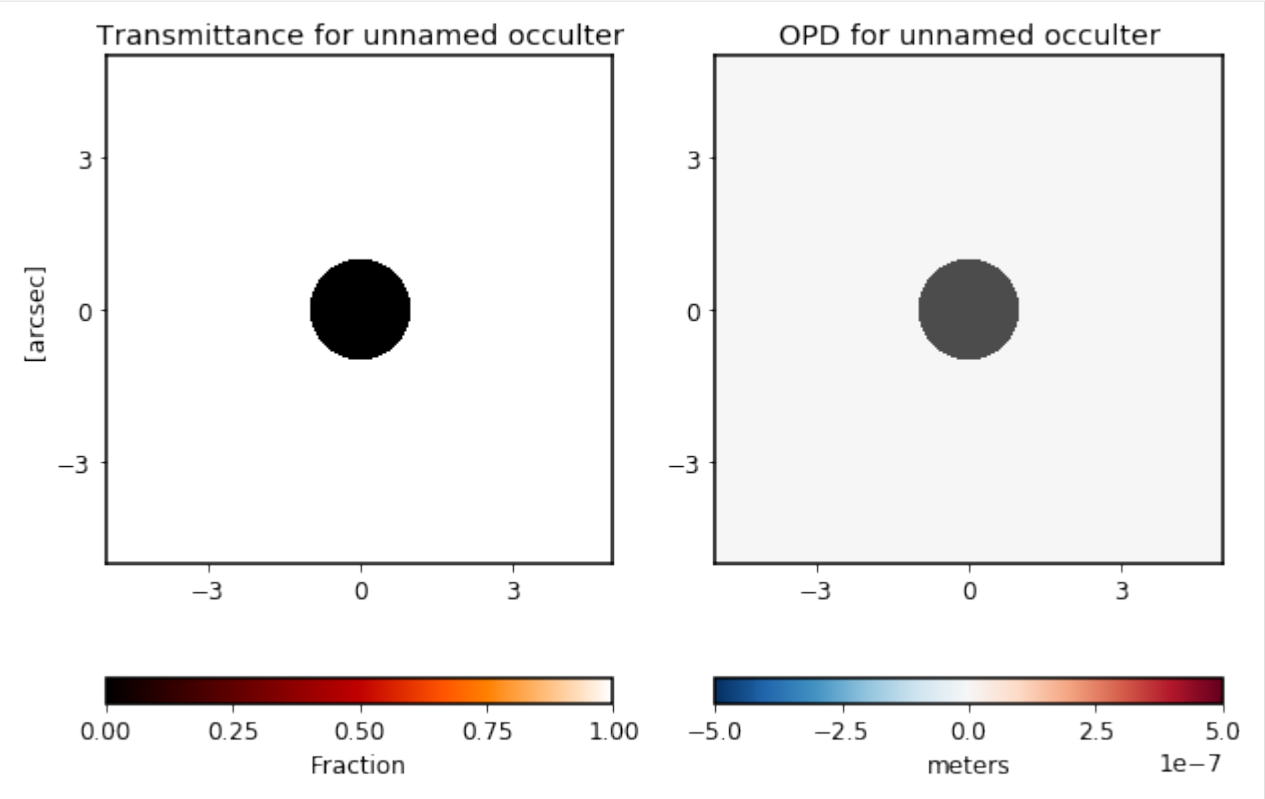
5.2.4 HexagonFieldStop

[17]:
```
optic = poppy.optics.HexagonFieldStop()
optic.display(what='both');
```



5.2.5 CircularOcculter

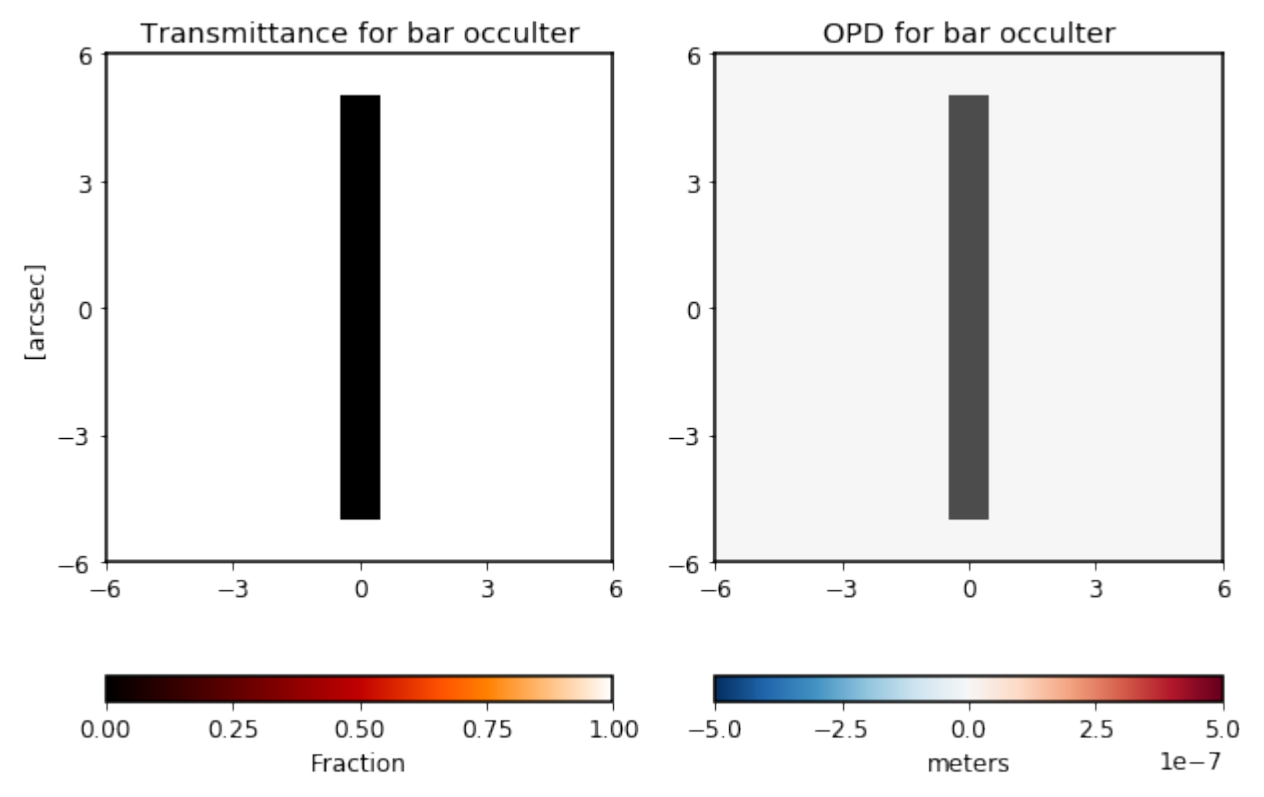
[19]:
```
optic = poppy.optics.CircularOcculter()
optic.display(what='both');
```



5.2.6 BarOcculter

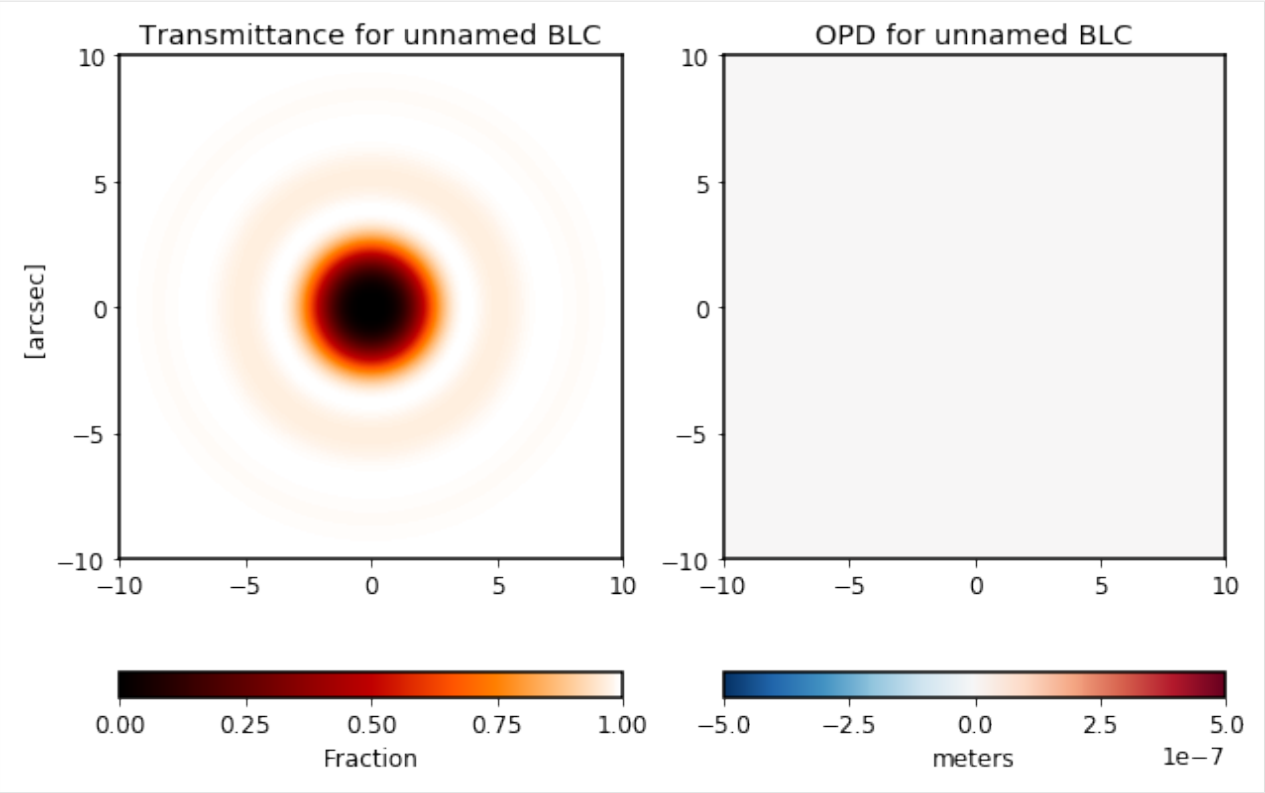
This is an opaque bar or line.

[20]:
```
optic = poppy.optics.BarOcculter(width=1, height=10)
optic.display(what='both');
```



5.2.7 BandLimitedCoronagraph

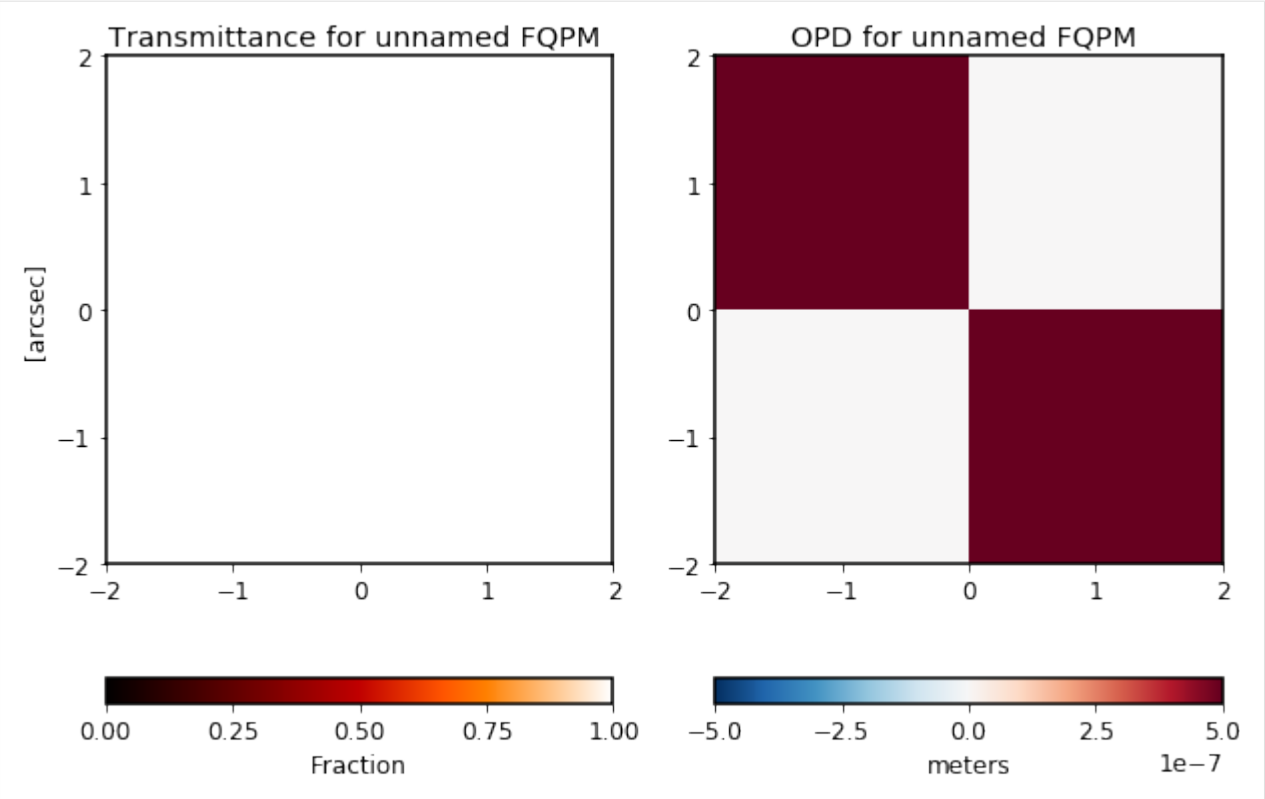
[21]:
```
optic = poppy.BandLimitedCoronagraph()
optic.display(what='both');
```



5.2.8 Four Quadrant Phase Mask

The class is named `IdealFQPM` because this implements a notionally perfect 4QPM at some given wavelength; it has precisely half a wave retardance at the reference wavelength.

[22]:
```
optic = poppy.IdealFQPM(wavelength=1*u.micron)
optic.display(what='both');
```

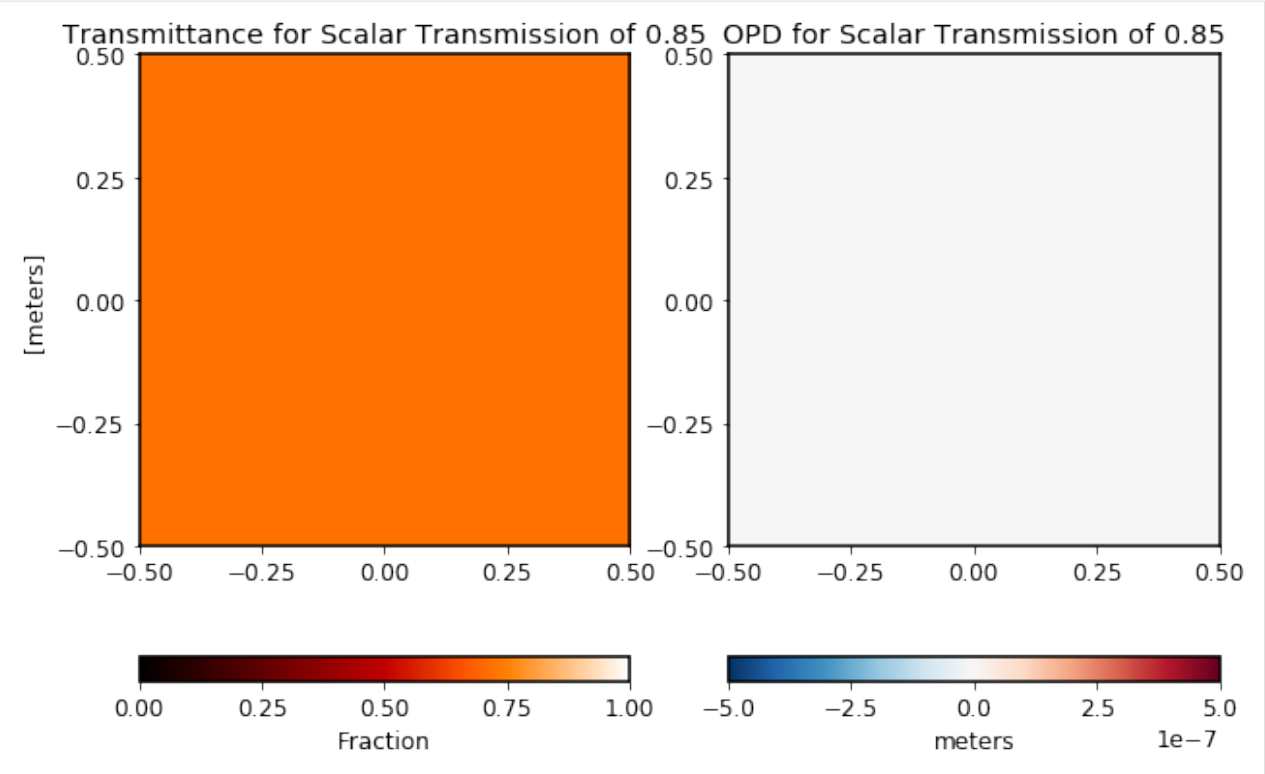


5.3 General Purpose Elements

5.3.1 ScalarTransmission

This class implements a uniformly multiplicative transmission factor, i.e. a neutral density filter.

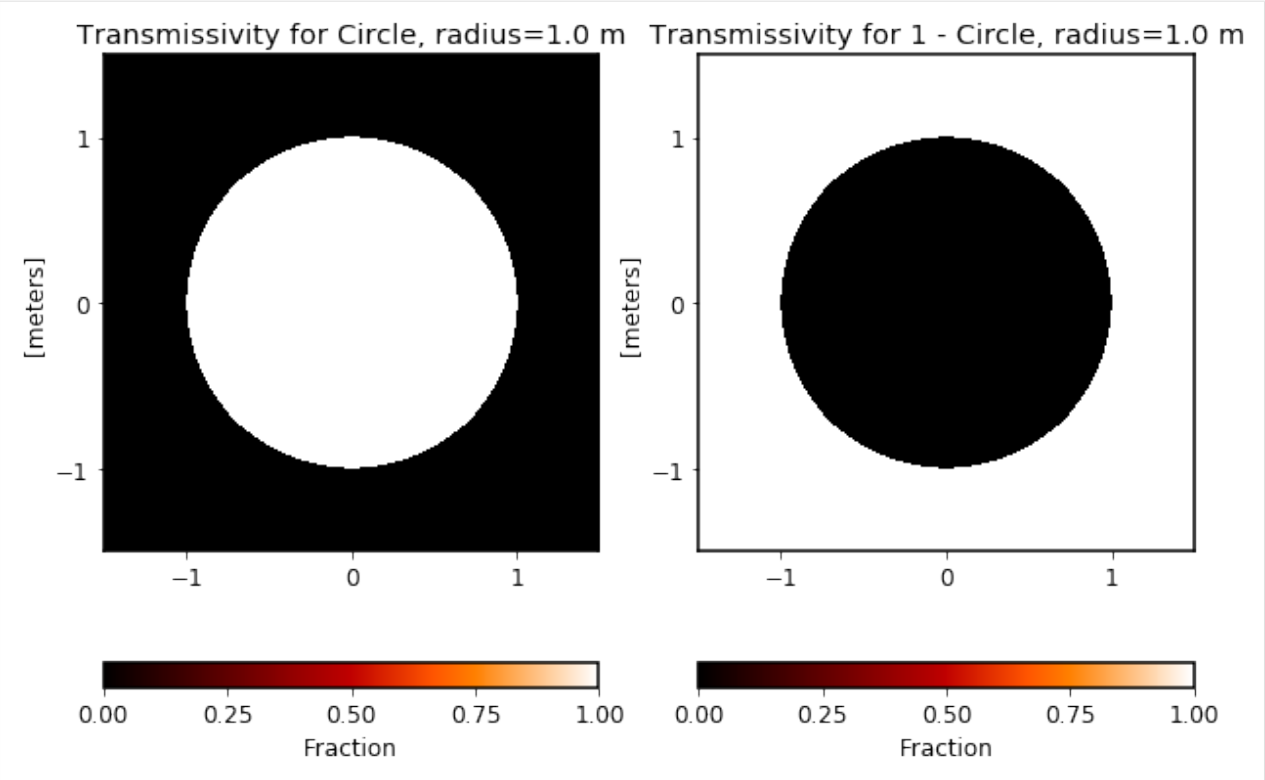
[3]:
```
optic = poppy.ScalarTransmission(transmission=0.85)
optic.display(what='both');
```



5.3.2 Inverse Transmission

This optic acts on another to flip the transmission: Areas which were 0 become 1 and vice versa. This operation is not meant as a representative of some real physical process itself, but can be useful in building certain types of compound optics.

[4]:
```
circ = poppy.CircularAperture(radius=1)
ax1= plt.subplot(121)
circ.display(what='amplitude', ax=ax1)

ax2= plt.subplot(122)
inverted_circ = poppy.InverseTransmission(circ)
inverted_circ.display(grid_size=3, what='amplitude', ax=ax2)
```



5.4 Wavefront Errors

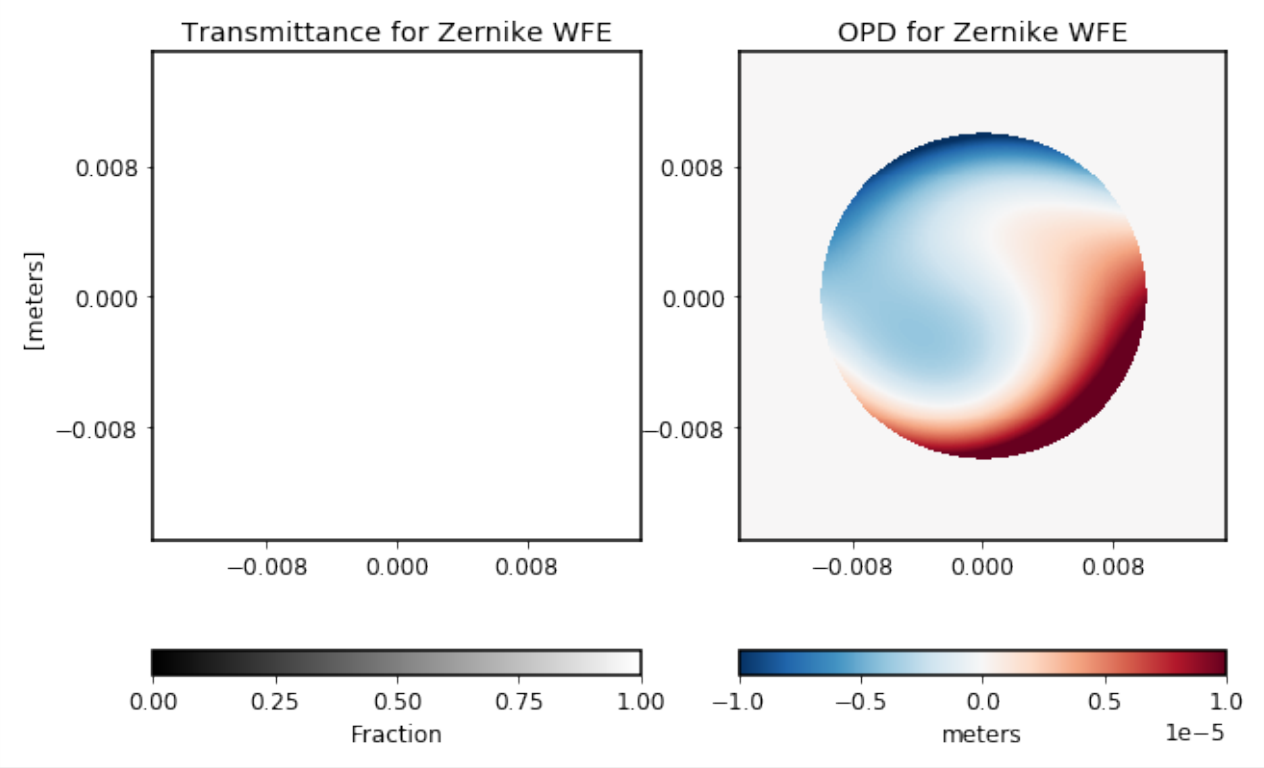
Wavefront error classes can be used to represent various forms of phase delay, typically at a pupil plane. Further documentation on these classes can be found here.

5.4.1 ZernikeWFE

Wavefront errors can be specified by giving a list of Zernike coefficients, which are ordered using the Noll indexing convention. The different Zernikes are then added together to make an overall wavefront map.

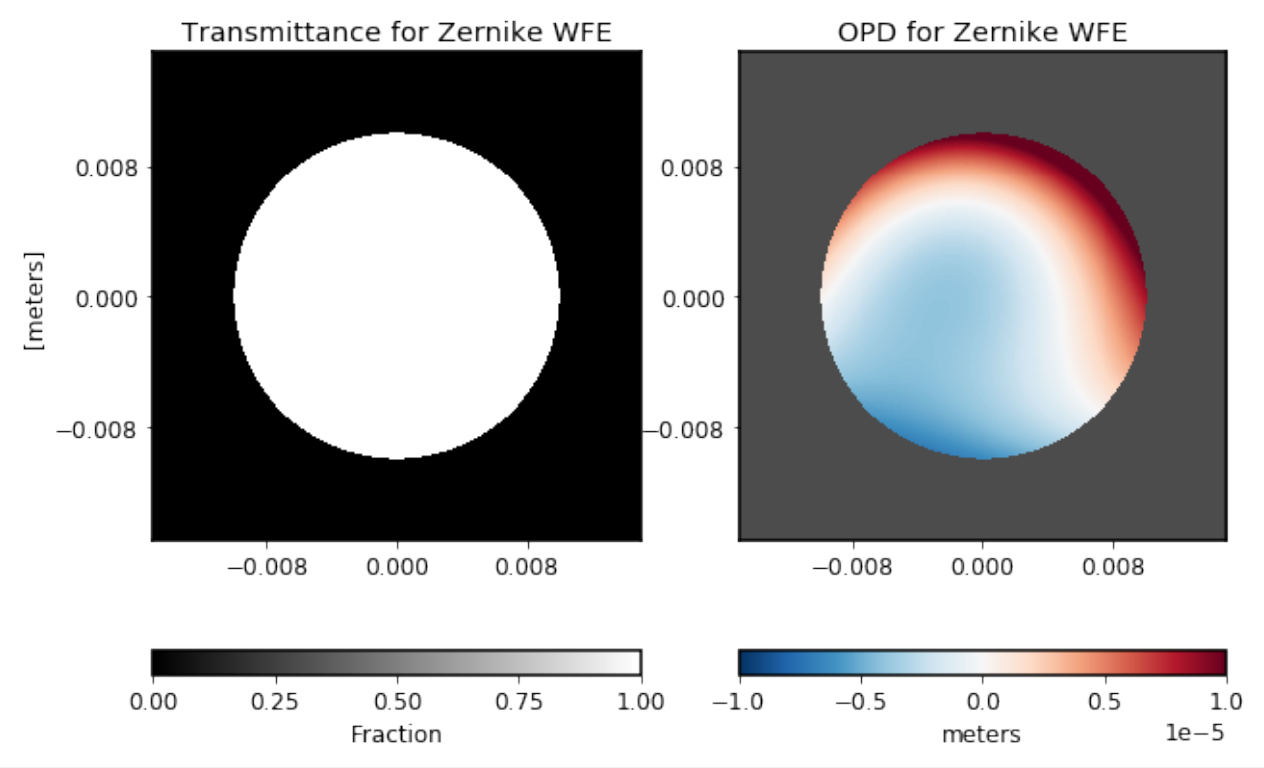
Note that while the Zernikes are defined with respect to some notional circular aperture of a given radius, by default this class just implements the *wavefront error* part, not the aperture stop.

[17]:
```
optic = poppy.ZernikeWFE(radius=1*u.cm,
                         coefficients=[0.1e-6, 3e-6, -3e-6, 1e-6, -7e-7, 0.4e-6, -2e-6],
                         aperture_stop=False)
optic.display(what='both', opd_vmax=1e-5, grid_size=0.03);
```



You can optionally set the parameter `aperture_stop=True` to ZernikeWFE if you want it to also act as a circular aperture stop.

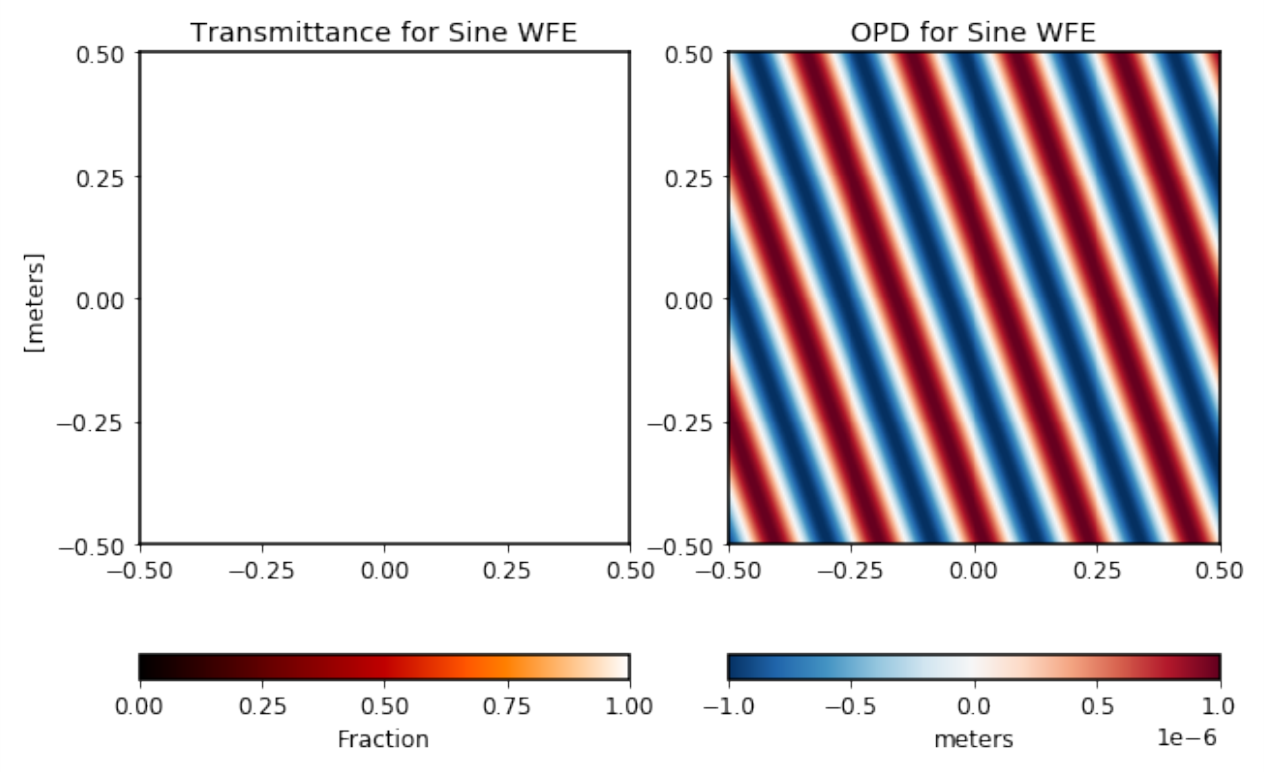
[18]:
```
optic = poppy.ZernikeWFE(radius=1*u.cm,
                         coefficients=[0, 2e-6, 3e-6, 2e-6, 0, 0.4e-6, 1e-6],
                         aperture_stop=True)
optic.display(what='both', opd_vmax=1e-5, grid_size=0.03);
```



5.4.2 SineWaveWFE

A sinusoidal ripple across a mirror, for instance a deformable mirror. Specify the ripple by the spatial frequency (i.e. cycles per meter). Use the `amplitude`, `rotation` and `phaseoffset` parameters to adjust the sine wave.

[16]:
```
optic = poppy.SineWaveWFE(spatialfreq=5/u.meter,
                          amplitude=1*u.micron,
                          rotation=20)
optic.display(what='both', opd_vmax=1*u.micron);
```

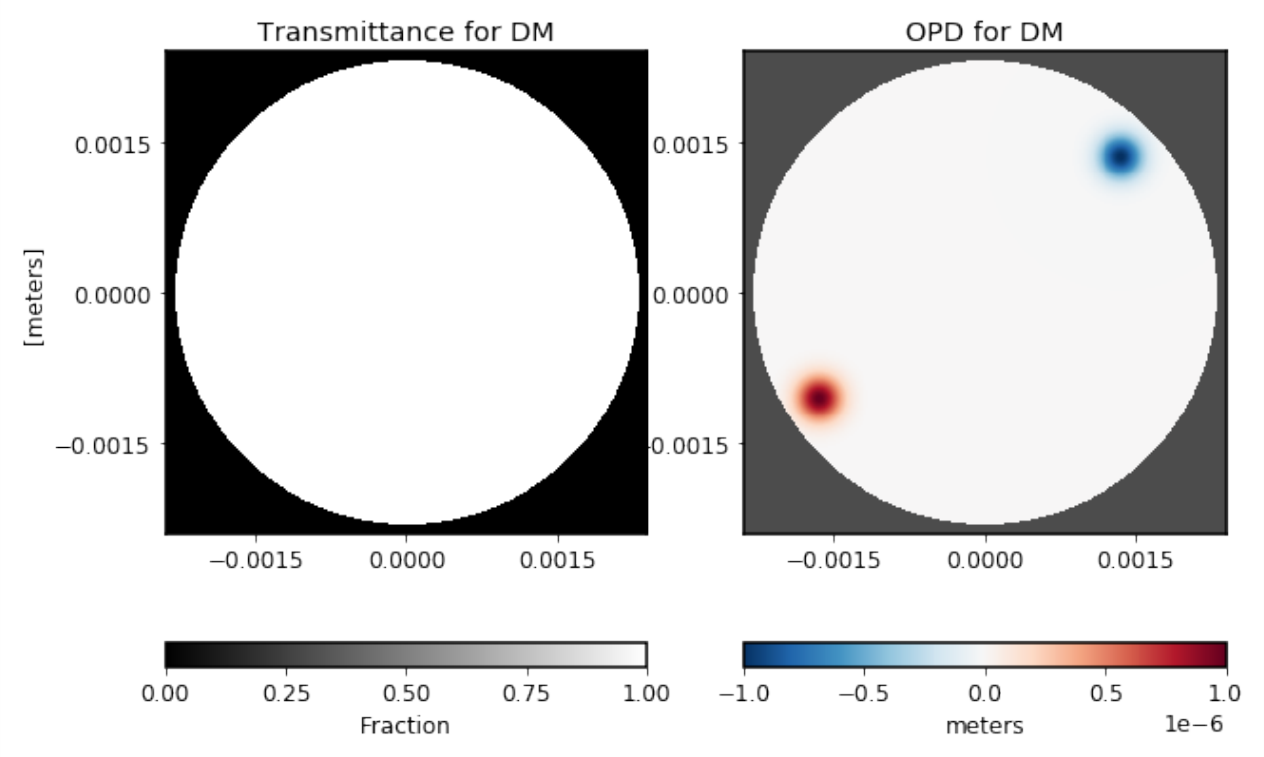


5.5 Deformable Mirrors

5.5.1 Continuous Deformable Mirrors

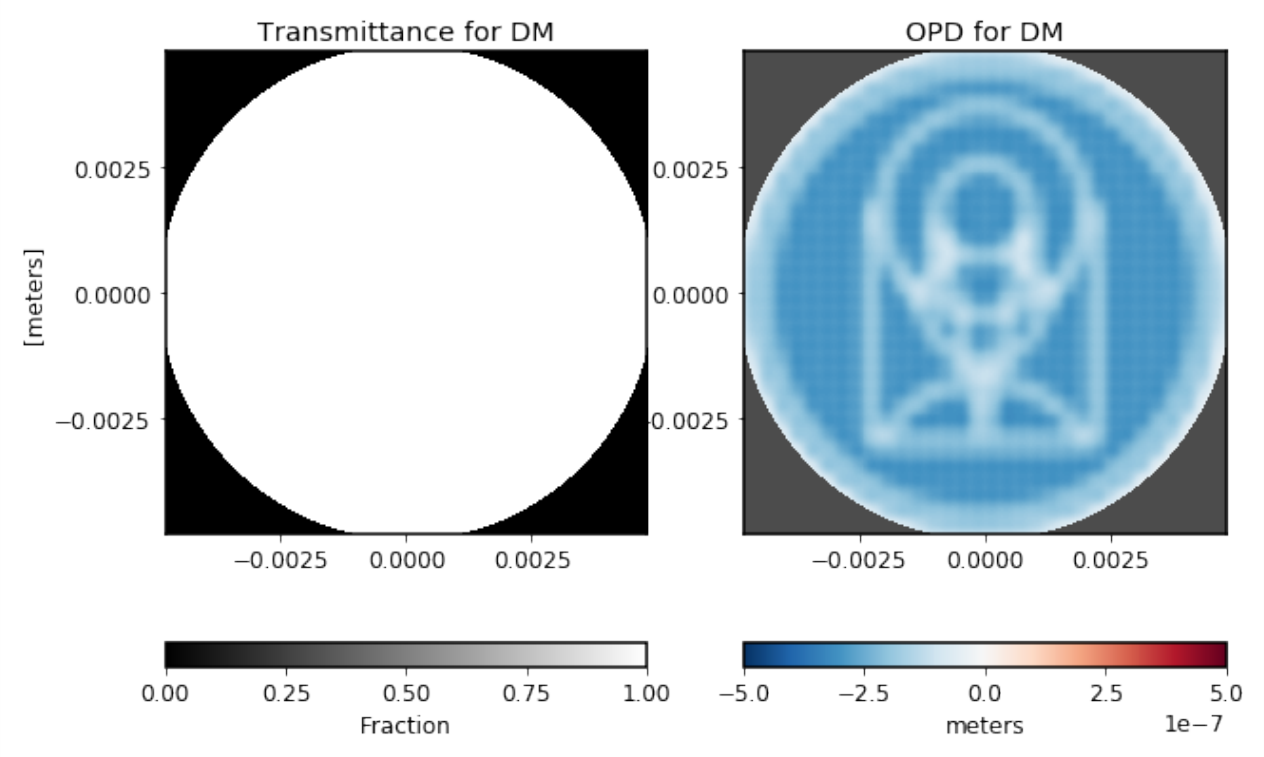
Represents a continuous phase-sheet DM, such as from Boston Micromachines or AOA Xinetics. Individual actuators in the DM can be addressed and poked with the `set_actuator` function. That function takes 3 parameters: `set_actuator(x_act, y_act, piston)`.

[3]:
```
dm = poppy.dms.ContinuousDeformableMirror(dm_shape=(16,16), actuator_spacing=0.3*u.mm, radius=2.3*u.mm)
dm.set_actuator(2, 4, 1e-6)
dm.set_actuator(12, 12, -1e-6)
dm.display(what='both', opd_vmax=1*u.micron);
```



You can also set all actuators in the entire DM surface at once by calling `set_surface` with a suitably-sized array.

[4]:
```
dm = poppy.dms.ContinuousDeformableMirror(dm_shape=(32,32), actuator_spacing=0.3*u.mm, radius=4.9*u.mm)
target_surf = fits.getdata('dm_example_surface.fits')
dm.set_surface(target_surf);
dm.display(what='both');
```



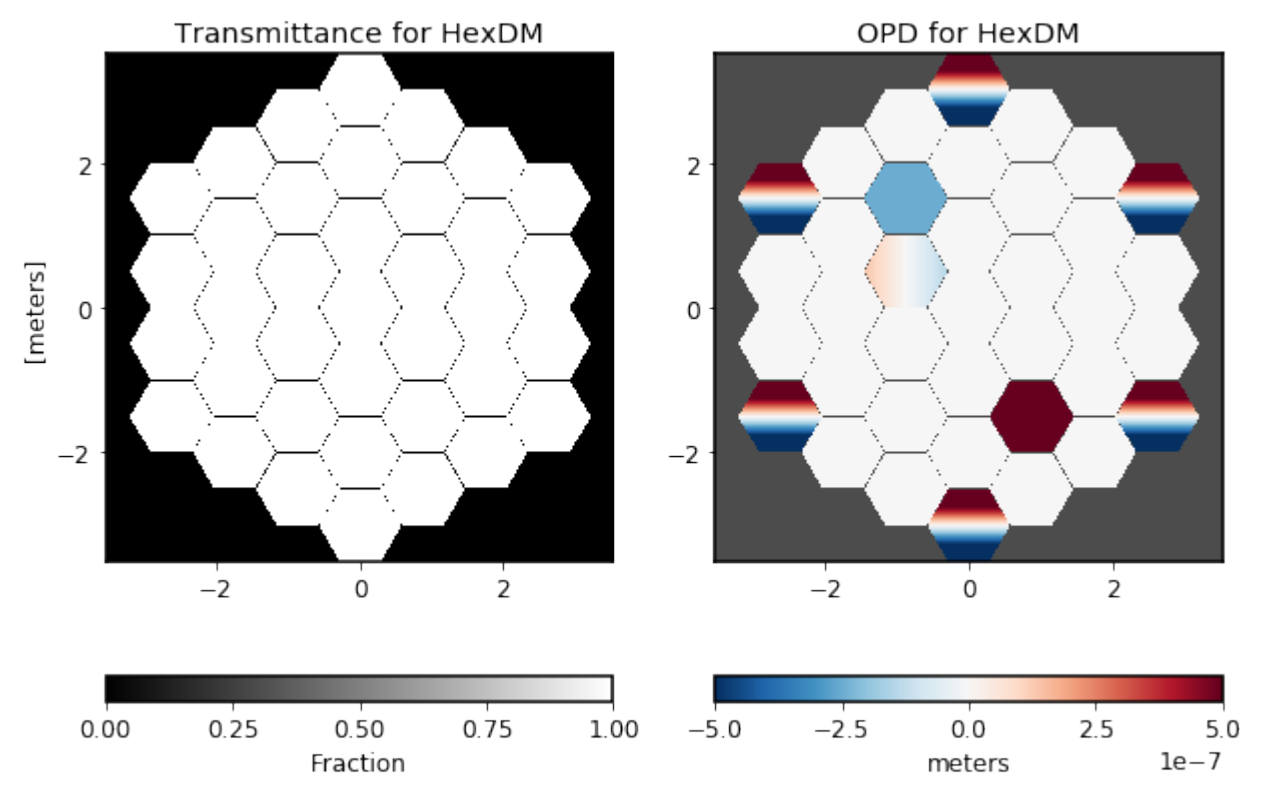
The DM class has much more functionality than currently shown here, including loading measured influence functions and models of high-spatial-frequency actuator print-through. These features are not yet fully documented but please contact Marshall if you're interested.

5.6 Hexagonally Segmented Deformable Mirrors

For instance, one of the devices made by Iris AO.

In this case, the `set_actuator` function takes 4 parameters: `set_actuator(segment_number, piston, tip, tilt)`.

[5]:
```
hexdm = poppy.dms.HexSegmentedDeformableMirror(rings=3)
hexdm.set_actuator(12, 0.5*u.micron, 0, 0)
hexdm.set_actuator(18, -0.25*u.micron, 0, 0)
hexdm.set_actuator(6, 0, -0.25*u.microradian, 0)
for i in range (19, 37, 3): hexdm.set_actuator(i, 0, 0, 2*u.microradian)
hexdm.display(what='both');
```


Transmittance for HexDM
OPD for HexDM
[meters]
2
0
−2
−2
0
2
0.00
0.25
0.50
0.75
1.00
Fraction
−5.0
−2.5
0.0
2.5
5.0
meters
1e−7

[ ]:

CHAPTER 6

Representing sources of wavefront error

POPPY allows you to introduce wavefront error at any plane in an optical system through use of a wavefront error optical element. For Fraunhofer-domain propagation, the optical element types discussed here will act as pupil-conjugate planes. Currently, there is the `ZernikeWFE` optical element to represent wavefront error as a combination of Zernike terms, and the `ParameterizedWFE` optical element that offers the ability to specify basis sets other than Zernikes. Several such *basis functions* are provided, along with tools for composing and decomposing wavefront errors using these bases. There is also a `SineWaveWFE` element that lets you represent a simple sinusoidal phase ripple.

6.1 ZernikeWFE

The `ZernikeWFE` analytic optical element provides a way to introduce wavefront error as a combination of Zernike terms. The Zernike terms are both ordered in and normalized in the convention of Noll 1976.[1] This means that the polynomial terms carry a normalization factor to make $\int_0^{2\pi} \int_0^1 Z_j^2 \, \rho \, d\rho \, d\theta = \pi$. Additionally, the standard deviation of the optical path difference over the disk for any given term will be 1.0 meters.

Now, 1.0 meters is a lot of OPD, so coefficients supplied to ZernikeWFE for realistic situations will typically be on the order of the wavelength of light being passed through the system. For example, for 100 nanometers RMS wavefront error in a particular term, the coefficient would be 100e-9 *meters*. (For information on representing certain values of peak-to-valley wavefront error, see *Normalizing to desired peak-to-valley WFE*.)

To construct a ZernikeWFE optical element, the following keyword arguments are used:

- `coefficients` – A sequence of coefficients in the 1-D ordering of Noll 1976. Note that the first element in Noll's indexing convention is $j = 1$, but Python indices are numbered from 0, so `coefficients[0]` corresponds to the coefficient for $j = 1$.
- `radius` – The radius in meters (in the pupil plane) which the Zernike unit disk covers. Transmission beyond this radius is cut off, just as with a circular aperture of the same radius, so choose one large enough to enclose the entire pupil area.

Here's an example that creates a ZernikeWFE optic to apply zero piston, 30 nm RMS of "tip", and 200 nm RMS of "tilt" to an optical system with a 2 meter diameter pupil.

[1] Noll, R. J. "Zernike polynomials and atmospheric turbulence." JOSA, 1976. doi:10.1364/JOSA.66.000207

```
wfe = poppy.ZernikeWFE(radius=1.0, coefficients=[0, 3e-8, 2e-7])
```

The `wfe` optic can then be added to the OpticalSystem as usual. Next we'll look at some examples of using ZernikeWFE with their output.

The examples that follow make use of the following common constants:

```
RADIUS = 1.0 # meters
WAVELENGTH = 460e-9 # meters
PIXSCALE = 0.01 # arcsec / pix
FOV = 1 # arcsec
NWAVES = 1.0
```

6.1.1 Using ZernikeWFE to introduce astigmatism in a PSF

Per the 1-D indexing convention in Noll 1976, oblique astigmatism is $j = 5$ and vertical astigmatism is $j = 6$. For a contribution of 35 nm RMS from oblique astigmatism, the coefficients sequence looks like `[0, 0, 0, 0, 35e-9]`.

```
coefficients_sequence = [0, 0, 0, 0, 35e-9]

osys = poppy.OpticalSystem()
circular_aperture = poppy.CircularAperture(radius=RADIUS)
osys.add_pupil(circular_aperture)
thinlens = poppy.ZernikeWFE(radius=RADIUS, coefficients=coefficients_sequence)
osys.add_pupil(thinlens)
osys.add_detector(pixelscale=PIXSCALE, fov_arcsec=FOV)

psf_with_zernikewfe = osys.calc_psf(wavelength=WAVELENGTH, display_intermediates=True)
```

The resulting PSF and OPD map:

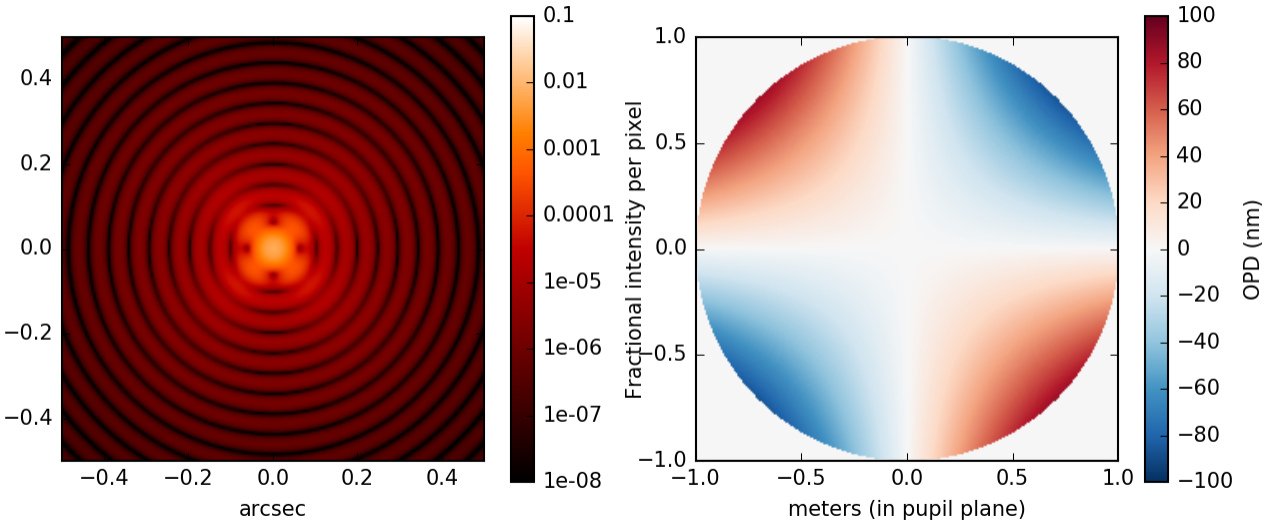


Fig. 1: PSF of a 2.0 meter diameter circular aperture with oblique astigmatism applied.

6.1.2 Using ZernikeWFE to analyze a variety of possible PSFs given a WFE budget

The API for ZernikeWFE also lends itself well to generating coefficients programmatically and passing it in. Say we have an error budget where we know the following about the RMS wavefront error in the Zernike components:

- **Piston**, *j=1* — disregarded for a telescope
- **Tilt X**, *j=2* — ±100 nm RMS
- **Tilt Y**, *j=3* — ±100 nm RMS
- **Focus**, *j=4* — ±50 nm RMS
- **Astigmatism 45**, *j=5* — ±36 nm RMS
- **Astigmatism 0**, *j=6* — ±36 nm RMS

We can use ZernikeWFE to generate a library of sample PSFs satisfying this error budget. First, we write a short function that can generate coefficients from our specifications.

```
wfe_budget = [0, 100, 100, 50, 36, 36]

def generate_coefficients(wfe_budget):
    coefficients = []
    for term in wfe_budget:
        coefficients.append(
            # convert nm to meters, get value in range
            np.random.uniform(low=-1e-9 * term, high=1e-9 * term)
        )
    return coefficients
```

Then we use this to generate a few sets of coefficients.

```
possible_coefficients = [generate_coefficients(wfe_budget) for i in range(5)]
```

Now we simply loop over the sets of coefficients, supplying them to ZernikeWFE:

```
plt.figure(figsize=(18,2))

results = []

for coefficient_set in possible_coefficients:
    osys = poppy.OpticalSystem()
    circular_aperture = poppy.CircularAperture(radius=RADIUS)
    osys.add_pupil(circular_aperture)
    zwfe = poppy.ZernikeWFE(
        coefficients=coefficient_set,
        radius=RADIUS
    )
    osys.add_pupil(zwfe)
    osys.add_detector(pixelscale=PIXSCALE, fov_arcsec=FOV)

    psf = osys.calc_psf(wavelength=WAVELENGTH, display=False)
    results.append(psf)
```

Here's a figure showing the various PSFs with their corresponding OPD maps pulled out for illustration purposes.

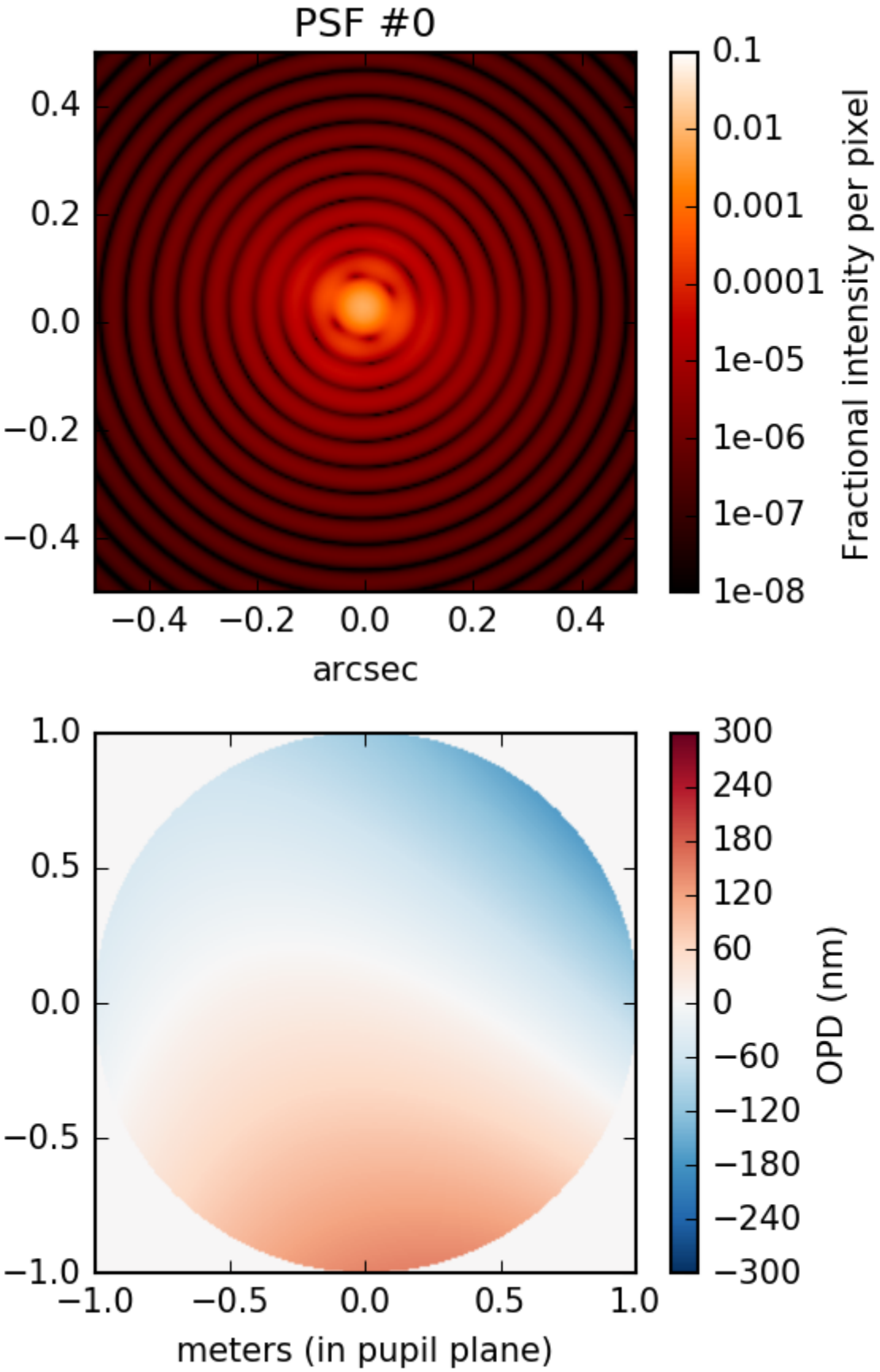


Fig. 2: PSF of a 2.0 meter diameter circular aperture with randomized combinations of aberrations applied.

6.1.3 Normalizing to desired peak-to-valley WFE

If you are trying to achieve a certain number of waves peak-to-valley in the optical path difference for your ZernikeWFE element, this normalization may be important! One example is defocus: In older conventions, the Zernike polynomial for defocus is $a(2\rho^2 - 1)$ and a is a defocus coefficient given in wavelengths of light center-to-peak. When expressing this in POPPY, there are a number of differences.

The first difference is that POPPY's ZernikeWFE deals in optical path difference rather than waves, so representing two waves of defocus at 1.5 um would be a coefficient of 3.0×10^{-6} meters.

The second difference is that we need a factor of a half to account for the fact that we are working in waves peak-to-valley rather than waves center-to-peak. That gives $\frac{3.0}{2} \times 10^{-6}$ meters

The final difference is that the normalization factor will have to be canceled out. The Zernike polynomial for defocus is $Z_n^m = Z_2^0$, and for terms with $m = 0$ the normalization coefficient applied is $\sqrt{n+1}$. (For all other terms except piston, it is $\sqrt{2}\sqrt{n+1}$. For piston, which has a constant value of 1.0, no additional normalization is necessary.) Therefore, to achieve 2 waves at 1.5 um, the coefficient supplied to ZernikeWFE should be $\frac{3.0}{2\sqrt{3}} \times 10^{-6}$ um.

This can be checked by comparing the `poppy.ThinLens` behavior with an equivalent ZernikeWFE optic. `ThinLens` takes a reference wavelength, a radius, and a number of waves (peak-to-valley) of defocus to apply.

First, the ThinLens:

```
osys = poppy.OpticalSystem()
circular_aperture = poppy.CircularAperture(radius=RADIUS)
osys.add_pupil(circular_aperture)

thinlens = poppy.ThinLens(nwaves=NWAVES, reference_wavelength=WAVELENGTH, radius=RADIUS)
osys.add_pupil(thinlens)

osys.add_detector(pixelscale=PIXSCALE, fov_arcsec=FOV)

psf_thinlens = osys.calc_psf(wavelength=WAVELENGTH, display_intermediates=True)
```

Second, the equivalent ZernikeWFE usage, with the appropriate coefficient:

```
defocus_coefficient = NWAVES * WAVELENGTH / (2 * np.sqrt(3))
coefficients_sequence = [0, 0, 0, defocus_coefficient]

osys = poppy.OpticalSystem()
circular_aperture = poppy.CircularAperture(radius=RADIUS)
osys.add_pupil(circular_aperture)
zernikewfe = poppy.ZernikeWFE(radius=RADIUS, coefficients=coefficients_sequence)
osys.add_pupil(zernikewfe)
osys.add_detector(pixelscale=PIXSCALE, fov_arcsec=FOV)

psf_zernikewfe = osys.calc_psf(wavelength=WAVELENGTH, display_intermediates=True)
```

If we plot `psf_thinlens`, `psf_zernikewfe`, and their difference (for confirmation) we will see:

6.2 ParameterizedWFE

The `ParameterizedWFE` class allows additional flexibility in expressing a wavefront error in a subset of cases that can be expressed as a linear combination of OPD (phase) terms in the pupil. Zernike polynomials are a special case of this. There are also "hexike" functions in POPPY, defined to be orthonormal on a unit hexagon, which can be used with ParameterizedWFE.

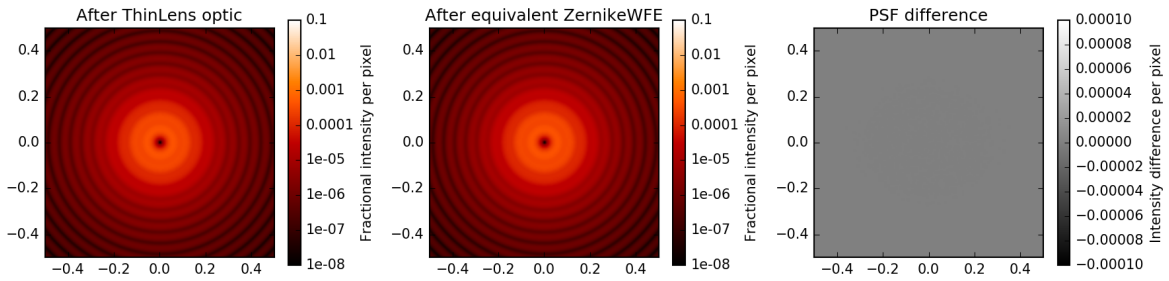


Fig. 3: Comparison of PSF from a ThinLens with 1 wave of defocus to a PSF from an equivalent ZernikeWFE optic.

The way you select a basis for your ParameterizedWFE optic is through the `basis_factory` keyword argument. This can be a function like `poppy.zernike.zernike_basis`, `poppy.zernike.hexike_basis`, or a function of your own design. These functions will be called during the calculation with some arguments (described below), and must return a data cube where the first (outermost) axis is planes corresponding to the first *n* terms of the basis.

Any basis factory function must accept an `nterms` argument indicating how many terms are to be calculated, `npix` as one way to set the number of pixels on a side for the planes of the data cube, `rho` and `theta` arrays which (in the absence of being passed `npix`) provide the radial and angular coordinate for each pixel in the pupil. (You can expect pixels with `rho` less than or equal to 1.0 to lie in the illuminated region of the pupil plane.) Lastly, they must accept an `outside` argument which defines the appropriate value for pixels in your basis terms that lie outside of the illuminated region. (Typical default values are 0.0 and `nan`.)

Rather than attempt to exhaustively describe the construction of such functions, we recommend consulting the code for `poppy.zernike.zernike_basis`, `poppy.zernike.hexike_basis`, and `poppy.ParameterizedWFE` to see how it all fits together.

6.2.1 Example comparing `zernike_basis` and `hexike_basis`

As a brief demonstration, let's adapt the *defocus example* above to use `zernike_basis`.

```
from poppy import zernike

osys = poppy.OpticalSystem()
circular_aperture = poppy.CircularAperture(radius=RADIUS)
osys.add_pupil(circular_aperture)
thinlens = poppy.ParameterizedWFE(radius=RADIUS,
    coefficients=[0, 0, 0, NWAVES * WAVELENGTH / (2 * np.sqrt(3))],
    basis_factory=zernike.zernike_basis   # here's where we specify the basis set
)
osys.add_pupil(thinlens)
osys.add_detector(pixelscale=PIXSCALE, fov_arcsec=FOV)

psf_with_zernikewfe = osys.calc_psf(wavelength=WAVELENGTH, display_intermediates=True)
```

If you plot this PSF, you will see one identical to that shown above. Now let's modify it to use `hexike_basis`. The first change is to replace the `CircularAperture` with a `HexagonAperture`. The second is to supply `basis_factory=zernike.hexike_basis`. Here's the code sample:

```
from poppy import zernike

osys = poppy.OpticalSystem()
hex_aperture = poppy.HexagonAperture(side=RADIUS)  # modified to use hexagonal aperture
```

(continues on next page)

(continued from previous page)

```
osys.add_pupil(hex_aperture)
thinlens = poppy.ParameterizedWFE(radius=RADIUS,
   coefficients=[0, 0, 0, NWAVES * WAVELENGTH / (2 * np.sqrt(3))],
   basis_factory=zernike.hexike_basis   # now using the 'hexike' basis
)
osys.add_pupil(thinlens)
osys.add_detector(pixelscale=PIXSCALE, fov_arcsec=FOV)

psf_with_hexikewfe = osys.calc_psf(wavelength=WAVELENGTH, display_intermediates=True, return_
↪intermediates=True)
```

If we plot the new PSF, we will get a hexagonal PSF with a central minimum typical of a single wave of defocus. (Using the same setup with a hexagon aperture and a *Zernike* basis gets a much less pronounced central minimum, as the Zernike polynomials are only orthonormal over the unit circle.)

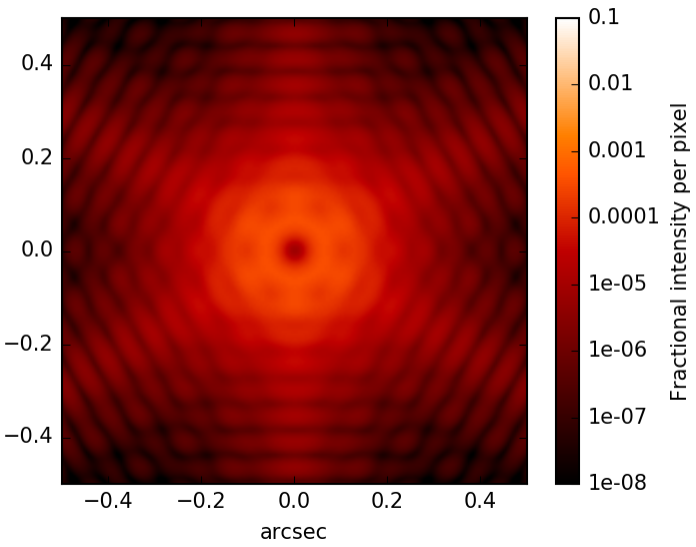


Fig. 4: A defocused PSF from a hexagonal aperture and a "Hexike" polynomial term to express the defocus.

6.3 Basis functions for modeling wavefront error

Zernike polynomials are most generally referred to via a tuple of indices (n, m). The following functions compute Zernikes given (n, m).

- `poppy.zernike.zernike()` computes a 2D array for a specified Zernike
- `poppy.zernike.str_zernike()` returns the analytic Zernike polynomial in LaTeX formatting.

But in many cases it is more practical to reference Zernikes via a 1-dimensional index. `poppy` does this using the so-called Noll indexing convention (Noll et al. JOSA 1976). Conversion from 1-D to 2-D indices is via the `poppy.zernike.noll_indices()` function.

- `poppy.zernike.zernike1()` returns the Zernike polynomial Z_j.
- `poppy.zernike.cached_zernike1()` is a faster but somewhat less flexible computation of Z_j.
- `poppy.zernike.zern_name()` returns a descriptive name such as "spherical" or "coma" for a given Z_j.

Frequently we want to work with an entire basis set of Zernike polynomials at once. poppy implements this via "basis functions", which each return 3-d ndarray datacubes including the first n terms of a given basis set. Several such bases are available. Each of these functions takes as arguments the number of terms desired in the basis, as well as the desired sampling (how many pixels across each side of the output arrays).

- poppy.zernike.zernike_basis() is the standard Zernike polynomials over a unit circle
- poppy.zernike.hexike_basis() is the Hexikes over a unit hexagon
- poppy.zernike.arbitrary_basis() uses the Gram-Schmidt orthonormalization algorthm to generate an orthonormal basis for any supplied arbitrary aperture shape.
- poppy.zernike.Segment_Piston_Basis implements bases defined by hexagonal segments controlled in piston only. Unlike the prior basis functions, this one is a function class: first you must instantiate it to specify the desired number of hexagons and other pupil geometry information, and then you can use the resulting object as a basis function to compute the hexikes.
- poppy.zernike.Segment_PTT_Basis is similar, but each segment can be controlled in piston, tip, and tilt.

Using any of the above basis functions, OPD arrays can be decomposed into coefficients per each term, or conversely OPD arrays can be generated from provided coefficients. There are several functions provided for OPD decomposition, tuned for different usage scenarios.

- poppy.zernike.opd_from_zernikes() generates an OPD from providend coefficients. Despite the name this function can actually be used with any of the above basis sets.
- poppy.zernike.opd_expand() projects a given OPD into a Zernike or Hexike basis, and returns the resulting coefficients. This works best when dealing with cases closer to ideal, i.e. Zernikes over an actually-circular aperture.
- poppy.zernike.opd_expand_nonorthonormal() does the same, but uses an alternate iterative algorithm that works better when dealing with basis sets that are not strictly orthonormal over the given aperture.
- poppy.zernike.opd_expand_segments() uses the same iterative algorithm but with some adjustments to better handle spatially disjoint basis elements such as different segments. Use this for best results if you're dealing with a segmented aperture.

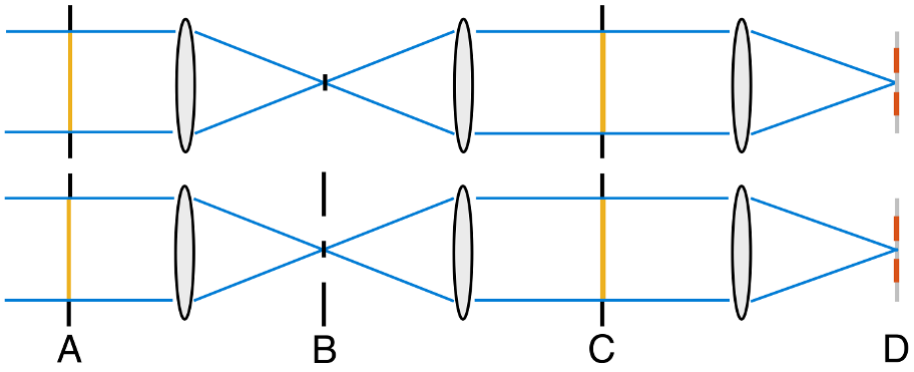
CHAPTER 7

Efficient Lyot coronagraph propagation

Poppy has extensive functionality to faciliate the modeling of coronagraph point spread functions. In addition to the general summary of those capabilities here, see the examples in the notebooks subdirectory: POPPY Examples and MatrixFTCoronagraph_demo.

7.1 Introduction

By default, an optical system defined in Poppy uses the Fast Fourier Transform (FFT) to propagate the scalar field between pupil and image planes. While the FFT is a powerful tool for general Fraunhofer diffraction calculations, it is rarely the most computationally efficient approach for a coronagraph model. Consider the two coronagraph schematics below, from Zimmerman et al (2016):



The upper design in the figure represents the classical Lyot coronagraph and its widely implemented, optimized descendent, the apodized pupil Lyot coronagraph (APLC). In this case an intermediate focal plane (labeled B) is occulted by a round, opaque mask. By applying the principle of electromagnetic field superposition, combined with knowledge of how FFT complexity scales with array size, Soummer et al. (2007) showed that the number of operations needed to compute the PSF is greatly reduced by replacing the FFT with discrete Fourier transfoms, implemented in a vectorized fashion and spatially restricted to the *occulted* region of the intermediate focal plane. This is the now widely-used **semi-analytical** computational method for numerically modeling Lyot coronagraphs.

The lower design in the above figure shows a slightly different Lyot coronagraph design case. Here the focal plane mask (FPM) is a diaphragm that restricts the outer edge of the transmitted field. Zimmerman et al. (2016) showed how this design variant can solve the same starlight cancellation problem, in particular for the baseline design of WFIRST. With this FPM geometry, the superposition simplification of Soummer et al. (2007) is not valid. However, again the execution time is greatly reduced by using discrete, vectorized Fourier transforms, now spatially restricted to the *transmitted* region of the intermediate focal plane.

In Poppy, two subclasses of OpticalSystem exploit the computational methods described above: SemiAnalyticCoronagraph and MatrixFTCoronagraph. Let's see how to make use of these subclasses to speed up Lyot corongraph PSF calculations.

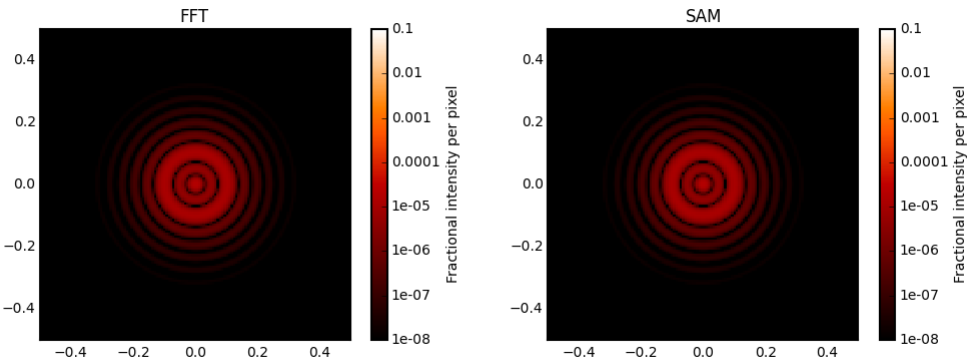
7.2 Lyot coronagraph using the SemiAnalytic subclass

In this example we consider a Lyot coronagraph with a conventional, opaque occulting spot. This configuration corresponds to the upper half of the schematic described above.

The semi-analytic method is specified by first creating an OpticalSystem as usual, and then casting it to a SemiAnalyticCoronagraph class (which has a special customized propagation method implementing the alternate algorithm):

The following code performs the same calculation both with semi-analytical and FFT propagation, and compares their speeds:

```
radius = 6.5/2
lyot_radius = 6.5/2.5
pixelscale = 0.060

osys = poppy.OpticalSystem("test", oversample=8)
osys.add_pupil( poppy.CircularAperture(radius=radius), name='Entrance Pupil')
osys.add_image( poppy.CircularOcculter(radius = 0.1) )
osys.add_pupil( poppy.CircularAperture(radius=lyot_radius), name='Lyot Pupil')
osys.add_detector(pixelscale=pixelscale, fov_arcsec=5.0)


plt.figure(1)
sam_osys = poppy.SemiAnalyticCoronagraph(osys, oversample=8, occulter_box=0.15)

import time
t0s = time.time()
psf_sam = sam_osys.calc_psf(display_intermediates=True)
t1s = time.time()

plt.figure(2)
t0f = time.time()
psf_fft = osys.calc_psf(display_intermediates=True)
t1f = time.time()

plt.figure(3)
plt.clf()
plt.subplot(121)
poppy.utils.display_psf(psf_fft, title="FFT")
plt.subplot(122)
poppy.utils.display_psf(psf_sam, title="SAM")

print "Elapsed time, FFT:  %.3s" % (t1f-t0f)
print "Elapsed time, SAM:  %.3s" % (t1s-t0s)
```



On a circa-2010 Mac Pro, the results are dramatic:

```
Elapsed time, FFT:  62.
Elapsed time, SAM:  4.1
```

7.3 Lyot coronagraph using the MatrixFTCoronagraph subclass

This coronagraph uses an annular diaphragm in the intermediate focal plane, corresponding to the lower half of the diagram above.

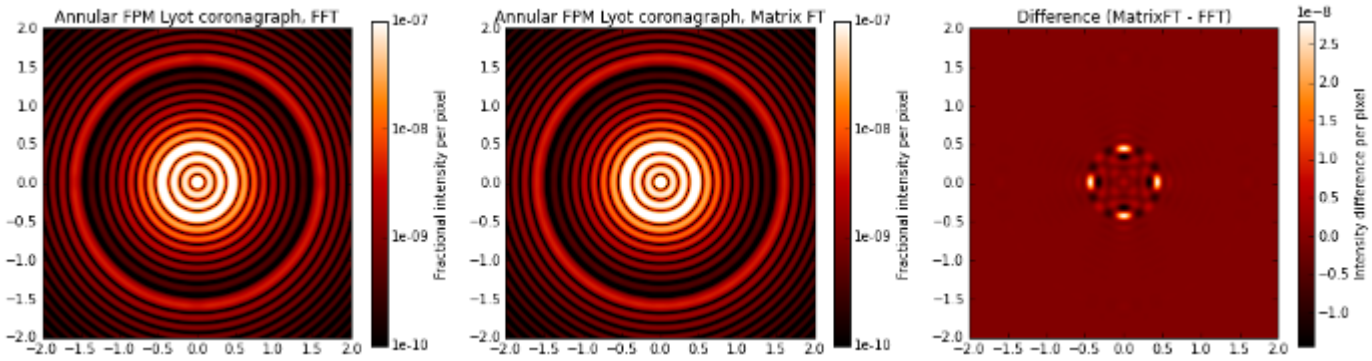
The procedure to enable the MatrixFTCoronagraph propagation is analogous to the SemiAnalytic case. We create an OpticalSystem as usual, and then cast it to a MatrixFTCoronagraph class.

Again we will compare the execution time with the FFT case.:

```
D = 2.
wavelen = 1e-6
ovsamp = 8

# Annular diaphragm FPM, inner radius ~ 4 lam/D, outer rad ~ 16 lam/D
fftcoron_annFPM_osys = poppy.OpticalSystem(oversamp=ovsamp)
fftcoron_annFPM_osys.add_pupil( poppy.CircularAperture(radius=D/2) )
spot = poppy.CircularOcculter( radius=0.4  )
diaphragm = poppy.InverseTransmission( poppy.CircularOcculter( radius=1.6 ) )
annFPM = poppy.CompoundAnalyticOptic( opticslist = [diaphragm, spot] )
fftcoron_annFPM_osys.add_image( annFPM )
fftcoron_annFPM_osys.add_pupil( poppy.CircularAperture(radius=0.9*D/2) )
fftcoron_annFPM_osys.add_detector( pixelscale=0.05, fov_arcsec=4. )

# Re-cast as MFT coronagraph with annular diaphragm FPM
matrixFTcoron_annFPM_osys = poppy.MatrixFTCoronagraph( fftcoron_annFPM_osys, occulter_box=diaphragm.
↪uninverted_optic.radius_inner )
t0_fft = time.time()
annFPM_fft_psf, annFPM_fft_interm = fftcoron_annFPM_osys.calc_psf(wavelen, display_intermediates=True,\
                                                    return_intermediates=True)
t1_fft = time.time()

t0_mft = time.time()
annFPM_mft_psf, annFPM_mft_interm = matrixFTcoron_annFPM_osys.calc_psf(wavelen, display_
↪intermediates=True,\
                                                        return_intermediates=True)
t1_mft = time.time()
```

Plot the results:

```
plt.figure(figsize=(16,3.5))
plt.subplots_adjust(left=0.10, right=0.95, bottom=0.02, top=0.98, wspace=0.2, hspace=None)
plt.subplot(131)
ax_fft, cbar_fft = poppy.display_psf(annFPM_fft_psf, vmin=1e-10, vmax=1e-7, title='Annular FPM Lyot␣
↪coronagraph, FFT',
                            return_ax=True)
plt.subplot(132)
poppy.display_psf(annFPM_mft_psf, vmin=1e-10, vmax=1e-7, title='Annular FPM Lyot coronagraph, Matrix FT
↪')
plt.subplot(133)
diff_vmin = np.min(annFPM_mft_psf[0].data - annFPM_fft_psf[0].data)
diff_vmax = np.max(annFPM_mft_psf[0].data - annFPM_fft_psf[0].data)
poppy.display_psf_difference(annFPM_mft_psf, annFPM_fft_psf, vmin=diff_vmin, vmax=diff_vmax, cmap='gist_
↪heat')
plt.title('Difference (MatrixFT - FFT)')
```



Print some of the propagation parameters:

```
lamoD_asec = wavelen/fftcoron_annFPM_osys.planes[0].pupil_diam * 180/np.pi * 3600
print "System diffraction resolution element scale (lambda/D) in arcsec: %.3f" % lamoD_asec
print "Array width in first focal plane, FFT: %d" % annFPM_fft_interm[1].amplitude.shape[0]
print "Array width in first focal plane, MatrixFT: %d" % annFPM_mft_interm[1].amplitude.shape[0]
print "Array width in Lyot plane, FFT: %d" % annFPM_fft_interm[2].amplitude.shape[0]
print "Array width in Lyot plane, MatrixFT: %d" % annFPM_mft_interm[2].amplitude.shape[0]

System diffraction resolution element scale (lambda/D) in arcsec: 0.103
Array width in first focal plane, FFT: 8192
Array width in first focal plane, MatrixFT: 248
Array width in Lyot plane, FFT: 8192
Array width in Lyot plane, MatrixFT: 1024
```

Compare the elapsed time:

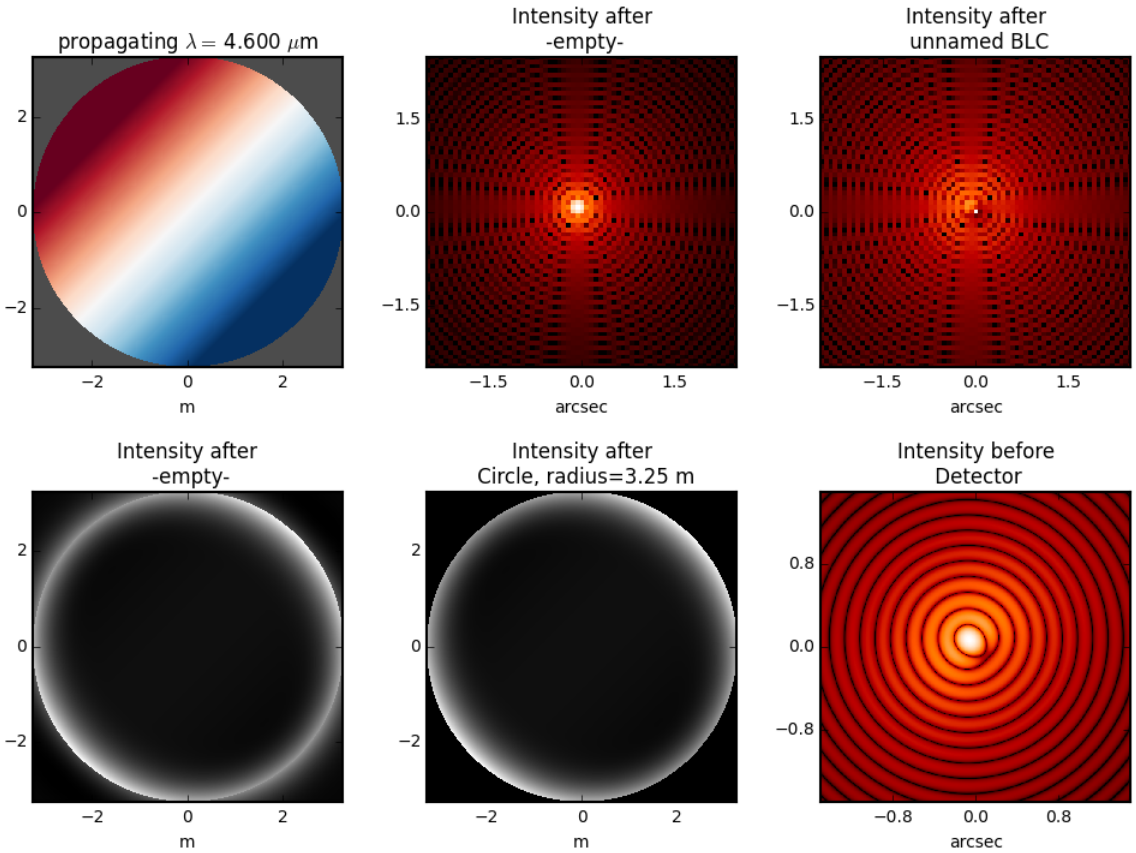
```
print "Elapsed time, FFT:  %.1f s" % (t1_fft-t0_fft)
print "Elapsed time, Matrix FT:  %.1f s" % (t1_mft-t0_mft)

Elapsed time, FFT:  142.0 s
Elapsed time, Matrix FT:  3.0 s
```

7.4 Band-limited coronagraph

Depending on the specific implementation, a Lyot coronagraph with a band-limited occulter can also benefit from the semi-analytical method in Poppy. For additional band-limited coronagraph examples, see the JWST NIRCam coronagraph modes included in WebbPSF.

As an example of a more complicated coronagraph PSF calculation than the ones above, here's a NIRCam-style band limited coronagraph with the source not precisely centered:

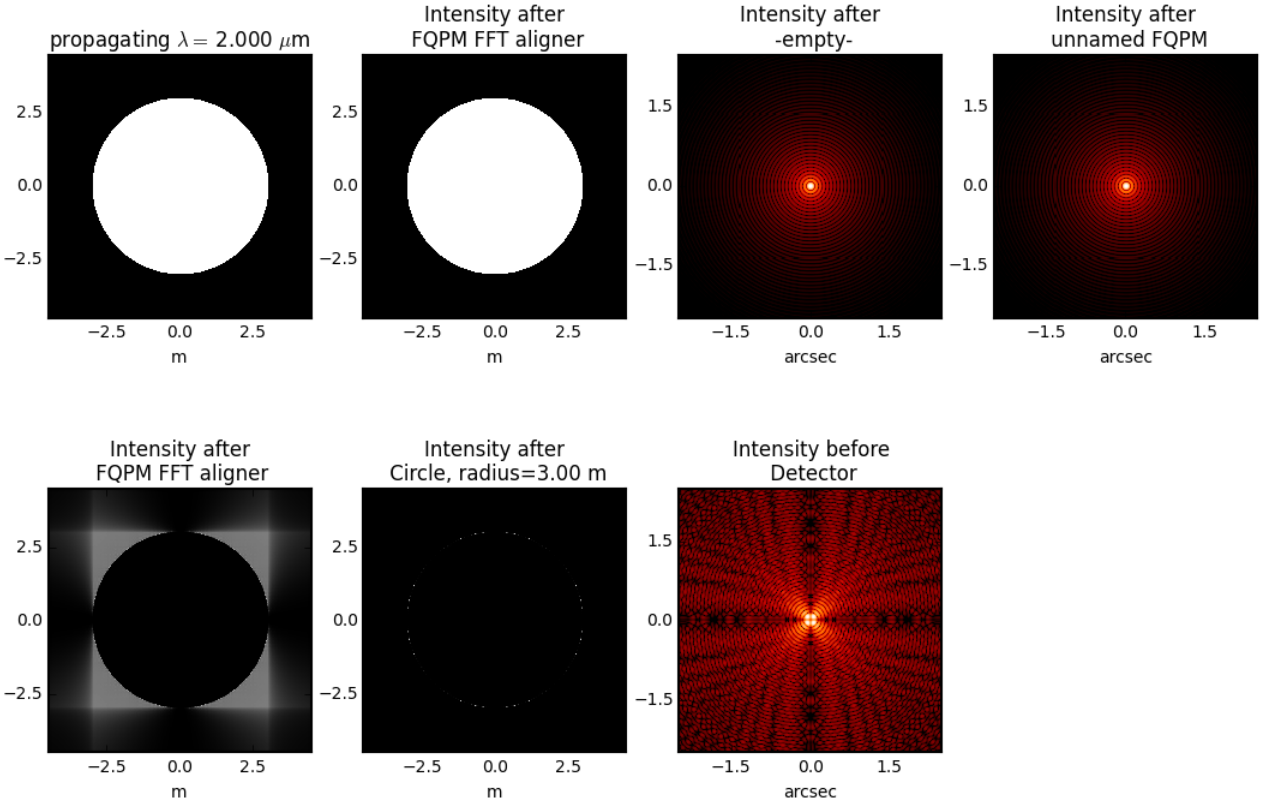
```
oversample=2
pixelscale = 0.010  #arcsec/pixel
wavelength = 4.6e-6

osys = poppy.OpticalSystem("test", oversample=oversample)
osys.add_pupil(poppy.CircularAperture(radius=6.5/2))
osys.add_image()
osys.add_image(poppy.BandLimitedCoron(kind='circular',  sigma=5.0))
osys.add_pupil()
osys.add_pupil(poppy.CircularAperture(radius=6.5/2))
osys.add_detector(pixelscale=pixelscale, fov_arcsec=3.0)

osys.source_offset_theta = 45.
osys.source_offset_r =  0.1  # arcsec
psf = osys.calc_psf(wavelength=wavelength, display_intermediates=True)
```



7.5 FQPM coronagraph

Due to the wide (ideally infinite) spatial extension of its focal plane phase-shifting optic, the four-quadrant phase mask (FQPM) coronagraphs relies on FFT propagation. Another unique complication of the FQPM coronagraph class is its array alignment requirement between the FFT result in the intermediate focal plane with the center of the phase mask. This is done using a virtual optic called an 'FQPM FFT aligner' as follows:

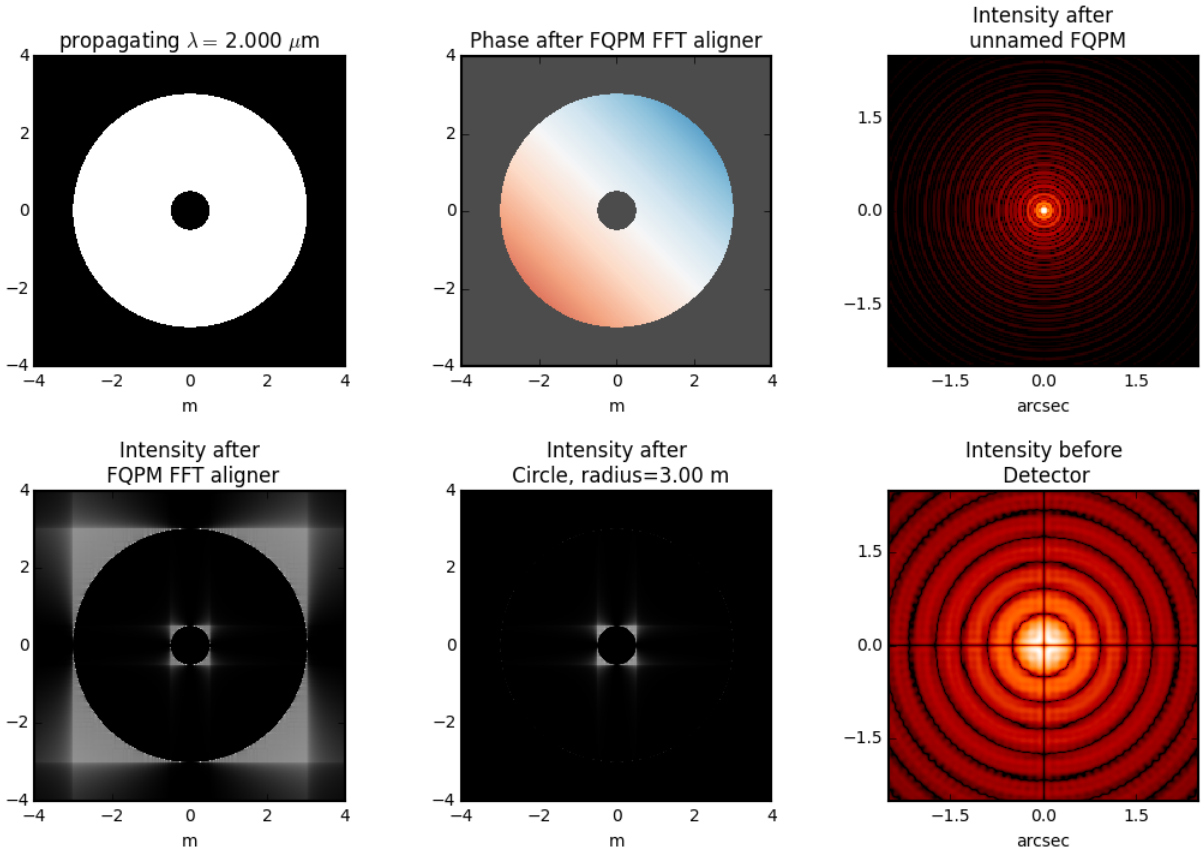
```
optsys = poppy.OpticalSystem()
optsys.add_pupil( poppy.CircularAperture( radius=3, pad_factor=1.5)) #pad display area by 50%
optsys.add_pupil( poppy.FQPM_FFT_aligner())   # ensure the PSF is centered on the FQPM cross hairs
optsys.add_image()  # empty image plane for "before the mask"
optsys.add_image( poppy.IdealFQPM(wavelength=2e-6))
optsys.add_pupil( poppy.FQPM_FFT_aligner(direction='backward'))  # undo the alignment tilt after going␣
↪back to the pupil plane
optsys.add_pupil( poppy.CircularAperture( radius=3)) # Lyot mask - change radius if desired
optsys.add_detector(pixelscale=0.01, fov_arcsec=10.0)


psf = optsys.calc_psf(wavelength=2e-6, display_intermediates=True)
```



7.6 FQPM on an Obscured Aperture (demonstrates compound optics)

As a variation, we can add a secondary obscuration. This can be done by creating a compound optic consisting of the circular outer aperture plus an opaque circular obscuration. The latter we can make using the InverseTransmission class.

```
primary = poppy.CircularAperture( radius=3)
secondary = poppy.InverseTransmission( poppy.CircularAperture(radius=0.5) )
aperture = poppy.CompoundAnalyticOptic( opticslist = [primary, secondary] )

optsys = poppy.OpticalSystem()
optsys.add_pupil( aperture)
optsys.add_pupil( poppy.FQPM_FFT_aligner())   # ensure the PSF is centered on the FQPM cross hairs
optsys.add_image( poppy.IdealFQPM(wavelength=2e-6))
optsys.add_pupil( poppy.FQPM_FFT_aligner(direction='backward'))  # undo the alignment tilt after going
↪back to the pupil plane
optsys.add_pupil( poppy.CircularAperture( radius=3)) # Lyot mask - change radius if desired
optsys.add_detector(pixelscale=0.01, fov_arcsec=10.0)

optsys.display()

psf = optsys.calc_psf(wavelength=2e-6, display_intermediates=True)
```



CHAPTER 8

Fresnel Propagation

POPPY now includes support for Fresnel propagation as well as Fraunhofer. Particular credit is due to Ewan Douglas for initially developing this code. This substantial upgrade to poppy enables calculation of wavefronts propagated arbitrary distances in free space, for applications such as Gaussian beam propagation and modeling of Talbot effect mixing between phase and amplitude aberrations.

Caution: The Fresnel code has been cross-checked against the PROPER library by John Krist to verify accuracy and correctness of output. A test suite is provided along with poppy in the tests subdirectory and users are encouraged to run these tests themselves.

8.1 Usage of the Fresnel code

The API has been kept as similar as possible to the original Fraunhofer mode of poppy. There are `FresnelWavefront` and `FresnelOpticalSystem` classes, which can be used for the most part similar to the `Wavefront` and `OpticalSystem` classes.

Users are encouraged to consult the Jupyter notebook Fresnel_Propagation_Demo for examples of how to use the Fresnel code.

8.1.1 Key Differences from Fraunhofer mode

The Fresnel propagation API necessarily differs in several ways from the original Fraunhofer API in poppy. Let's highlight a few of the key differences. First, when we define a Fresnel wavefront, the first argument specifies the desired diameter of the wavefront, and must be given as an astropy.Quantity of dimension length:

```
import astropy.units as u
wf_fresnel = poppy.FresnelWavefront(0.5*u.m,wavelength=2200e-9,npix=npix,oversample=4)
# versus:
wf_fraunhofer = poppy.Wavefront(diam=0.5, wavelength=2200e-9,npix=npix,oversample=4)
```

The Fresnel code relies on the Quantity framework to enforce consistent units and dimensionality. You can use any desired unit of length, from nanometers to parsecs and beyond, and the code will convert units appropriately. This also shows up when requesting an optical propagation. Rather than having implicit transformations between pupil and image planes, for Fresnel propagation a specific distance must be given. This too is a Quantity giving a length.

```
wf.propagate_fresnel(5*u.km)
```

The parameters of a Gaussian beam may be modified (making it converging or diverging) by adding optical power. In poppy this is represented with the `QuadraticLens` class. This is so named because it applies a purely quadratic phase term, i.e. representative of a parabolic mirror or a lens considered in the paraxial approximation. Right now, only the Fresnel `QuadraticLens` class will actually cause the Gaussian beam parameters to change. You won't get that effect by adding wavefront error with some other `OpticalElement` class.

8.1.2 Using the FreselOpticalSystem class

Just like the `OpticalSystem` serves as a high-level container for `OpticalElement` instances in Fraunhofer propagation, the `FresnelOpticalSystem` serves the same purpose in Fresnel propagation. Note that when adding an `OpticalElement` to the `FresnelOpticalSystem`, you use the function `add_optic()` and must specify a physical distance separating that optic from the previous optic, again as an astropy.Quantity of dimension length. This replaces the `add_image` and `add_pupil` methods used in Fraunhofer propagation. For example:

```
osys = poppy.FresnelOpticalSystem(pupil_diameter = 0.05*u.m, npix = npix, beam_ratio = 0.25)
osys.add_optic(poppy.CircularAperture(radius=0.025) )
osys.add_optic(poppy.ScalarTransmission(), distance = 10*u.m )
```

If you want the output from a Fresnel calculation to have a particular pixel sampling, you may either (1) adjust the `npix` and `oversample` or `beam_ratio` parameters so that the propagation output naturally has the desired sampling, or (2) add a `Detector` instance as the *last* optical plane to define the desired sampling, in which case the output wavefront will be interpolated onto the desired sampling and number of pixels. Note that when specifying Detectors in a Fresnel system you must use physical sizes of pixels, e.g. `10*u.micron/u.pixel`, and NOT angular sizes in arcsec/pixel like in a regular Fraunhofer OpticalSystem. For instance:

```
osys.add_detector(pixelscale=20*u.micron/u.pixel, fov_pixels=512)
```

8.1.3 Example Jupyter Notebooks

Fresnel tutorial notebook

For more details and examples of code usage, consult the Jupyter notebook Fresnel_Propagation_Demo. In addition to details on code usage, this includes a worked example of a Fresnel model of the Hubble Space Telescope.

A non-astronomical example

A worked example of a compound microscope in POPPY is available here, reproducing the microscope example case provided in the PROPER manual.

8.1.4 Fresnel calculations with Physical units

A subclass, `PhysicalFresnelWavefront`, enables calculations using wavefronts scaled to physical units, i.e. volts/meter for electric field and watts for total intensity (power). This code was developed and contributed by Phillip

Springer.

See this notebook for examples and further discussion.

8.2 References

The following references were helpful in the development of this code.

- Goodman, Fourier Optics
- **Lawrence, G. N. (1992), Optical Modeling, in Applied Optics and Optical Engineering., vol. XI,** edited by R. R. Shannon and J. C. Wyant., Academic Press, New York.
- IDEX Optics and Photonics(n.d.), Gaussian Beam Optics
- **Krist, J. E. (2007), PROPER: an optical propagation library for IDL** vol. 6675, p. 66750P-66750P-9.
- Andersen, T., and A. Enmark (2011), Integrated Modeling of Telescopes, Springer Science & Business Media.

CHAPTER 9

Options

9.1 Output PSF Normalization

Output PSFs can be normalized in different ways, based on the `normalization` keyword to `calc_psf`. The options are:

- `normalize="first"`: The wavefront is normalized to total intensity 1 over the entrance pupil. If there are obstructions downstream of the entrance pupil (e.g. coronagraph masks) then the output PSF intensity will be < 1.
- `normalize="last"`: The output PSF's integrated total intensity is normalized to 1.0, over whatever FOV that PSF has. (Note that this
- `normalize="exit_pupil"`: The wavefront is normalized to total intensity 1 at the exit pupil, i.e. the last pupil in the optical system. This means that the output PSF will have total intensity 1.0 if integrated over an arbitrarily large aperture. The total intensity over any finite aperture will be some number less than one. In other words, this option is equivalent to saying "Normalize the PSF to have integrated intensity 1 over an infinite aperture."

9.2 Logging

As noted on the *Examples* page, Poppy uses the Python `logging` mechanism for log message display. The default "info" level provides a modest amount of insight into the major steps of a calculation; the "debug" level provides an exhaustive and lengthy description of everything you could possibly want to know. You can switch between these like so:

```
import logging
logging.basicConfig(level=logging.INFO)
logging.basicConfig(level=logging.DEBUG)
```

See the python logging docs for more information and extensive options for directing log output to screen or file.

9.3 Configuration

Poppy makes use of the Astropy configuration system to store settings persistently between sessions. These settings are stored in a file in the user's home directory, for instance `~/.astropy/config/poppy.cfg`. Edit this text file to adjust settings.

Setting	Description	Default
use_multiprocessing	Should PSF calculations run in parallel using multiple processors?	False
n_processes	Maximum number of additional worker processes to spawn.	4
use_fftw	Should the pyFFTW library be used (if it is present)?	True
autosave_fftw_wisdom	Should POPPY automatically save and reload FFTW 'wisdom' (i.e. timing measurements of different FFT variants)	True
default_image_display_fov	Default display field of view for PSFs, in arcsec	5
default_logging_level	Default verbosity of logging to Python's logging framework	INFO
enable_speed_tests	Enable additional verbose logging of execution timing	False
enable_flux_tests	Enable additional verbose logging of flux conservation tests	False

CHAPTER 10

Extending POPPY by defining your own optics and instruments

POPPY is designed to make it straightforward to implement your own custom optics classes, which will interoperate with all the built-in classes. Conceptually all that is needed is defining the `get_transmission` and/or `get_opd` functions for each new class.

Many examples of this can be found in `poppy/optics.py`

10.1 Defining a custom optic from an analytic function

The complex phasor of each optic is calculated automatically from that optic's transmission (i.e. the throughput for the amplitude of the electromagnetic field) and optical path difference (i.e. the propagation delay in the phase of the electromagnetic field). Both of these quantities may vary as a function of position across the optic, and as a function of wavelength.

`AnalyticOpticalElement` subclasses must implement either or both of the functions `get_transmission()` and `get_opd()`. Each takes a `Wavefront` as its sole argument besides `self`. All other necessary parameters should be set up as part of the `__init__` function defining your optic.

Note: This is new in version 0.5 of poppy; prior versions used a single function getPhasor to handle computing the entire complex phasor in one step including both the transmission and OPD components. Version 0.5 now provides better flexibility and extensibility by allowing the transmission and OPD components to be defined in separate functions, and automatically takes care of combining them to produce the complex phasor behind the scenes.

Example skeleton code:

```
class myCustomOptic(poppy.AnalyticOpticalElement):
    def __init__(self, *args, **kwargs):
        """ If your optic has adjustible parameters, then save them as attributes here """
        poppy.AnalyticOpticalElement.__init__(**kwargs)

    def get_opd(self,wave):
```

(continues on next page)

(continued from previous page)

```
        y, x = self.get_coordinates(wave)
        opd = some_function(x,y, wave.wavelength, self)
        return opd

    def get_transmission(self, wave):
        y, x = self.get_coordinates(wave)
        transmission = other_function(x,y, wave.wavelength, self)
        return transmission

    # behind the scenes poppy  will calculate:
    #    phasor = transmission = np.exp(1.j * 2 * np.pi / wave.wavelength * opd)
```

Note the use of the `self.get_coordinates()` helper function, which returns `y` and `x` arrays giving the coordinates as appopriate for the sampling of the supplied `wave` object (by default in units of meters for most optics such as pupil planes, in arcseconds for image plane optics). You can use these coordinates to calculate the transmission and path delay appropriate for your optic. If your optic has wavelength dependent properties, access the `wave.wavelength` property to determine the the appropriate wavelength; this will be in units of meters.

The `get_coordinates()` function automatically includes support for offset shifts and rotations for any analytic optic: just add a `shift_x`, `shift_y` or `rotation` attribute for your optic object, and the coordinates will be shifted accordingly. These parameters should be passed to `poppy.AnalyticOpticalElement.__init__` via the `**kwargs` mechanism.

10.2 Defining a custom optic from a FITS file

Of course, any arbitrary optic can be represented in discrete form in 2D arrays and then read into poppy using the FITSOpticalElement class. The physical parameters of the array are defined using fitsheaders.

The transmission array should contain floating point values between 0.0 and 1.0. These represent the local transmission of the electric field amplitude, not the total intensity.

The OPD array should contain floating point numbers (positive and negative) representing a path delay in some physical units. The unit must be specified using the `BUNIT` keyword; allowed BUNITs are 'meter', 'micron', 'nanometer' and their standard metric abbreviations.

If you are using both an OPD and transmission together to define your optics, the arrays must have the same size.

The spatial or angular scale of these arrays must also be indicated by a FITS header keyword. By default, poppy checks for the keyword `PIXSCALE` for image plane pixel scale in arcseconds/pixel or `PUPLSCAL` for pupil plane scale in meters/pixel. However if your FITS file uses some alternate keyword, you can specify that keyword name with the `pupilscale=` argument in the call to the `FITSOpticalElement` constructor, i.e.:

```
myoptic = poppy.FITSOpticalElement(transmission='transfile.fits', opd='opdfile.fits', pupilscale=
↪"PIXELSCL")
```

Lastly if there is no such keyword available, you can specify the numerical scale directly via the same keyword by providing a float instead of a string:

```
myoptic = poppy.FITSOpticalElement(transmission='transfile.fits', opd='opdfile.fits', pupilscale=0.020)
```

10.3 Creating a custom instrument

POPPY provides an `Instrument` class to simplify certain types of calculations. For example, the WebbPSF project uses `Instrument` subclasses to provide selectable filters, pupil masks, and image masks for the instruments on JWST.

Any calculation you can set up with a bare POPPY `OpticalSystem` can be wrapped with an `Instrument` to present a friendlier API to end users. The `Instrument` will hold the selected instrument configuration and calculation options, passing them to a private method `_getOpticalSystem()` which implementors must override to build the `OpticalSystem` for the PSF calculation.

The general notion of an `Instrument` is that it consists of both

1. An optical system implemented in the usual fashion, optionally with several configurations such as selectable image plane or pupil plane stops or other adjustable properties, and
2. Some defined spectral bandpass(es) such as selectable filters. If the `pysynphot` module is available, it will be used to perform careful synthetic photometry of targets with a given spectrum observed in the given bandpass. If `pysynphot` is not installed, the code will fall back to a much simpler model assuming constant number of counts vs wavelength.

Configurable options such as optical masks and filters are specified as properties of the instrument instance; an appropriate `OpticalSystem` will be generated when the `calc_psf()` method is called.

The `Instrument` is fairly complex, and has a lot of internal submethods used to modularize the calculation and allow subclassing and customization. For developing your own instrument classes, it may be useful to start with the instrument classes in WebbPSF as worked examples.

You will at a minimum want to override the following class methods:

- _getOpticalSystem
- _getFilterList
- _getDefaultNLambda
- _getDefaultFOV
- _getFITSHeader

For more complicated systems you may also want to override:

- _validateConfig
- _getSynphotBandpass
- _applyJitter

An `Instrument` will get its configuration from three places:

(1) The `__init__` method of the `Instrument` subclass

During `__init__`, the subclass can set important attributes like `pixelscale`, add a custom `pupil` optic and OPD map, and set a default filter. (n.b. The current implementation may not do what you expect if you are accustomed to calling the superclass' `__init__` at the end of your subclass' `__init__` method. Look at the implementation in `poppy/instrument.py` for guidance.)

(2) The `options` dictionary attribute on the `Instrument` subclass

The options dictionary allows you to set a subset of options that are loosely considered to be independent of the instrument configuration (e.g. filter wheels) and of the particular calculation. This includes offsetting the source from the center of the FOV, shifting the pupil, applying jitter to the final image, or forcing the parity of the final output array.

Users are free to introduce new options by documenting an option name and retrieving the value at an appropriate point in their implementation of `_getOpticalSystem()` (to which the options dictionary is passed as keyword argument `options`).

(3) The `calc_psf()` method of the `Instrument` subclass

For interoperability, it's not recommended to change the function signature of `calc_psf()`. However, it is an additional way that users will pass configuration information into the calculation, and a starting point for more involved customization that cannot be achieved by overriding one of the private methods above.

Be warned that the `poppy.Instrument` API evolved in tandem with WebbPSF, and certain things are subject to change as we extend it to use cases beyond the requirements of WebbPSF.

CHAPTER 11

POPPY Class Listing

The key classes for POPPY are `OpticalSystem` and the various `OpticalElement` classes (of which there are many). There is also a `Wavefront` class that is used internally, but users will rarely need to instantiate that directly. Results are returned as FITS files, specifically astropy.io.fits.HDUList objects.

`OpticalSystem` is in essence a container for `OpticalElement` instances, which handles creating input wavefronts, propagating them through the individual optics, and then combining the results into a broadband output point spread function.

The `Instrument` class provides a framework for developing high-level models of astronomical instruments. An `OpticalSystem` does not include any information about spectral bandpasses, filters, or light source properties, it just propagates whatever specified list of wavelengths and weights it's provided with. The `Instrument` class provides the machinery for handling filters and sources to generate weighted source spectra, as well as support for configurable instruments with selectable mechanisms, and system-level impacts on PSFs such as pointing jitter.

Note that the `Instrument` class should not be used directly but rather is subclassed to implement the details of your particular instrument. See its class documentation for more details.

11.1 Optical Systems

- `OpticalSystem` is the fundamental optical system class, that propagates `Wavefront` objects between optics using Fourier transforms.
- `SemiAnalyticCoronagraph` implements the semi-analytic coronagraphic propagation algorithm of Soummer et al.
- `MatrixFTCoronagraph` enables efficient propagation calculations for Lyot coronagraphs with diaphragm-type focal plane masks, relevant to the WFIRST coronagraph and described by Zimmerman et al. (2016).

11.2 Optical Elements

- `OpticalElement` is the fundamental building block

- FITSOpticalElement implements optics defined numerically on discrete grids read in from FITS files
- AnalyticOpticalElement implements optics defined analytically on any arbitrary sampling. There are many of these.
  - ScalarTransmission is a simple floating-point throughput factor.
  - CompoundAnalyticOptic allows multiple analytic optics to be merged into one container object
- Pupil plane analytic optics include:
  - CircularAperture
  - SquareAperture
  - RectangleAperture
  - HexagonAperture
  - MultiHexagonAperture
  - NgonAperture
  - SecondaryObscuration
  - ThinLens
  - FQPM_FFT_aligner
- **Image plane analytic optics include:**
  - RectangularFieldStop
  - SquareFieldStop
  - CircularOcculter
  - BarOcculter
  - BandLimitedCoron
  - IdealFQPM
- InverseTransmission allows any optic, whether analytic or discrete, to be flipped in sign, a la the Babinet principle.
- Rotation represents a rotation of the axes of the wavefront, for instance to change coordinate systems between two optics that are rotated with respect to one another. The axis of rotation must be the axis of optical propagation.
- _poppy.CoordinateInversion represents a flip in orientation of the X or Y axis, or both at once.
- Detector represents a detector with some fixed sampling and pixel scale.

11.3 Wavefront Error Optical Elements

- poppy.wfe.ZernikeWFE
- poppy.wfe.SineWaveWFE

CHAPTER 12

API Reference

12.1 poppy Package

Physical Optics Propagation in PYthon (POPPY)

POPPY is a Python package that simulates physical optical propagation including diffraction. It implements a flexible framework for modeling Fraunhofer (far-field) diffraction and point spread function formation, particularly in the context of astronomical telescopes. POPPY was developed as part of a simulation package for JWST, but is more broadly applicable to many kinds of imaging simulations.

Developed by Marshall Perrin and colleagues at STScI, for use simulating the James Webb Space Telescope and other NASA missions.

Documentation can be found online at https://poppy-optics.readthedocs.io/

12.1.1 Functions

display_psf(HDUlist_or_filename[, ext, ...])	Display nicely a PSF from a given hdulist or filename
display_psf_difference([...])	Display nicely the difference of two PSFs from given files
display_ee([HDUlist_or_filename, ext, ...])	Display Encircled Energy curve for a PSF
measure_ee([HDUlist_or_filename, ext, ...])	measure encircled energy vs radius and return as an interpolator
measure_radius_at_ee([HDUlist_or_filename, ...])	measure encircled energy vs radius and return as an interpolator Returns a function object which when called returns the radius for a given Encircled Energy.
display_profiles([HDUlist_or_filename, ext, ...])	Produce two plots of PSF radial profile and encircled energy
radial_profile([hdulist_or_filename, ext, ...])	Compute a radial profile of the image.
measure_radial([HDUlist_or_filename, ext, ...])	measure azimuthally averaged radial profile of a PSF.

Continued on next page

Table 1 – continued from previous page

measure_fwhm(HDUlist_or_filename[, ext, ...])	Improved version of measuring FWHM, without any binning of image data.
measure_sharpness([HDUlist_or_filename, ext])	Compute image sharpness, the sum of pixel squares.
measure_centroid([HDUlist_or_filename, ext, ...])	Measure the center of an image via center-of-mass
measure_strehl([HDUlist_or_filename, ext, ...])	Measure Strehl for a PSF
measure_anisotropy([HDUlist_or_filename, ...])	
specFromSpectralType(sptype[, return_list, ...])	Get Pysynphot Spectrum object from a user-friendly spectral type string.

display_psf

poppy.**display_psf**(*HDUlist_or_filename*, *ext=0*, *vmin=1e-07*, *vmax=0.1*, *scale='log'*, *cmap=None*, *title=None*, *imagecrop=None*, *adjust_for_oversampling=False*, *normalize='None'*, *crosshairs=False*, *markcentroid=False*, *colorbar=True*, *colorbar_orientation='vertical'*, *pixelscale='PIXELSCL'*, *ax=None*, *return_ax=False*, *interpolation=None*, *cube_slice=None*)

Display nicely a PSF from a given hdulist or filename

This is extensively configurable. In addition to making an attractive display, for interactive usage this function provides a live display of the pixel value at a given (x,y) as you mouse around the image.

HDUlist_or_filename
[fits.hdulist or string] FITS file containing image to display.

ext
[int] FITS extension. default = 0

vmin, vmax
[float] min and max for image display scaling

scale
[str] 'linear' or 'log', default is log

cmap
[matplotlib.cm.Colormap instance or None] Colormap to use. If not given, taken from user's `poppy.conf.cmap_sequential` (Default: 'gist_heat').

title
[string, optional] Set the plot title explicitly.

imagecrop
[float] size of region to display (default is whole image)

adjust_for_oversampling
[bool] rescale to conserve surface brightness for oversampled PSFs? (Making this True conserves surface brightness but not total flux.) Default is False, to conserve total flux.

normalize
[string] set to 'peak' to normalize peak intensity =1, or to 'total' to normalize total flux=1. Default is no normalization.

crosshairs
[bool] Draw a crosshairs at the image center (0, 0)? Default: False.

markcentroid
[bool] Draw a crosshairs at the image centroid location? Centroiding is computed with the JWST-standard moving box algorithm. Default: False.

colorbar
[bool] Draw a colorbar on the image?

colorbar_orientation
['vertical' (default) or 'horizontal'] How should the colorbar be oriented? (Note: Updating a plot and changing the colorbar orientation is not supported. When replotting in the same axes, use the same colorbar orientation.)

pixelscale
[str or float] if str, interpreted as the FITS keyword name for the pixel scale in arcsec/pixels. if float, used as the pixelscale directly.

ax
[matplotlib.Axes instance] Axes to display into.

return_ax
[bool] Return the axes to the caller for later use? (Default: False) When True, this function returns a matplotlib.Axes instance, or a tuple of (ax, cb) where the second is the colorbar Axes.

interpolation
[string] Interpolation technique for PSF image. Default is None, meaning it is taken from matplotlib's `image.interpolation` rcParam.

cube_slice
[int or None] if input PSF is a datacube from calc_datacube, which slice of the cube should be displayed?

display_psf_difference

poppy.**display_psf_difference**(*hdulist_or_filename1=None*, *HDUlist_or_filename2=None*, *ext1=0*, *ext2=0*, *vmin=None*, *vmax=0.0001*, *title=None*, *imagecrop=None*, *adjust_for_oversampling=False*, *crosshairs=False*, *cmap=None*, *colorbar=True*, *colorbar_orientation='vertical'*, *print_=False*, *ax=None*, *return_ax=False*, *normalize=False*, *normalize_to_second=False*)
Display nicely the difference of two PSFs from given files

The two files may be FITS files on disk or FITS HDUList objects in memory. The two must have the same shape and size.

hdulist_or_filename1, HDUlist_or_filename2
[fits.HDUlist or string] FITS files containing images to difference

ext1, ext2
[int] FITS extension. default = 0

vmin, vmax
[float] Image intensity scaling min and max.

title
[string, optional] Title for plot.

imagecrop
[float] Size of region to display (default is whole image).

adjust_for_oversampling
[bool] Rescale to conserve surface brightness for oversampled PSFs? (Making this True conserves surface brightness but not total flux.) Default is False, to conserve total flux.

crosshairs
[bool] Plot crosshairs over array center?

cmap
[matplotlib.cm.Colormap instance or None] Colormap to use. If not given, use standard gray colormap.

colorbar
[bool] Draw a colorbar on the image?

colorbar_orientation
['vertical' (default) or 'horizontal'] How should the colorbar be oriented? (Note: Updating a plot and changing the colorbar orientation is not supported. When replotting in the same axes, use the same colorbar orientation.)

print_
[bool] Print RMS difference value for the images? (Default: False)

ax
[matplotlib.Axes instance] Axes to display into.

return_ax
[bool] Return the axes to the caller for later use? (Default: False) When True, this function returns a matplotlib.Axes instance, or a tuple of (ax, cb) where the second is the colorbar Axes.

normalize
[bool] Display (difference image)/(mean image) instead of just the difference image. Mutually exclusive to `normalize_to_second`. (Default: False)

normalize_to_second
[bool] Display (difference image)/(second image) instead of just the difference image. Mutually exclusive to `normalize`. (Default: False)

display_ee

poppy.**display_ee**(*HDUlist_or_filename=None*, *ext=0*, *overplot=False*, *ax=None*, *mark_levels=True*, ***kwargs*)
Display Encircled Energy curve for a PSF

The azimuthally averaged encircled energy is plotted as a function of radius.

HDUlist_or_filename
[fits.HDUlist or string] FITS file containing image to display encircled energy for.

ext
[bool] FITS extension to use. Default is 0

overplot
[bool] whether to overplot or clear and produce an new plot. Default false

ax
[matplotlib Axes instance] axis to plot into. If not provided, current axis will be used.

mark_levels
[bool] If set, mark and label on the plots the radii for 50%, 80%, 95% encircled energy. Default is True

measure_ee

poppy.**measure_ee**(*HDUlist_or_filename=None*, *ext=0*, *center=None*, *binsize=None*)
measure encircled energy vs radius and return as an interpolator

Returns a function object which when called returns the Encircled Energy inside a given radius, for any arbitrary desired radius smaller than the image size.

HDUlist_or_filename
[string] Either a fits.HDUList object or a filename of a FITS file on disk

ext
[int] Extension in that FITS file

center
[tuple of floats] Coordinates (x,y) of PSF center. Default is image center.

binsize:
size of step for profile. Default is pixel size.

encircled_energy: function
A function which will return the encircled energy interpolated to any desired radius.

```
>>> ee = measure_ee("someimage.fits")
>>> print "The EE at 0.5 arcsec is ", ee(0.5)
```

measure_radius_at_ee

poppy.**measure_radius_at_ee**(*HDUlist_or_filename=None*, *ext=0*, *center=None*, *binsize=None*)
measure encircled energy vs radius and return as an interpolator Returns a function object which when called returns the radius for a given Encircled Energy. This is the inverse function of measure_ee

HDUlist_or_filename
[string] Either a fits.HDUList object or a filename of a FITS file on disk

ext
[int] Extension in that FITS file

center
[tuple of floats] Coordinates (x,y) of PSF center. Default is image center.

binsize:
size of step for profile. Default is pixel size.

radius: function
A function which will return the radius of a desired encircled energy.

```
>>> ee = measure_radius_at_ee("someimage.fits")
>>> print "The EE is 50% at {} arcsec".format(ee(0.5))
```

display_profiles

poppy.**display_profiles**(*HDUlist_or_filename=None*, *ext=0*, *overplot=False*, *title=None*, ***kwargs*)
Produce two plots of PSF radial profile and encircled energy

See also the display_ee function.

HDUlist_or_filename1,2
[fits.HDUlist or string] FITS files containing image to difference

ext
[bool] FITS extension to use. Default is 0

overplot
[bool] whether to overplot or clear and produce an new plot. Default false

title
[string, optional] Title for plot

radial_profile

poppy.**radial_profile**(*hdulist_or_filename=None*, *ext=0*, *ee=False*, *center=None*, *stddev=False*, *binsize=None*, *maxradius=None*, *normalize='None'*, *pa_range=None*)
Compute a radial profile of the image.

This computes a discrete radial profile evaluated on the provided binsize. For a version interpolated onto a continuous curve, see measure_radial().

Code taken pretty much directly from pydatatut.pdf

hdulist_or_filename
[string] FITS HDUList object or path to a FITS file. NaN values in the FITS data array are treated as masked and ignored in computing bin statistics.

ext
[int] Extension in FITS file

ee
[bool] Also return encircled energy (EE) curve in addition to radial profile?

center
[tuple of floats] Coordinates (x,y) of PSF center, in pixel units. Default is image center.

binsize
[float] size of step for profile. Default is pixel size.

stddev
[bool] Compute standard deviation in each radial bin, not average?

normalize
[string] set to 'peak' to normalize peak intensity =1, or to 'total' to normalize total flux=1. Default is no normalization.

pa_range
[list of floats, optional] Optional specification for [min, max] position angles to be included in the radial profile. I.e. calculate that profile only for some wedge, not the full image. Specify the PA in degrees counterclockwise from +Y axis=0. Note that you can specify ranges across zero using negative numbers, such as pa_range=[-10,10]. The allowed PA range runs from -180 to 180 degrees.

results
[tuple] Tuple containing (radius, profile) or (radius, profile, EE) depending on what is requested. The radius gives the center radius of each bin, while the EE is given inside the whole bin so you should use (radius+binsize/2) for the radius of the EE curve if you want to be as precise as possible.

measure_radial

poppy.**measure_radial**(*HDUlist_or_filename=None*, *ext=0*, *center=None*, *binsize=None*)
measure azimuthally averaged radial profile of a PSF.

Returns a function object which when called returns the mean value at a given radius.

HDUlist_or_filename
[string] what it sounds like.

ext
[int] Extension in FITS file

center
[tuple of floats] Coordinates (x,y) of PSF center. Default is image center.

binsize:
size of step for profile. Default is pixel size.

radial_profile: function
A function which will return the mean PSF value at any desired radius.

```
>>> rp = measure_radial("someimage.fits")
>>> radius = np.linspace(0, 5.0, 100)
>>> plot(radius, rp(radius), label="PSF")
```

measure_fwhm

poppy.**measure_fwhm**(*HDUlist_or_filename*, *ext=0*, *center=None*, *plot=False*, *threshold=0.1*)
Improved version of measuring FWHM, without any binning of image data.

Method: Pick out the image pixels which are above some threshold relative to the peak intensity, then fit a Gaussian to those. Infer the FWHM based on the width of the Gaussian.

HDUlist_or_filename
[string] what it sounds like.

ext
[int] Extension in FITS file

center
[tuple of floats] Coordinates (x,y) of PSF center, in pixel units. Default is image center.

threshold
[float] Fraction relative to the peak pixel that is used to select the bright peak pixels used in fitting the Gaussian. Default is 0.1, i.e. pixels brighter that 0.1 of the maximum will be included. This is chosen semi-arbitrarily to include most of the peak but exclude the first Airy ring for typical cases.

plot
[bool] Display a diagnostic plot.

fwhm
[float] FWHM in arcseconds

measure_sharpness

poppy.**measure_sharpness**(*HDUlist_or_filename=None*, *ext=0*)
Compute image sharpness, the sum of pixel squares.

See Makidon et al. JWST-STScI-001157 for a discussion of this image metric and its relationship to noise equivalent pixels.

HDUlist_or_filename, ext
[string, int] Same as above

measure_centroid

poppy.**measure_centroid**(*HDUlist_or_filename=None*, *ext=0*, *slice=0*, *boxsize=20*, *verbose=False*, *units='pixels'*, *relativeto='origin'*, ***kwargs*)

Measure the center of an image via center-of-mass

The centroid method used is the floating-box center of mass algorithm by Jeff Valenti et al., which has been adopted for JWST target acquisition measurements on orbit. See JWST technical reports JWST-STScI-001117 and JWST-STScI-001134 for details.

HDUlist_or_filename
[string] Either a fits.HDUList object or a filename of a FITS file on disk

ext
[int] Extension in that FITS file

slice
[int, optional] If that extension is a 3D datacube, which slice (plane) of that datacube to use

boxsize
[int] Half box size for centroid

relativeto
[string] either 'origin' for relative to pixel (0,0) or 'center' for relative to image center. Default is 'origin'

units
[string] either 'pixels' for position in pixels or 'arcsec' for arcseconds. Relative to the relativeto parameter point in either case.

CoM
[array_like] [Y, X] coordinates of center of mass.

measure_strehl

poppy.**measure_strehl**(*HDUlist_or_filename=None*, *ext=0*, *slice=0*, *center=None*, *display=True*, *verbose=True*, *cache_perfect=False*)

Measure Strehl for a PSF

NOTE - deprecated / removed function. Moved to webbpsf package instead.

This stub is just here to provide information on that transfer, and will be removed in a future version of poppy.

measure_anisotropy

poppy.**measure_anisotropy**(*HDUlist_or_filename=None*, *ext=0*, *slice=0*, *boxsize=50*)

specFromSpectralType

poppy.**specFromSpectralType**(*sptype*, *return_list=False*, *catalog=None*)

Get Pysynphot Spectrum object from a user-friendly spectral type string.

Given a spectral type such as 'A0IV' or 'G2V', this uses a fixed lookup table to determine an appropriate spectral model from Castelli & Kurucz 2004 or the Phoenix model grids. Depends on pysynphot and CDBS. This is just a convenient access function.

catalog: str
'ck04' for Castelli & Kurucz 2004, 'phoenix' for Phoenix models. If not set explicitly, the code will check if the phoenix models are present inside the $PYSYN_CDBS directory. If so, those are the default; otherwise, it's CK04.

12.1.2 Classes

Instrument([name])	A generic astronomical instrument, composed of
Wavefront([wavelength, npix, dtype, diam, ...])	Wavefront in the Fraunhofer approximation: a monochromatic wavefront that can be transformed between pupil and image planes only, not to intermediate planes
OpticalSystem([name, verbose, oversample, ...])	A class representing a series of optical elements, either Pupil, Image, or Detector planes, through which light can be propagated.
CompoundOpticalSystem([optsyslist, name])	A concatenation of two or more optical systems, acting as a single larger optical system.
OpticalElement([name, verbose, planetype, ...])	Base class for all optical elements, whether from FITS files or analytic functions.
ArrayOpticalElement([opd, transmission, ...])	Defines an arbitrary optic, based on amplitude transmission and/or OPD given as numpy arrays.
FITSOpticalElement([name, transmission, ...])	Defines an arbitrary optic, based on amplitude transmission and/or OPD FITS files.
Rotation([angle, units, hide])	Performs a rotation of the axes in the optical train.
Detector([pixelscale, fov_pixels, ...])	A Detector is a specialized type of OpticalElement that forces a wavefront onto a specific fixed pixelization of an Image plane.
AnalyticOpticalElement([shift_x, shift_y, ...])	Defines an abstract analytic optical element, i.e.
ScalarTransmission([name, transmission])	Uniform transmission between 0 and 1.0 in intensity.
InverseTransmission([optic])	Given any arbitrary OpticalElement with transmission T(x,y) return the inverse transmission 1 - T(x,y)
BandLimitedCoron	alias of poppy.optics.BandLimitedCoronagraph
BandLimitedCoronagraph([name, kind, sigma, ...])	Defines an ideal band limited coronagraph occulting mask.
IdealFQPM([name, wavelength])	Defines an ideal 4-quadrant phase mask coronagraph, with its retardance set perfectly to 0.5 waves at one specific wavelength and varying linearly on either side of that.
CircularPhaseMask([name, radius, ...])	Circular phase mask coronagraph, with its retardance set perfectly at one specific wavelength and varying linearly on either side of that.
RectangularFieldStop([name, width, height])	Defines an ideal rectangular field stop
SquareFieldStop([name, size])	Defines an ideal square field stop
AnnularFieldStop([name, radius_inner, ...])	Defines a circular field stop with an (optional) opaque circular center region
HexagonFieldStop([name, side, diameter, ...])	Defines an ideal hexagonal field stop
CircularOcculter([name, radius])	Defines an ideal circular occulter (opaque circle)
BarOcculter([name, width, height])	Defines an ideal bar occulter (like in MIRI's Lyot coronagraph)
FQPM_FFT_aligner([name, direction])	Helper class for modeling FQPMs accurately
CircularAperture([name, radius, pad_factor, ...])	Defines an ideal circular pupil aperture

Continued on next page

Table 2 – continued from previous page

HexagonAperture([name, side, diameter, ...])	Defines an ideal hexagonal pupil aperture
MultiHexagonAperture([name, flattoflat, ...])	Defines a hexagonally segmented aperture
NgonAperture([name, nsides, radius, rotation])	Defines an ideal N-gon pupil aperture.
RectangleAperture([name, width, height, ...])	Defines an ideal rectangular pupil aperture
SquareAperture([name, size])	Defines an ideal square pupil aperture
SecondaryObscuration([name, ...])	Defines the central obscuration of an on-axis telescope including secondary mirror and supports
AsymmetricSecondaryObscuration([...])	Defines a central obscuration with one or more supports which can be oriented at arbitrary angles around the primary mirror, a la the three supports of JWST
ThinLens([name, nwaves, ...])	An idealized thin lens, implemented as a Zernike defocus term.
GaussianAperture([name, fwhm, w, pupil_diam])	Defines an ideal Gaussian apodized pupil aperture, or at least as much of one as can be fit into a finite-sized array
CompoundAnalyticOptic([opticslist, name, ...])	Define a compound analytic optical element made up of the combination of two or more individual optical elements.
QuadPhase([z, planetype, name])	Quadratic phase factor, q(z) suitable for representing a radially-dependent wavefront curvature.
QuadraticLens([f_lens, planetype, name])	Gaussian Lens
FresnelWavefront(beam_radius[, units, ...])	Wavefront for Fresnel diffraction calculation.
FresnelOpticalSystem([name, pupil_diameter, ...])	Class representing a series of optical elements, through which light can be propagated using the Fresnel formalism.
WavefrontError(**kwargs)	A base class for different sources of wavefront error
ParameterizedWFE([name, coefficients, ...])	Define an optical element in terms of its distortion as decomposed into a set of orthonormal basis functions (e.g.
ZernikeWFE([name, coefficients, radius, ...])	Define an optical element in terms of its Zernike components by providing coefficients for each Zernike term contributing to the analytic optical element.
SineWaveWFE([name, spatialfreq, amplitude, ...])	A single sine wave ripple across the optic

Instrument

class poppy.**Instrument**(*name=''*, **args*, ***kwargs*)

Bases: object

A generic astronomical instrument, composed of

(1) an optical system implemented using POPPY, optionally with several configurations such as selectable image plane or pupil plane stops, and

(2) some defined spectral bandpass(es) such as selectable filters, implemented using pysynphot.

This provides the capability to model both the optical and spectral responses of a given system. PSFs may be calculated for given source spectral energy distributions and output as FITS files, with substantial flexibility.

It also provides capabilities for modeling some PSF effects not due to wavefront aberrations, for instance blurring caused by pointing jitter.

This is a base class for Instrument functionality - you cannot easily use this directly, but rather should subclass it for your particular instrument of interest. Some of the complexity of this class is due to splitting up functionality

into many separate routines to allow users to subclass just the relevant portions for a given task. There's a fair amount of functionality here but the learning curve is steeper than elsewhere in POPPY.

You will at a minimum want to override the following class methods:

- _get_optical_system
- _get_filter_list
- _get_default_nlambda
- _get_default_fov
- _get_fits_header

For more complicated systems you may also want to override:

- _validate_config
- _get_synphot_bandpass
- _apply_jitter

Attributes Summary

filter	Currently selected filter name (e.g.
filter_list	List of available filter names for this instrument
name	
options	A dictionary capable of storing other arbitrary options, for extensibility.
pixelscale	Detector pixel scale, in arcseconds/pixel (default: 0.025)
pupil	Aperture for this optical system.
pupilopd	Pupil OPD for this optical system.

Methods Summary

calc_datacube(wavelengths, *args, **kwargs)	Calculate a spectral datacube of PSFs
calc_psf([outfile, source, nlambda, ...])	Compute a PSF.
display()	Display the currently configured optical system on screen

Attributes Documentation

filter
Currently selected filter name (e.g. F200W)

filter_list = None
List of available filter names for this instrument

name = 'Instrument'

options = {}
A dictionary capable of storing other arbitrary options, for extensibility. The following are all optional, and may or may not be meaningful depending on which instrument is selected.

source_offset_r
[float] Radial offset of the target from the center, in arcseconds

source_offset_theta
[float] Position angle for that offset

pupil_shift_x, pupil_shift_y
[float] Relative shift of a coronagraphic pupil in X and Y, expressed as a decimal between 0.0-1.0 Note that shifting an array too much will wrap around to the other side unphysically, but for reasonable values of shift this is a non-issue.

jitter
[string "gaussian" or None] Type of jitter model to apply. Currently only convolution with a Gaussian kernel of specified width `jitter_sigma` is implemented. (default: None)

jitter_sigma
[float] Width of the jitter kernel in arcseconds (default: 0.007 arcsec)

parity
[string "even" or "odd"] You may wish to ensure that the output PSF grid has either an odd or even number of pixels. Setting this option will force that to be the case by increasing npix by one if necessary.

`pixelscale = 0.025`
Detector pixel scale, in arcseconds/pixel (default: 0.025)

`pupil = None`
Aperture for this optical system. May be a FITS filename, FITS HDUList object, or poppy.OpticalElement

`pupilopd = None`
Pupil OPD for this optical system. May be a FITS filename, or FITS HDUList. If the file contains a datacube, you may set this to a tuple (filename, slice) to select a given slice, or else the first slice will be used.

Methods Documentation

`calc_datacube`(*wavelengths*, **args*, ***kwargs*)
Calculate a spectral datacube of PSFs

wavelengths
[iterable of floats] List or ndarray or tuple of floating point wavelengths in meters, such as you would supply in a call to calc_psf via the "monochromatic" option

`calc_psf`(*outfile=None*, *source=None*, *nlambda=None*, *monochromatic=None*, *fov_arcsec=None*, *fov_pixels=None*, *oversample=None*, *detector_oversample=None*, *fft_oversample=None*, *overwrite=True*, *display=False*, *save_intermediates=False*, *return_intermediates=False*, *normalize='first'*)
Compute a PSF. The result can either be written to disk (set outfile="filename") or else will be returned as a FITS HDUlist object.

Output sampling may be specified in one of two ways:

1) Set `oversample=`. This will use that oversampling factor beyond detector pixels for output images, and beyond Nyquist sampling for any FFTs to prior optical planes.

2) set `detector_oversample=` and `fft_oversample=`. This syntax lets you specify distinct oversampling factors for intermediate and final planes.

By default, both oversampling factors are set equal to 2.

More advanced PSF computation options (pupil shifts, source positions, jitter, ...) may be set by configuring the `options` dictionary attribute of this class.

source
[pysynphot.SourceSpectrum or dict] specification of source input spectrum. Default is a 5700 K sunlike star.

nlambda
[int] How many wavelengths to model for broadband? The default depends on how wide the filter is: (5,3,1) for types (W,M,N) respectively

monochromatic
[float, optional] Setting this to a wavelength value (in meters) will compute a monochromatic PSF at that wavelength, overriding filter and nlambda settings.

fov_arcsec
[float] field of view in arcsec. Default=5

fov_pixels
[int] field of view in pixels. This is an alternative to fov_arcsec.

outfile
[string] Filename to write. If None, then result is returned as an HDUList

oversample, detector_oversample, fft_oversample
[int] How much to oversample. Default=4. By default the same factor is used for final output pixels and intermediate optical planes, but you may optionally use different factors if so desired.

overwrite
[bool] overwrite output FITS file if it already exists?

display
[bool] Whether to display the PSF when done or not.

save_intermediates, return_intermediates
[bool] Options for saving to disk or returning to the calling function the intermediate optical planes during the propagation. This is useful if you want to e.g. examine the intensity in the Lyot plane for a coronagraphic propagation.

normalize
[string] Desired normalization for output PSFs. See doc string for OpticalSystem.calc_psf. Default is to normalize the entrance pupil to have integrated total intensity = 1.

outfits
[fits.HDUList] The output PSF is returned as a fits.HDUlist object. If `outfile` is set to a valid filename, the output is also written to that file.

`display()`
Display the currently configured optical system on screen

Wavefront

class poppy.**Wavefront**(*wavelength=<Quantity 1.e-06 m>*, *npix=1024*, *dtype=None*, *diam=<Quantity 8. m>*, *oversample=2*, *pixelscale=None*)
Bases: `poppy.poppy_core.BaseWavefront`

Wavefront in the Fraunhofer approximation: a monochromatic wavefront that can be transformed between pupil and image planes only, not to intermediate planes

In a pupil plane, a wavefront object `wf` has

- `wf.diam`, a diameter in meters
- `wf.pixelscale`, a scale in meters/pixel

In an image plane, it has

- `wf.fov`, a field of view in arcseconds
- `wf.pixelscale`, a scale in arcsec/pixel

Use the `wf.propagate_to()` method to transform a wavefront between conjugate planes. This will update those properties as appropriate.

By default, `Wavefronts` are created in a pupil plane. Set `pixelscale=#` to make an image plane instead.

wavelength
[float] Wavelength of light in meters

npix
[int] Size parameter for wavefront array to create, per side.

diam
[float, optional] For _PUPIL wavefronts, sets physical size corresponding to npix. Units are meters. At most one of diam or pixelscale should be set when creating a wavefront.

pixelscale
[float, optional] For PlaneType.image PLANE wavefronts, use this pixel scale.

oversample
[int, optional] how much to oversample by in FFTs. Default is 2. Note that final propagations to Detectors use a different algorithm and, optionally, a separate oversampling factor.

dtype
[numpy.dtype, optional] default is double complex.

Methods Summary

`coordinates`()	Return Y, X coordinates for this wavefront, in the manner of numpy.indices()
`from_fresnel_wavefront`(fresnel_wavefront[, ...])	Convert a Fresnel type wavefront object to a Fraunhofer one
`image_coordinates`(shape, pixelscale, ...)	Utility function to generate coordinates arrays for an image plane wavefront
`propagate_to`(optic)	Propagates a wavefront object to the next optic in the list.
`pupil_coordinates`(shape, pixelscale)	Utility function to generate coordinates arrays for a pupil plane wavefront

Methods Documentation

`coordinates`()
Return Y, X coordinates for this wavefront, in the manner of numpy.indices()

This function knows about the offset resulting from FFTs. Use it whenever computing anything measured in wavefront coordinates.

Y, X
[array_like] Wavefront coordinates in either meters or arcseconds for pupil and image, respectively

classmethod from_fresnel_wavefront(*fresnel_wavefront*, *verbose=False*)
Convert a Fresnel type wavefront object to a Fraunhofer one

Note, this function implicitly assumes this wavefront is at a pupil plane, so the resulting Fraunhofer wavefront will have pixelscale in meters/pix rather than arcsec/pix.

fresnel_wavefront
[Wavefront] The (Fresnel-type) wavefront to be converted.

static image_coordinates(*shape*, *pixelscale*, *last_transform_type*, *image_centered*)
Utility function to generate coordinates arrays for an image plane wavefront

shape
[tuple of ints] Shape of the wavefront array

pixelscale
[float or 2-tuple of floats] the pixelscale in meters/pixel, optionally different in X and Y

last_transform_type
[string] Was the last transformation on the Wavefront an FFT or an MFT?

image_centered
[string] Was POPPY trying to keeping the center of the image on a pixel, crosshairs ('array_center'), or corner?

propagate_to(*optic*)
Propagates a wavefront object to the next optic in the list. Modifies this wavefront object itself.

Transformations between pupil and detector planes use MFT or inverse MFT. Transformations between pupil and other (non-detector) image planes use FFT or inverse FFT, unless explicitly tagged to use MFT via a propagation hint. Transformations from any frame through a rotation or coordinate transform plane simply transform the wavefront accordingly.

optic
[OpticalElement] The optic to propagate to. Used for determining the appropriate optical plane.

static pupil_coordinates(*shape*, *pixelscale*)
Utility function to generate coordinates arrays for a pupil plane wavefront

shape
[tuple of ints] Shape of the wavefront array

pixelscale
[float or 2-tuple of floats] the pixel scale in meters/pixel, optionally different in X and Y

OpticalSystem

class poppy.**OpticalSystem**(*name='unnamed system'*, *verbose=True*, *oversample=2*, *npix=None*, *pupil_diameter=None*)
Bases: poppy.poppy_core.BaseOpticalSystem

A class representing a series of optical elements, either Pupil, Image, or Detector planes, through which light can be propagated.

The difference between Image and Detector planes is that Detectors have fixed pixels in terms of arcsec/pixel regardless of wavelength (computed via MFT) while Image planes have variable pixels scaled in terms of lambda/D. Pupil planes are some fixed size in meters, of course.

name
[string] descriptive name of optical system

oversample
[int] Either how many times *above* Nyquist we should be (for pupil or image planes), or how many times a fixed detector pixel will be sampled. E.g. `oversample=2` means image plane sampling lambda/4*D (twice Nyquist) and detector plane sampling 2x2 computed pixels per real detector pixel. Default is 2.

verbose
[bool] whether to be more verbose with log output while computing

pupil_diameter
[astropy.Quantity of dimension length] Diameter of entrance pupil. Defaults to size of first optical element if unspecified, or else 1 meter.

Methods Summary

`add_image`([optic, function, index])	Add an image plane optic to the optical system
`add_pupil`([optic, function, index])	Add a pupil plane optic from file(s) giving transmission or OPD
`describe`()	Print out a string table describing all planes in an optical system
`input_wavefront`([wavelength])	Create a Wavefront object suitable for sending through a given optical system, based on the size of the first optical plane, assumed to be a pupil.
`propagate`(wavefront[, normalize, ...])	Core low-level routine for propagating a wavefront through an optical system

Methods Documentation

add_image(*optic=None*, *function=None*, *index=None*, ***kwargs*)
Add an image plane optic to the optical system

That image plane optic can be specified either

1) **from file(s) giving transmission or OPD**
   [set arguments `transmission=filename` and/or `opd=filename`]
2) **from an analytic function**
   [set `function='circle, fieldstop, bandlimitedcoron, or FQPM'` and set additional kwargs to define shape etc.
3) **from an already-created OpticalElement object**
   [set `optic=that object`]

optic
[poppy.OpticalElement] An already-created OpticalElement you would like to add

function: string
Name of some analytic function to add. Optional `kwargs` can be used to set the parameters of that function. Allowable function names are CircularOcculter, fieldstop, BandLimitedCoron, FQPM

opd, transmission
[string] Filenames of FITS files describing the desired optic.

index
[int] Index into the optical system's planes for where to add the new optic. Defaults to appending the optic to the end of the plane list.

poppy.OpticalElement subclass
The pupil optic added (either `optic` passed in, or a new OpticalElement created)

Now you can use the optic argument for either an OpticalElement or a string function name, and it will do the right thing depending on type. Both existing arguments are left for back compatibility for now.

add_pupil(*optic=None*, *function=None*, *index=None*, ***kwargs*)
Add a pupil plane optic from file(s) giving transmission or OPD

1) **from file(s) giving transmission and/or OPD**
[set arguments `transmission=filename` and/or `opd=filename`]

2) **from an already-created `OpticalElement` object**
[set `optic=that object`]

optic
[poppy.OpticalElement, optional] An already-created `OpticalElement` object you would like to add

function
[string, optional] Deprecated. The name of some analytic function you would like to use. Optional `kwargs` can be used to set the parameters of that function. Allowable function names are Circle, Square, Hexagon, Rectangle, and FQPM_FFT_Aligner

opd, transmission
[string, optional] Filenames of FITS files describing the desired optic.

index
[int] Index into the optical system's planes for where to add the new optic. Defaults to appending the optic to the end of the plane list.

poppy.OpticalElement subclass
The pupil optic added (either `optic` passed in, or a new OpticalElement created)

Note: Now you can use the optic argument for either an OpticalElement or a string function name, and it will do the right thing depending on type. Both existing arguments are left for compatibility for now.

Any provided parameters are passed to `OpticalElement`.

describe()
Print out a string table describing all planes in an optical system

input_wavefront(*wavelength=<Quantity 1.e-06 m>*)
Create a Wavefront object suitable for sending through a given optical system, based on the size of the first optical plane, assumed to be a pupil.

Defining this needs both a number of pixels (npix) and physical size (diam) to set the sampling.

If this OpticalSystem has a provided `npix` attribute that is not None, use that to set the input wavefront size. Otherwise, check if the first optical element has a defined sampling. If not, default to 1024 pixels.

Uses self.source_offset to assign an off-axis tilt, if requested.

The convention here is that the desired source position is specified with respect to the **final focal plane** of the optical system. If there are any intervening coordinate transformation planes, this function attempts to take them into account when setting the tilt of the input wavefront. This is subtle trickery and may not work properly in all instances.

wavelength
[float] Wavelength in meters

wavefront
[poppy.Wavefront instance] A wavefront appropriate for passing through this optical system.

propagate(*wavefront*, *normalize='none'*, *return_intermediates=False*, *display_intermediates=False*)
Core low-level routine for propagating a wavefront through an optical system

This is a **linear operator** that acts on an input complex wavefront to give an output complex wavefront.

wavefront
[Wavefront instance] Wavefront to propagate through this optical system

normalize
[string] How to normalize the wavefront? * 'first' = set total flux = 1 after the first optic, presumably a pupil * 'last' = set total flux = 1 after the entire optical system. * 'exit_pupil' = set total flux = 1 at the last pupil of the optical system.

display_intermediates
[bool] Should intermediate steps in the calculation be displayed on screen? Default: False.

return_intermediates
[bool] Should intermediate steps in the calculation be returned? Default: False. If True, the second return value of the method will be a list of `poppy.Wavefront` objects representing intermediate optical planes from the calculation.

Returns a wavefront, and optionally also the intermediate wavefronts after each step of propagation.

CompoundOpticalSystem

class poppy.**CompoundOpticalSystem**(*optsyslist=None*, *name=None*, ***kwargs*)
Bases: `poppy.poppy_core.OpticalSystem`

A concatenation of two or more optical systems, acting as a single larger optical system.

This can be used to combine together multiple existing OpticalSystem instances, including mixed lists of both Fraunhofer and Fresnel type systems.

Create combined optical system,

optsyslist : List of OpticalSystem and/or FresnelOpticalSystem instances.

Methods Summary

`input_wavefront`([wavelength])	Create input wavefront for a CompoundOpticalSystem
`propagate`(wavefront[, normalize, ...])	Core low-level routine for propagating a wavefront through an optical system

Methods Documentation

input_wavefront(*wavelength=<Quantity 1.e-06 m>*)
Create input wavefront for a CompoundOpticalSystem

Input wavefronts for a compound system are defined by the first OpticalSystem in the list. We tweak the _display_hint_expected_planes to reflect the full compound system however.

propagate(*wavefront*, *normalize='none'*, *return_intermediates=False*, *display_intermediates=False*)
Core low-level routine for propagating a wavefront through an optical system

See docstring of OpticalSystem.propagate for details

OpticalElement

class poppy.**OpticalElement**(*name='unnamed optic'*, *verbose=True*, *planetype=<PlaneType.unspecified: 0>*, *oversample=1*, *interp_order=3*)

Bases: object

Base class for all optical elements, whether from FITS files or analytic functions.

If instantiated on its own, this just produces a null optical element (empty space, i.e. an identity function on transmitted wavefronts.) Use one of the many subclasses to create a nontrivial optic.

The OpticalElement class follows the behavoior of the Wavefront class, using units of meters/pixel in pupil space and arcsec/pixel in image space.

The internal implementation of this class represents an optic with an array for the electric field amplitude transmissivity (or reflectivity), plus an array for the optical path difference in units of meters. This representation was chosen since most typical optics of interest will have wavefront error properties that are independent of wavelength. Subclasses particularly the AnalyticOpticalElements extend this paradigm with optics that have wavelength-dependent properties.

The get_phasor() function is used to obtain the complex phasor for any desired wavelength based on the amplitude and opd arrays. Those can individually be obtained from the get_transmission() and get_opd() functions.

name
[string] descriptive name for optic

verbose
[bool] whether to be more verbose in log outputs while computing

planetype
[int] either poppy.PlaneType.image or poppy.PlaneType.pupil

oversample
[int] how much to oversample beyond Nyquist.

interp_order
[int] the order (0 to 5) of the spline interpolation used if the optic is resized.

Attributes Summary

shape	Return shape of the OpticalElement, as a tuple

Methods Summary

display([nrows, row, what, crosshairs, ax, ...])	Display plots showing an optic's transmission and OPD.
get_opd(wave)	Return the optical path difference, given a wavelength.
get_phasor(wave)	Compute a complex phasor from an OPD, given a wavelength.
get_transmission(wave)	Return the electric field amplitude transmission, given a wavelength.

Attributes Documentation

shape
Return shape of the OpticalElement, as a tuple

Methods Documentation

display(*nrows=1*, *row=1*, *what='intensity'*, *crosshairs=False*, *ax=None*, *colorbar=True*, *colorbar_orientation=None*, *title=None*, *opd_vmax=<Quantity 5.e-07 m>*)
Display plots showing an optic's transmission and OPD.

what
[str] What to display: 'intensity', 'amplitude', 'phase', 'opd', or 'both' (meaning intensity and OPD in two subplots)

ax
[matplotlib.Axes instance] Axes to display into

nrows, row
[integers] number of rows and row index for subplot display

crosshairs
[bool] Display crosshairs indicating the center?

colorbar
[bool] Show colorbar?

colorbar_orientation
[bool] Desired orientation, horizontal or vertical? Default is horizontal if only 1 row of plots, else vertical

opd_vmax
[float] Max absolute value for OPD image display, in meters.

title
[string] Plot label

get_opd(*wave*)
Return the optical path difference, given a wavelength.

wave
[float or obj] either a scalar wavelength or a Wavefront object

ndarray giving OPD in meters

get_phasor(*wave*)
Compute a complex phasor from an OPD, given a wavelength.

The returned value should be the complex phasor array as appropriate for multiplying by the wavefront amplitude.

wave
[float or obj] either a scalar wavelength or a Wavefront object

get_transmission(*wave*)
Return the electric field amplitude transmission, given a wavelength.

wave
[float or obj] either a scalar wavelength or a Wavefront object

ndarray giving electric field amplitude transmission between 0 - 1.0

ArrayOpticalElement

class poppy.**ArrayOpticalElement**(*opd=None*, *transmission=None*, *pixelscale=None*, ***kwargs*)
Bases: poppy.poppy_core.OpticalElement

Defines an arbitrary optic, based on amplitude transmission and/or OPD given as numpy arrays.

This is a very lightweight wrapper for the base OpticalElement class, which just provides some additional convenience features in the initializer..

FITSOpticalElement

class poppy.**FITSOpticalElement**(*name='unnamed optic'*, *transmission=None*, *opd=None*, *opdunits=None*, *rotation=None*, *pixelscale=None*, *planetype=None*, *transmission_index=None*, *opd_index=None*, *shift=None*, *shift_x=None*, *shift_y=None*, *flip_x=False*, *flip_y=False*, ***kwargs*)
Bases: poppy.poppy_core.OpticalElement

Defines an arbitrary optic, based on amplitude transmission and/or OPD FITS files.

This optic could be a pupil or field stop, an aberrated mirror, a phase mask, etc. The FITSOpticalElement class follows the behavior of the Wavefront class, using units of meters/pixel in pupil space and arcsec/pixel in image space.

The interface is **very** flexible. You can define a FITSOpticalElement either from

- a single FITS file giving the amplitude transmission (in which case phase is zero)
- a single FITS file giving the OPD (in which case transmission is 1 everywhere)
- two FITS files specifying both transmission and OPD.

The FITS file argument(s) can be supplied either as

1. a string giving the path to a file on disk,
2. a FITS HDUlist object, or
3. in the case of OPDs, a tuple consisting of a path to a datacube and an integer index of a slice in that datacube.

A better interface for slice selection in datacubes is the transmission_index and opd_index keyword parameters listed below, but the tuple interface is retained for back compatibility with existing code.

name
[string] descriptive name for optic

transmission, opd
[string or fits HDUList] Either FITS filenames *or* actual fits.HDUList objects for the transmission (from 0-1) and opd (in meters)

transmission_slice, opd_slice
[integers, optional] If either transmission or OPD files are datacubes, you can specify the slice index using this argument.

opdunits
[string] units for the OPD file. Default is 'meters'. can be 'meter', 'meters', 'micron(s)', 'nanometer(s)', or their SI abbreviations. If this keyword is not set explicitly, the BUNIT keyword in the FITS header will be checked.

planetype
[int] either PlaneType.image or PlaneType.pupil

oversample
[int] how much to oversample beyond Nyquist.

flip_x, flip_y
[bool] Should the FITS file be inverted in either of these axes after being loaded? Useful for matching coordinate system orientations. If a flip is specified, it takes place prior to any shift or rotation operations.

shift
[tuple of floats, optional] 2-tuple containing X and Y fractional shifts for the pupil. These shifts are implemented by rounding them to the nearest integer pixel, and doing integer pixel shifts on the data array, without interpolation. If a shift is specified, it takes place after any rotation operations.

shift_x, shift_y
[floats, optional] Alternate way of specifying shifts, given in meters of shift per each axis. This is consistent with how AnalyticOpticalElement classes specify shifts. If a shift is specified, it takes place after any rotation operations. If both shift and shift_x/shift_y are specified, an error is raised.

rotation
[float] Rotation for that optic, in degrees counterclockwise. This is implemented using spline interpolation via the scipy.ndimage.interpolation.rotate function.

pixelscale
[optical str or float] By default, poppy will attempt to determine the appropriate pixel scale by examining the FITS header, checking keywords "PUPLSCAL" and 'PIXSCALE' for pupil and image planes respectively. If you would like to override and use a different keyword, provide that as a string here. Alternatively, you can just set a floating point value directly too (in meters/pixel or arcsec/pixel, respectively, for pupil or image planes).

transmission_index, opd_index
[ints, optional] If the input transmission or OPD files are datacubes, provide a scalar index here for which cube slice should be used.

NOTE: All mask files must be *squares*.

Also, please note that the adopted convention is for the spectral throughput (transmission) to be given in appropriate units for acting on the *amplitude* of the electric field. Thus for example an optic with a uniform transmission of 0.5 will reduce the electric field amplitude to 0.5 relative to the input, and thus reduce the total power to 0.25. This distinction only matters in the case of semitransparent (grayscale) masks.

Attributes Summary

pupil_diam	Diameter of the pupil (if this is a pupil plane optic)

Attributes Documentation

pupil_diam
Diameter of the pupil (if this is a pupil plane optic)

Rotation

class poppy.**Rotation**(*angle=0.0*, *units='degrees'*, *hide=False*, ***kwargs*)
Bases: poppy.poppy_core.CoordinateTransform

Performs a rotation of the axes in the optical train.

This is not an actual optic itself, of course, but can be used to model a rotated optic by appling a Rotation before and/or after light is incident on that optic.

This is basically a placeholder to indicate the need for a rotation at a given part of the optical train. The actual rotation computation is performed in the Wavefront object's propagation routines.

angle
[float] Rotation angle, counterclockwise. By default in degrees.

units
['degrees' or 'radians'] Units for the rotation angle.

hide
[bool] Should this optic be displayed or hidden when showing the planes of an OpticalSystem?

Detector

class poppy.**Detector**(*pixelscale=<Quantity 1. arcsec / pix>*, *fov_pixels=None*, *fov_arcsec=None*, *oversample=1*, *name='Detector'*, *offset=None*, ***kwargs*)
Bases: poppy.poppy_core.OpticalElement

A Detector is a specialized type of OpticalElement that forces a wavefront onto a specific fixed pixelization of an Image plane.

This class is in effect just a metadata container for the desired sampling; all the machinery for transformation of a wavefront to that sampling happens within Wavefront.

Note that this is *not* in any way a representation of real noisy detectors; no model for read noise, imperfect sensitivity, etc is included whatsoever.

name
[string] Descriptive name

pixelscale
[float or astropy.units.Quantity] Pixel scale, either in angular units such as arcsec/pixel, or (for Fresnel optical systems only) in physical units such as micron/pixel. Units should be specified as astropy Quantities. If pixelscale is given as a float without an explicit unit, it will be interpreted as in arcsec/pixel. Note, this value may be further subdivided by specifying the oversample parameter > 1.

fov_pixels, fov_arcsec
[float or astropy.units.Quantity] The field of view may be specified either in arcseconds or by a number of pixels. Either is acceptable and the pixel scale is used to convert as needed. You may specify a non-square FOV by providing two elements in an iterable. Note that this follows the usual Python convention of ordering axes (Y,X), so put your desired Y axis size first. For Fresnel optical systems, if specifying pixelscale in microns/pixel then you must specify fov_pixels rather than fov_arcsec.

oversample
[int] Oversampling factor beyond the detector pixel scale. The returned array will have sampling that much finer than the specified pixelscale.

offset
[tuple (X,Y)] Offset for the detector center relative to a hypothetical off-axis PSF. Specifying this lets you pick a different sub-region for the detector to compute, if for some reason you are computing a small subarray around an off-axis source. (Has not been tested!)

Attributes Summary

shape	Return shape of the OpticalElement, as a tuple

Attributes Documentation

shape
Return shape of the OpticalElement, as a tuple

AnalyticOpticalElement

class poppy.**AnalyticOpticalElement**(*shift_x=None*, *shift_y=None*, *rotation=None*, ***kwargs*)
Bases: poppy.poppy_core.OpticalElement

Defines an abstract analytic optical element, i.e. one definable by some formula rather than by an input OPD or pupil file.

This class is useless on its own; instead use its various subclasses that implement appropriate get_opd and/or get_transmission functions. It exists mostly to provide some behaviors & initialization common to all analytic optical elements.

name, verbose, oversample, planetype
[various] Same as for OpticalElement

transmission, opd
[string] These are *not allowed* for Analytic optical elements, and this class will raise an error if you try to set one.

shift_x, shift_y
[Optional floats] Translations of this optic, given in meters relative to the optical axis for pupil plane elements, or arcseconds relative to the optical axis for image plane elements.

rotation
[Optional float] Rotation of the optic around its center, given in degrees counterclockwise. Note that if you apply both shift and rotation, the optic rotates around its own center, rather than the optical axis.

Attributes Summary

shape	Return shape of the OpticalElement, as a tuple

Methods Summary

display([nrows, row, wavelength, npix, ...])	Display an Analytic optic by first computing it onto a grid...
get_coordinates(wave)	Get coordinates of this optic, optionally including shifts
get_opd(wave)	Return the optical path difference, given a wavelength.
get_phasor(wave)	Compute a complex phasor from an OPD, given a wavelength.
get_transmission(wave)	Note that this is the **amplitude** transmission, not the total intensity transmission.

Continued on next page

Table 13 – continued from previous page

sample([wavelength, npix, grid_size, what, ...])	Sample the Analytic Optic onto a grid and return the array
to_fits([outname, what, wavelength, npix])	Save an analytic optic computed onto a grid to a FITS file

Attributes Documentation

shape
Return shape of the OpticalElement, as a tuple

Methods Documentation

display(*nrows=1*, *row=1*, *wavelength=<Quantity 1.e-06 m>*, *npix=512*, *grid_size=None*, *what='intensity'*, ***kwargs*)
Display an Analytic optic by first computing it onto a grid...

wavelength
[float] Wavelength to evaluate this optic's properties at

npix
[int] Number of pixels to use when sampling the analytic optical element.

grid_size
[float] Diameter of the grid on which to sample this optic in meters (for pupil planes) or arcseconds (for image planes)

what
[str] What to display: 'intensity', 'phase', 'opd', or 'both' which shows intensity and phase.

ax
[matplotlib.Axes instance] Axes to display into

nrows, row
[integers] # of rows and row index for subplot display

crosshairs
[bool] Display crosshairs indicating the center?

colorbar
[bool] Show colorbar?

colorbar_orientation
[bool] Desired orientation, horizontal or vertical? Default is horizontal if only 1 row of plots, else vertical

opd_vmax
[float] Max value for OPD image display, in meters.

title
[string] Plot label

get_coordinates(*wave*)
Get coordinates of this optic, optionally including shifts

Method: Calls the supplied wave object's coordinates() method, then checks for the existence of the following attributes: "shift_x", "shift_y", "rotation" If any of them are present, then the coordinates are modified accordingly.

Shifts are given in meters for pupil optics and arcseconds for image optics.

get_opd(*wave*)
Return the optical path difference, given a wavelength.

wave
[float or obj] either a scalar wavelength or a Wavefront object

ndarray giving OPD in meters

get_phasor(*wave*)
Compute a complex phasor from an OPD, given a wavelength.

The returned value should be the complex phasor array as appropriate for multiplying by the wavefront amplitude.

wave
[float or obj] either a scalar wavelength or a Wavefront object

get_transmission(*wave*)
Note that this is the **amplitude** transmission, not the total intensity transmission.

sample(*wavelength=<Quantity 1.e-06 m>*, *npix=512*, *grid_size=None*, *what='amplitude'*, *return_scale=False*, *phase_unit='waves'*)
Sample the Analytic Optic onto a grid and return the array

wavelength
[astropy.units.Quantity or float] Wavelength (in meters if unit not given explicitly)

npix
[integer] Number of pixels for sampling the array

grid_size
[float] Field of view grid size (diameter) for sampling the optic, in meters for pupil plane optics and arcseconds for image planes. Default value is taken from the optic's properties, if defined. Otherwise defaults to 6.5 meters or 2 arcseconds depending on plane.

what
[string] What to return: optic 'amplitude' transmission, 'intensity' transmission, 'phase', or 'opd'. Note that optical path difference, OPD, is given in meters.

phase_unit
[string] Unit for returned phase array IF what=='phase'. One of 'radians', 'waves', 'meters'. ('meters' option is deprecated; use what='opd' instead.)

return_scale
[float] if True, will return a tuple containing the desired array and a float giving the pixel scale.

to_fits(*outname=None*, *what='amplitude'*, *wavelength=<Quantity 1.e-06 m>*, *npix=512*, ***kwargs*)
Save an analytic optic computed onto a grid to a FITS file

The FITS file is returned to the calling function, and may optionally be saved directly to disk.

what
[string] What quantity to save. See the sample function of this class

wavelength
[float] Wavelength in meters.

npix
[integer] Number of pixels.

outname
[string, optional] Filename to write out a FITS file to disk

See the sample() function for additional optional parameters.

ScalarTransmission

class poppy.**ScalarTransmission**(*name=None*, *transmission=1.0*, ***kwargs*)
Bases: poppy.optics.AnalyticOpticalElement

Uniform transmission between 0 and 1.0 in intensity.

Either a null optic (empty plane) or some perfect ND filter... But most commonly this is just used as a null optic placeholder

Methods Summary

get_transmission(wave)	Note that this is the **amplitude** transmission, not the total intensity transmission.

Methods Documentation

get_transmission(*wave*)
Note that this is the **amplitude** transmission, not the total intensity transmission.

InverseTransmission

class poppy.**InverseTransmission**(*optic=None*)
Bases: poppy.optics.AnalyticOpticalElement

Given any arbitrary OpticalElement with transmission T(x,y) return the inverse transmission 1 - T(x,y)

This is a useful ingredient in the SemiAnalyticCoronagraph algorithm.

Attributes Summary

shape	Return shape of the OpticalElement, as a tuple

Methods Summary

display(**kwargs)	Display an Analytic optic by first computing it onto a grid...
get_opd(wave)	Return the optical path difference, given a wavelength.
get_transmission(wave)	Note that this is the **amplitude** transmission, not the total intensity transmission.

Attributes Documentation

shape
Return shape of the OpticalElement, as a tuple

Methods Documentation

display(***kwargs*)
Display an Analytic optic by first computing it onto a grid...

wavelength
[float] Wavelength to evaluate this optic's properties at

npix
[int] Number of pixels to use when sampling the analytic optical element.

grid_size
[float] Diameter of the grid on which to sample this optic in meters (for pupil planes) or arcseconds (for image planes)

what
[str] What to display: 'intensity', 'phase', 'opd', or 'both' which shows intensity and phase.

ax
[matplotlib.Axes instance] Axes to display into

nrows, row
[integers] # of rows and row index for subplot display

crosshairs
[bool] Display crosshairs indicating the center?

colorbar
[bool] Show colorbar?

colorbar_orientation
[bool] Desired orientation, horizontal or vertical? Default is horizontal if only 1 row of plots, else vertical

opd_vmax
[float] Max value for OPD image display, in meters.

title
[string] Plot label

get_opd(*wave*)
Return the optical path difference, given a wavelength.

wave
[float or obj] either a scalar wavelength or a Wavefront object

ndarray giving OPD in meters

get_transmission(*wave*)
Note that this is the **amplitude** transmission, not the total intensity transmission.

BandLimitedCoron

poppy.**BandLimitedCoron**
alias of `poppy.optics.BandLimitedCoronagraph`

BandLimitedCoronagraph

class poppy.**BandLimitedCoronagraph**(*name='unnamed BLC'*, *kind='circular'*, *sigma=1*, *wavelength=None*, ***kwargs*)

Bases: poppy.optics.AnalyticImagePlaneElement

Defines an ideal band limited coronagraph occulting mask.

name
[string] Descriptive name

kind
[string] Either 'circular' or 'linear'. The linear ones are custom shaped to NIRCAM's design with flat bits on either side of the linear tapered bit. Also includes options 'nircamcircular' and 'nircamwedge' specialized for the JWST NIRCam occulters, including the off-axis ND acq spots and the changing width of the wedge occulter.

sigma
[float] The numerical size parameter, as specified in Krist et al. 2009 SPIE

wavelength
[float] Wavelength this BLC is optimized for, only for the linear ones.

Attributes Summary

allowable_kinds	Allowable types of BLC supported by this class

Methods Summary

get_transmission(wave)	Compute the amplitude transmission appropriate for a BLC for some given pixel spacing corresponding to the supplied Wavefront.

Attributes Documentation

allowable_kinds = ['circular', 'linear']
Allowable types of BLC supported by this class

Methods Documentation

get_transmission(*wave*)
Compute the amplitude transmission appropriate for a BLC for some given pixel spacing corresponding to the supplied Wavefront.

Based on the Krist et al. SPIE paper on NIRCam coronagraph design

Note that the equations in Krist et al specify the intensity transmission of the occulter, but what we want to return here is the amplitude transmittance. That is the square root of the intensity, of course, so the equations as implemented here all differ from those written in Krist's SPIE paper by lacking an exponential factor of 2. Thanks to John Krist for pointing this out.

IdealFQPM

class poppy.**IdealFQPM**(*name='unnamed FQPM '*, *wavelength=<Quantity 1.065e-05 m>*, ***kwargs*)
Bases: poppy.optics.AnalyticImagePlaneElement

Defines an ideal 4-quadrant phase mask coronagraph, with its retardance set perfectly to 0.5 waves at one specific wavelength and varying linearly on either side of that. "Ideal" in the sense of ignoring chromatic effects other than just the direct scaling of the wavelength.

name
[string] Descriptive name

wavelength
[float] Wavelength in meters for which the FQPM was designed, and at which there is exactly 1/2 a wave of retardance.

Methods Summary

get_opd(wave)	Compute the OPD appropriate for a 4QPM for some given pixel spacing corresponding to the supplied Wavefront

Methods Documentation

get_opd(*wave*)
Compute the OPD appropriate for a 4QPM for some given pixel spacing corresponding to the supplied Wavefront

CircularPhaseMask

class poppy.**CircularPhaseMask**(*name=None*, *radius=<Quantity 1. arcsec>*, *wavelength=<Quantity 1.e-06 m>*, *retardance=0.5*, ***kwargs*)
Bases: poppy.optics.AnalyticImagePlaneElement

Circular phase mask coronagraph, with its retardance set perfectly at one specific wavelength and varying linearly on either side of that.

name
[string] Descriptive name

radius
[float] Radius of the mask

wavelength
[float] Wavelength in meters for which the phase mask was designed

retardance
[float] Optical path delay at that wavelength, specified in waves relative to the reference wavelengt. Default is 0.5.

Methods Summary

get_opd(wave)	Compute the OPD appropriate for that phase mask for some given pixel spacing corresponding to the supplied Wavefront

Methods Documentation

get_opd(*wave*)
Compute the OPD appropriate for that phase mask for some given pixel spacing corresponding to the supplied Wavefront

RectangularFieldStop

class poppy.**RectangularFieldStop**(*name='unnamed field stop'*, *width=<Quantity 0.5 arcsec>*, *height=<Quantity 5. arcsec>*, ***kwargs*)
Bases: poppy.optics.AnalyticImagePlaneElement

Defines an ideal rectangular field stop

name
[string] Descriptive name

width, height: float
Size of the field stop, in arcseconds. Default 0.5 width, height 5.

Methods Summary

get_transmission(wave)	Compute the transmission inside/outside of the field stop.

Methods Documentation

get_transmission(*wave*)
Compute the transmission inside/outside of the field stop.

SquareFieldStop

class poppy.**SquareFieldStop**(*name='unnamed field stop'*, *size=<Quantity 20. arcsec>*, ***kwargs*)
Bases: poppy.optics.RectangularFieldStop

Defines an ideal square field stop

name
[string] Descriptive name

size
[float] Size of the field stop, in arcseconds. Default 20.

AnnularFieldStop

class poppy.**AnnularFieldStop**(*name='unnamed annular field stop'*, *radius_inner=0.0*, *radius_outer=1.0*, ***kwargs*)
Bases: poppy.optics.AnalyticImagePlaneElement

Defines a circular field stop with an (optional) opaque circular center region

name
[string] Descriptive name

radius_inner
[float] Radius of the central opaque region, in arcseconds. Default is 0.0 (no central opaque spot)

radius_outer
[float] Radius of the circular field stop outer edge. Default is 10. Set to 0.0 for no outer edge.

Methods Summary

get_transmission(wave)	Compute the transmission inside/outside of the field stop.

Methods Documentation

get_transmission(*wave*)
Compute the transmission inside/outside of the field stop.

HexagonFieldStop

class poppy.**HexagonFieldStop**(*name=None*, *side=None*, *diameter=None*, *flattoflat=None*, ***kwargs*)
Bases: poppy.optics.AnalyticImagePlaneElement

Defines an ideal hexagonal field stop

Specify either the side length (= corner radius) or the flat-to-flat distance, or the point-to-point diameter, in angular units

name
[string] Descriptive name

side
[float, optional] side length (and/or radius) of hexagon, in arcsec. Overrides flattoflat if both are present.

flattoflat
[float, optional] Distance between sides (flat-to-flat) of the hexagon, in arcsec. Default is 1.0

diameter
[float, optional] point-to-point diameter of hexagon. Twice the side length. Overrides flattoflat, but is overridden by side.

Note you can also specify the standard parameter "rotation" to rotate the hexagon by some amount.

Attributes Summary

diameter
flat_to_flat

Methods Summary

get_transmission(wave)	Compute the transmission inside/outside of the occulter.

Attributes Documentation

diameter

flat_to_flat

Methods Documentation

get_transmission(*wave*)
Compute the transmission inside/outside of the occulter.

CircularOcculter

class poppy.**CircularOcculter**(*name='unnamed occulter'*, *radius=1.0*, ***kwargs*)
Bases: poppy.optics.AnnularFieldStop

Defines an ideal circular occulter (opaque circle)

name
[string] Descriptive name

radius
[float] Radius of the occulting spot, in arcseconds. Default is 1.0

BarOcculter

class poppy.**BarOcculter**(*name='bar occulter'*, *width=<Quantity 1. arcsec>*, *height=<Quantity 10. arcsec>*, ***kwargs*)
Bases: poppy.optics.AnalyticImagePlaneElement

Defines an ideal bar occulter (like in MIRI's Lyot coronagraph)

name
[string] Descriptive name

width
[float] width of the bar stop, in arcseconds. Default is 1.0

height: float
heightof the bar stop, in arcseconds. Default is 10.0

Methods Summary

get_transmission(wave)	Compute the transmission inside/outside of the occulter.

Methods Documentation

get_transmission(*wave*)
Compute the transmission inside/outside of the occulter.

FQPM_FFT_aligner

class poppy.**FQPM_FFT_aligner**(*name='FQPM FFT aligner'*, *direction='forward'*, ***kwargs*)
Bases: poppy.optics.AnalyticOpticalElement

Helper class for modeling FQPMs accurately

Adds (or removes) a slight wavelength- and pixel-scale-dependent tilt to a pupil wavefront, to ensure the correct alignment of the image plane FFT'ed PSF with the desired quad pixel alignment for the FQPM.

This is purely a computational convenience tool to work around the pixel coordinate restrictions imposed by the FFT algorithm, not a representation of any physical optic.

direction
[string] 'forward' or 'backward'

Methods Summary

get_opd(wave)	Compute the required tilt needed to get the PSF centered on the corner between the 4 central pixels, not on the central pixel itself.

Methods Documentation

get_opd(*wave*)
Compute the required tilt needed to get the PSF centered on the corner between the 4 central pixels, not on the central pixel itself.

CircularAperture

class poppy.**CircularAperture**(*name=None*, *radius=<Quantity 1. m>*, *pad_factor=1.0*, *planetype=<PlaneType.unspecified: 0>*, ***kwargs*)
Bases: poppy.optics.AnalyticOpticalElement

Defines an ideal circular pupil aperture

name
[string] Descriptive name

radius
[float] Radius of the pupil, in meters. Default is 1.0

pad_factor
[float, optional] Amount to oversize the wavefront array relative to this pupil. This is in practice not very useful, but it provides a straightforward way of verifying during code testing that the amount of padding (or size of the circle) does not make any numerical difference in the final result.

Methods Summary

get_transmission(wave)	Compute the transmission inside/outside of the aperture.

Methods Documentation

get_transmission(*wave*)
Compute the transmission inside/outside of the aperture.

HexagonAperture

class poppy.**HexagonAperture**(*name=None*, *side=None*, *diameter=None*, *flattoflat=None*, ***kwargs*)
Bases: poppy.optics.AnalyticOpticalElement

Defines an ideal hexagonal pupil aperture

Specify either the side length (= corner radius) or the flat-to-flat distance, or the point-to-point diameter.

name
[string] Descriptive name

side
[float, optional] side length (and/or radius) of hexagon, in meters. Overrides flattoflat if both are present.

flattoflat
[float, optional] Distance between sides (flat-to-flat) of the hexagon, in meters. Default is 1.0

diameter
[float, optional] point-to-point diameter of hexagon. Twice the side length. Overrides flattoflat, but is overridden by side.

Attributes Summary

diameter
flat_to_flat

Methods Summary

get_transmission(wave)	Compute the transmission inside/outside of the occulter.

Attributes Documentation

diameter

flat_to_flat

Methods Documentation

get_transmission(*wave*)
Compute the transmission inside/outside of the occulter.

MultiHexagonAperture

class poppy.**MultiHexagonAperture**(*name='MultiHex'*, *flattoflat=1.0*, *side=None*, *gap=0.01*, *rings=1*, *segmentlist=None*, *center=False*, ***kwargs*)
Bases: poppy.optics.AnalyticOpticalElement

Defines a hexagonally segmented aperture

name
[string] Descriptive name

rings
[integer] The number of rings of hexagons to include, not counting the central segment (i.e. 2 for a JWST-like aperture, 3 for a Keck-like aperture, and so on)

side
[float, optional] side length (and/or radius) of hexagon, in meters. Overrides flattoflat if both are present.

flattoflat
[float, optional] Distance between sides (flat-to-flat) of the hexagon, in meters. Default is 1.0

gap: float, optional
Gap between adjacent segments, in meters. Default is 0.01 m = 1 cm

center
[bool, optional] should the central segment be included? Default is False.

segmentlist
[list of ints, optional] This allows one to specify that only a subset of segments are present, for a partially populated segmented telescope, non-redundant segment set, etc. Segments are numbered from 0 for the center segment, 1 for the segment immediately above it, and then clockwise around each ring. For example, segmentlist=[1,3,5] would make an aperture of 3 segments.

Note that this routine becomes a bit slow for nrings >4. For repeated computations on the same aperture, it will be faster to create this once, save it to a FITS file using the toFITS() method, and then use that.

Methods Summary

get_transmission(wave)	Compute the transmission inside/outside of the occulter.

Methods Documentation

get_transmission(*wave*)
Compute the transmission inside/outside of the occulter.

NgonAperture

class poppy.**NgonAperture**(*name=None*, *nsides=6*, *radius=<Quantity 1. m>*, *rotation=0.0*, ***kwargs*)
Bases: poppy.optics.AnalyticOpticalElement

Defines an ideal N-gon pupil aperture.

name
[string] Descriptive name

nsides
[integer] Number of sides. Default is 6.

radius
[float] radius to the vertices, meters. Default is 1.

rotation
[float] Rotation angle to first vertex, in degrees counterclockwise from the +X axis. Default is 0.

Methods Summary

get_transmission(wave)	Compute the transmission inside/outside of the occulter.

Methods Documentation

get_transmission(*wave*)
Compute the transmission inside/outside of the occulter.

RectangleAperture

class poppy.**RectangleAperture**(*name=None*, *width=<Quantity 0.5 m>*, *height=<Quantity 1. m>*, *rotation=0.0*, ***kwargs*)
Bases: poppy.optics.AnalyticOpticalElement

Defines an ideal rectangular pupil aperture

name
[string] Descriptive name

width
[float] width of the rectangle, in meters. Default is 0.5

height
[float] height of the rectangle, in meters. Default is 1.0

rotation
[float] Rotation angle for 'width' axis. Default is 0.

Methods Summary

get_transmission(wave)	Compute the transmission inside/outside of the occulter.

Methods Documentation

get_transmission(*wave*)
Compute the transmission inside/outside of the occulter.

SquareAperture

class poppy.**SquareAperture**(*name=None*, *size=<Quantity 1. m>*, ***kwargs*)

Bases: poppy.optics.RectangleAperture

Defines an ideal square pupil aperture

name
: [string] Descriptive name

size: float
: side length of the square, in meters. Default is 1.0

rotation
: [float] Rotation angle for the square. Default is 0.

Attributes Summary

size

Attributes Documentation

size

SecondaryObscuration

class poppy.**SecondaryObscuration**(*name=None*, *secondary_radius=<Quantity 0.5 m>*, *n_supports=4*, *support_width=<Quantity 0.01 m>*, *support_angle_offset=0.0*, ***kwargs*)

Bases: poppy.optics.AnalyticOpticalElement

Defines the central obscuration of an on-axis telescope including secondary mirror and supports

The number of supports is adjustable but they are always radially symmetric around the center. See AsymmetricSecondaryObscuration if you need more flexibility.

secondary_radius
: [float or astropy Quantity length] Radius of the circular secondary obscuration, in meters or other unit. Default 0.5 m

n_supports
: [int] Number of secondary mirror supports ("spiders"). These will be spaced equally around a circle. Default is 4.

support_width
: [float or astropy Quantity length] Width of each support, in meters or other unit. Default is 0.01 m = 1 cm.

support_angle_offset
: [float] Angular offset, in degrees, of the first secondary support from the X axis.

Methods Summary

get_transmission(wave)	Compute the transmission inside/outside of the obscuration

Methods Documentation

get_transmission(*wave*)
Compute the transmission inside/outside of the obscuration

AsymmetricSecondaryObscuration

class poppy.**AsymmetricSecondaryObscuration**(*support_angle=(0, 90, 240)*, *support_width=<Quantity 0.01 m>*, *support_offset_x=0.0*, *support_offset_y=0.0*, ***kwargs*)

Bases: poppy.optics.SecondaryObscuration

Defines a central obscuration with one or more supports which can be oriented at arbitrary angles around the primary mirror, a la the three supports of JWST

This also allows for secondary supports that do not intersect with the primary mirror center; use the support_offset_x and support_offset_y parameters to apply offsets relative to the center for the origin of each strut.

secondary_radius
[float] Radius of the circular secondary obscuration. Default 0.5 m

support_angle
[ndarray or list of floats] The angle measured counterclockwise from +Y for each support

support_width
[float or astropy Quantity of type length, or list of those] if scalar, gives the width for all support struts if a list, gives separately the width for each support strut independently. Widths in meters or other unit if specified. Default is 0.01 m = 1 cm.

support_offset_x
[float, or list of floats.] Offset in the X direction of the start point for each support. if scalar, applies to all supports; if a list, gives a separate offset for each.

support_offset_y
[float, or list of floats.] Offset in the Y direction of the start point for each support. if scalar, applies to all supports; if a list, gives a separate offset for each.

Methods Summary

get_transmission(wave)	Compute the transmission inside/outside of the obscuration

Methods Documentation

get_transmission(*wave*)
Compute the transmission inside/outside of the obscuration

ThinLens

class poppy.**ThinLens**(*name='Thin lens'*, *nwaves=4.0*, *reference_wavelength=<Quantity 1.e-06 m>*, *radius=<Quantity 1. m>*, ***kwargs*)

Bases: poppy.optics.CircularAperture

An idealized thin lens, implemented as a Zernike defocus term.

The sign convention adopted is the usual for lenses: a "positive" lens is converging (i.e. convex), a "negative" lens is diverging (i.e. concave).

In other words, a positive number of waves of defocus indicates a lens with positive OPD at the center, and negative at its rim. (Note, this is opposite the sign convention for Zernike defocus)

nwaves
[float] The number of waves of defocus, peak to valley. May be positive or negative. This is applied as a normalization over an area defined by the circumscribing circle of the input wavefront. That is, there will be nwaves defocus peak-to-valley over the region of the pupil that has nonzero input intensity.

reference_wavelength
[float] Wavelength, in meters, at which that number of waves of defocus is specified.

radius
[float] Pupil radius, in meters, over which the Zernike defocus term should be computed such that rho = 1 at r = radius.

Methods Summary

get_opd(wave)	Return the optical path difference, given a wavelength.

Methods Documentation

get_opd(*wave*)
Return the optical path difference, given a wavelength.

wave
[float or obj] either a scalar wavelength or a Wavefront object

ndarray giving OPD in meters

GaussianAperture

class poppy.**GaussianAperture**(*name=None*, *fwhm=None*, *w=None*, *pupil_diam=None*, ***kwargs*)

Bases: poppy.optics.AnalyticOpticalElement

Defines an ideal Gaussian apodized pupil aperture, or at least as much of one as can be fit into a finite-sized array

The Gaussian's width must be set with either the fwhm or w parameters.

Note that this makes an optic whose electric *field amplitude* transmission is the specified Gaussian; thus the intensity transmission will be the square of that Gaussian.

name
[string] Descriptive name

fwhm
[float, optional.] Full width at half maximum for the Gaussian, in meters.

w
[float, optional] Beam width parameter, equal to fwhm/(2*sqrt(ln(2))).

pupil_diam
[float, optional] default pupil diameter for cases when it is not otherwise specified (e.g. displaying the optic by itself.) Default value is 3x the FWHM.

Attributes Summary

fwhm	

Methods Summary

get_transmission(wave)	Compute the transmission inside/outside of the aperture.

Attributes Documentation

fwhm

Methods Documentation

get_transmission(*wave*)
Compute the transmission inside/outside of the aperture.

CompoundAnalyticOptic

class poppy.**CompoundAnalyticOptic**(*opticslist=None*, *name='unnamed'*, *mergemode='and'*, *verbose=True*, ***kwargs*)
Bases: poppy.optics.AnalyticOpticalElement

Define a compound analytic optical element made up of the combination of two or more individual optical elements.

This is just a convenience routine for semantic organization of optics. It can be useful to keep the list of optical planes cleaner, but you can certainly just add a whole bunch of planes all in a row without using this class to group them.

All optics should be of the same plane type (pupil or image); propagation between different optics contained inside one compound is not supported.

opticslist
[list] A list of AnalyticOpticalElements to be merged together.

mergemode
[string, default = 'and']

Method for merging transmissions:

'and'
[resulting transmission is product of constituents. (E.g] trans = trans1*trans2)

'or'
[resulting transmission is sum of constituents, with overlap] subtracted. (E.g. trans = trans1 + trans2 - trans1*trans2)

In both methods, the resulting OPD is the sum of the constituents' OPDs.

Methods Summary

get_opd(wave)	Return the optical path difference, given a wavelength.
get_transmission(wave)	Note that this is the **amplitude** transmission, not the total intensity transmission.

Methods Documentation

get_opd(*wave*)
Return the optical path difference, given a wavelength.

wave
[float or obj] either a scalar wavelength or a Wavefront object

ndarray giving OPD in meters

get_transmission(*wave*)
Note that this is the **amplitude** transmission, not the total intensity transmission.

QuadPhase

class poppy.**QuadPhase**(*z=<Quantity 1. m>*, *planetype=<PlaneType.intermediate: 5>*, *name='Quadratic Wavefront Curvature Operator'*, ***kwargs*)
Bases: poppy.optics.AnalyticOpticalElement

Quadratic phase factor, q(z) suitable for representing a radially-dependent wavefront curvature.

z
[float or astropy.Quantity of type length] radius of curvature

planetype
[poppy.PlaneType constant] plane type

name
[string] Descriptive string name

Lawrence eq. 88

Methods Summary

get_phasor(wave)	return complex phasor for the quadratic phase

Methods Documentation

get_phasor(*wave*)
return complex phasor for the quadratic phase

wave
[obj] a Fresnel Wavefront object

QuadraticLens

class poppy.**QuadraticLens**(*f_lens=<Quantity 1. m>*, *planetype=<PlaneType.unspecified: 0>*, *name='Quadratic Lens'*, ***kwargs*)
Bases: poppy.fresnel.QuadPhase

Gaussian Lens

Thin wrapper for QuadPhase

f_lens
[float or astropy.Quantity of type length] Focal length of this lens

name
[string] Descriptive string name

planetype
[poppy.PlaneType constant] plane type

FresnelWavefront

class poppy.**FresnelWavefront**(*beam_radius*, *units=Unit("m")*, *rayleigh_factor=2.0*, *oversample=2*, ***kwargs*)
Bases: poppy.poppy_core.BaseWavefront

Wavefront for Fresnel diffraction calculation.

This class inherits from and extends the Fraunhofer-domain poppy.Wavefront class.

beam_radius
[astropy.Quantity of type length] Radius of the illuminated beam at the initial optical plane. I.e. this would be the pupil aperture radius in an entrance pupil.

units
[astropy.units.Unit] Astropy units of input parameters

rayleigh_factor:
Threshold for considering a wave spherical.

oversample
[float] Padding factor to apply to the wavefront array, multiplying on top of the beam radius.

- **Lawrence, G. N. (1992), Optical Modeling, in Applied Optics and Optical Engineering., vol. XI,** edited by R. R. Shannon and J. C. Wyant., Academic Press, New York.
- https://en.wikipedia.org/wiki/Gaussian_beam
- **IDEX Optics and Photonics(n.d.), Gaussian Beam Optics,** [online] Available from: https://marketplace.idexop.com/store/SupportDocuments/All_About_Gaussian_Beam_OpticsWEB.pdf

- **Krist, J. E. (2007), PROPER: an optical propagation library for IDL,** vol. 6675, p. 66750P-66750P-9. [online] Available from: http://dx.doi.org/10.1117/12.731179
- Andersen, T., and A. Enmark (2011), Integrated Modeling of Telescopes, Springer Science & Business Media.

Attributes Summary

angular_coordinates	Should coordinates be expressed in arcseconds instead of meters at the current plane?
divergence	Half-angle divergence of the gaussian beam
fov	FOV in arcseconds, if applicable
param_str	Formatted string of gaussian beam parameters.
pixelscale	Pixelscale, in meters by default or in arcseconds if angular_coordinates is True
waists	each [z_w_0,w_0] for each waist generated by an optic
z_r	Rayleigh distance for the gaussian beam, based on current beam waist and wavelength.

Methods Summary

apply_lens_power(optic[, ignore_wavefront])	Adds lens wavefront curvature to the wavefront corresponding to the lens' focal length f_l, and updates the Gaussian beam parameters of the wavefront.
coordinates()	Return Y, X coordinates for this wavefront, in the manner of numpy.indices()
display(*args, **kwargs)	Display wavefront on screen
from_wavefront(wavefront)	Convert a Fraunhofer type wavefront object to a Fresnel one
planar_range(z)	Returns True if the input range z is within the Rayleigh range of the waist.
propagate_direct(z)	Implements the direct propagation algorithm as described in Andersen & Enmark (2011).
propagate_fresnel(delta_z[, display_intermed])	Top-level routine for Fresnel diffraction propagation
propagate_to(optic, distance)	Propagates a wavefront object to the next optic in the list, after some separation distance (which might be zero).
pupil_coordinates(x, y, pixelscale)	Utility function to generate coordinates arrays for a pupil plane wavefront
r_c([z])	The gaussian beam radius of curvature as a function of distance z
spot_radius([z])	radius of a propagating gaussian wavefront, at a distance z

Attributes Documentation

angular_coordinates = False
Should coordinates be expressed in arcseconds instead of meters at the current plane?

divergence

Half-angle divergence of the gaussian beam

I.e. the angle between the optical axis and the beam radius (at a large distance from the waist) in radians.

fov
FOV in arcseconds, if applicable

param_str
Formatted string of gaussian beam parameters.

pixelscale
Pixelscale, in meters by default or in arcseconds if angular_coordinates is True

waists
each [z_w_0,w_0] for each waist generated by an optic

z_r
Rayleigh distance for the gaussian beam, based on current beam waist and wavelength.

I.e. the distance along the propagation direction from the beam waist at which the area of the cross section has doubled. The depth of focus is conventionally twice this distance.

Methods Documentation

apply_lens_power(*optic*, *ignore_wavefront=False*)
Adds lens wavefront curvature to the wavefront corresponding to the lens' focal length f_l, and updates the Gaussian beam parameters of the wavefront.

optic
[QuadraticLens] An optic

ignore_wavefront
[boolean] If True then only gaussian beam propagation parameters will be updated and the wavefront surface will not be calculated. Useful for quick calculations of gaussian laser beams

coordinates()
Return Y, X coordinates for this wavefront, in the manner of numpy.indices()

This function knows about the offset resulting from FFTs. Use it whenever computing anything measured in wavefront coordinates.

The behavior for Fresnel wavefronts is slightly different from Fraunhofer wavefronts, in that the optical axis is *not* the exact center of an array (the corner between pixels for an even number of pixels), but rather is a specific pixel (e.g. pixel 512,512 for a 1024x1024 array). This is for consistency with the array indexing convention used in FFTs since this class depends on FFTs rather than the more flexible matrix DFTs for its propagation.

For Fresnel wavefronts, this depends on the focal length to get the image scale right.

Y, X
[array_like] Wavefront coordinates in either meters or arcseconds for pupil and image, respectively

display(**args*, ***kwargs*)
Display wavefront on screen

what
[string] What to display. Must be one of {intensity, phase, best}. 'Best' implies to display the phase if there is nonzero OPD, or else display the intensity for a perfect pupil.

nrows
[int] Number of rows to display in current figure (used for showing steps in a calculation)

row
[int] Which row to display this one in? If set to None, use the wavefront's self.current_plane_index

vmin, vmax
[floats] min and maximum values to display. When left unspecified, these default to [0, intens.max()] for linear (scale='linear') intensity plots, [1e-6*intens.max(), intens.max()] for logarithmic (scale='log') intensity plots, and [-0.25, 0.25] waves for phase plots.

scale
[string] 'log' or 'linear', to define the desired display scale type for intensity. Default is log for image planes, linear otherwise.

imagecrop
[float, optional] Crop the displayed image to a smaller region than the full array. For image planes in angular coordinates, this is given in units of arcseconds. The default is 5, so only the innermost 5x5 arcsecond region will be shown. This default may be changed in the POPPY config file. If the image size is < 5 arcsec then the entire image is displayed. For planes in linear physical coordinates such as pupils, this is given in units of meters, and the default is no cropping (i.e. the entire array will be displayed unless this keyword is set explicitly).

showpadding
[bool, optional] For wavefronts that have been padded with zeros for oversampling, show the entire padded arrays, or just the good parts? Default is False, to show just the central region of interest.

colorbar
[bool] Display colorbar

crosshairs
[bool] Display a crosshairs indicator showing the axes centered on (0,0)

ax
[matplotlib Axes, optional] axes to display into. If not set, will create new axes.

use_angular_coordinates
[bool, optional] Should the axes be labeled in angular units of arcseconds? This is used by FresnelWavefront, where non-angular coordinates are possible everywhere. When using Fraunhofer propagation, this should be left as None so that the coordinates are inferred from the planetype attribute. (Default: None, infer coordinates from planetype)

figure
[matplotlib figure] The current figure is modified.

`classmethod from_wavefront(`*wavefront*`)`
Convert a Fraunhofer type wavefront object to a Fresnel one

Note, for now this function only works if the input wavefront is at a pupil plane, so the Fraunhofer wavefront has pixelscale in meters/pix rather than arcsec/pix. Conversion from image planes may be added later.

wavefront
[Wavefront] The (Fraunhofer-type) wavefront to be converted

`planar_range(`*z*`)`
Returns True if the input range z is within the Rayleigh range of the waist.

z
[float] distance from the beam waist

propagate_direct(*z*)
Implements the direct propagation algorithm as described in Andersen & Enmark (2011). Works best for far field propagation. Not part of the Gaussian beam propagation method.

z
[float or Astropy.Quantity length] the distance from the current location to propagate the beam.

propagate_fresnel(*delta_z*, *display_intermed=False*)
Top-level routine for Fresnel diffraction propagation

Each spherical wavefront is propagated to a waist and then to the next appropriate plane
(spherical or planar).

delta_z
[float] the distance from the current location to propagate the beam.

display_intermed
[boolean] If True, display the complex start, intermediates waist and end surfaces.

propagate_to(*optic*, *distance*)
Propagates a wavefront object to the next optic in the list, after some separation distance (which might be zero). Modifies this wavefront object itself.

Transformations between most planes use Fresnel propagation. If the target plane is an image plane, the output wavefront will be set to provide its coordinates in arcseconds based on its focal length, but it retains its internal dimensions in meters for future Fresnel propagations. Transformations to a Detector plane are handled separately to allow adjusting the pixel scale to match the target scale. Transformations from any frame through a rotation plane simply rotate the wavefront accordingly.

optic
[OpticalElement] The optic to propagate to. Used for determining the appropriate optical plane.

distance
[astropy.Quantity of dimension length] separation distance of this optic relative to the prior optic in the system.

static pupil_coordinates(*x*, *y*, *pixelscale*)
Utility function to generate coordinates arrays for a pupil plane wavefront

x, y
[array_like] pixel indices

pixelscale
[float or 2-tuple of floats] the pixel scale in meters/pixel, optionally different in X and Y

Y, X
[array_like] Wavefront coordinates in either meters or arcseconds for pupil and image, respectively

r_c(*z=None*)
The gaussian beam radius of curvature as a function of distance z

z
[float, optional] Distance along the optical axis. If not specified, the wavefront's current z coordinate will be used, returning the beam radius of curvature at the current position.

Astropy.units.Quantity of dimension length

spot_radius(*z=None*)
radius of a propagating gaussian wavefront, at a distance z

z
[float, optional] Distance along the optical axis. If not specified, the wavefront's current z coordinate will be used, returning the beam radius at the current position.

Astropy.units.Quantity of dimension length

FresnelOpticalSystem

class poppy.**FresnelOpticalSystem**(*name='unnamed system'*, *pupil_diameter=<Quantity 1. m>*, *npix=1024*, *beam_ratio=0.5*, *verbose=True*)

Bases: poppy.poppy_core.BaseOpticalSystem

Class representing a series of optical elements, through which light can be propagated using the Fresnel formalism.

This is comparable to the "regular" (Fraunhofer-domain) OpticalSystem, but adds functionality for propagation to arbitrary optical planes rather than just pupil and image planes.

name
[string] descriptive name of optical system

pupil_diameter
[astropy.Quantity of dimension length] Diameter of entrance pupil

npix
[int] Number of pixels across the entrance pupil by default 1024

beam_ratio
[float] Padding factor for the entrance pupil; what fraction of the array should correspond to the entrance pupil. Default is 0.5, which corresponds to Nyquist sampling (2 pixels per resolution element)

verbose
[bool] whether to be more verbose with log output while computing

Methods Summary

add_detector([pixelscale, fov_pixels, distance])	Add a detector to the optical system
add_optic([optic, distance, index])	Add an optic to the optical system
describe()	Print out a string table describing all planes in an optical system
input_wavefront([wavelength])	Create a Wavefront object suitable for sending through a given optical system.
propagate(wavefront[, normalize, ...])	Core low-level routine for propagating a wavefront through an optical system

Methods Documentation

add_detector(*pixelscale=<Quantity 10. micron / pix>*, *fov_pixels=<Quantity 10. pix>*, *distance=<Quantity 0. m>*)
Add a detector to the optical system

pixelscale
[astropy.Quantity, with units micron/pixel or equivalent] The pixel scale at the detector

fov_pixels
[astropy.Quantity with units pixel] The number of pixels per axis of the detector. Assumes square detector.

distance
[astropy.Quantity of dimension length] separation distance of this optic relative to the prior optic in the system.

add_optic(*optic=None*, *distance=<Quantity 0. m>*, *index=None*)
Add an optic to the optical system

optic
[OpticalElement instance] Some optic

distance
[astropy.Quantity of dimension length] separation distance of this optic relative to the prior optic in the system.

index
[int] Index at which to insert the new optical element

describe()
Print out a string table describing all planes in an optical system

input_wavefront(*wavelength=<Quantity 1.e-06 m>*)
Create a Wavefront object suitable for sending through a given optical system.

Uses self.source_offset to assign an off-axis tilt, if requested. (FIXME does not work for Fresnel yet)

wavelength
[float] Wavelength in meters

wavefront
[poppy.fresnel.FresnelWavefront instance] A wavefront appropriate for passing through this optical system.

propagate(*wavefront*, *normalize='none'*, *return_intermediates=False*, *display_intermediates=False*)
Core low-level routine for propagating a wavefront through an optical system

See docstring of OpticalSystem.propagate for details

WavefrontError

class poppy.**WavefrontError**(***kwargs*)
Bases: poppy.optics.AnalyticOpticalElement

A base class for different sources of wavefront error

Analytic optical elements that represent wavefront error should derive from this class and override methods appropriately. Defined to be a pupil-plane optic.

Methods Summary

get_opd(wave[, units])	Construct the optical path difference array for a wavefront error source as evaluated across the pupil for an input wavefront wave
peaktovalley()	Peak-to-valley wavefront error induced by this surface
rms()	RMS wavefront error induced by this surface

Methods Documentation

get_opd(*wave*, *units='meters'*)
Construct the optical path difference array for a wavefront error source as evaluated across the pupil for an input wavefront `wave`

wave
[Wavefront] Wavefront object with a `coordinates` method that returns (y, x) coordinate arrays in meters in the pupil plane

units
['meters' or 'waves'] The units of optical path difference (Default: meters)

peaktovalley()
Peak-to-valley wavefront error induced by this surface

rms()
RMS wavefront error induced by this surface

ParameterizedWFE

class poppy.**ParameterizedWFE**(*name='Parameterized Distortion'*, *coefficients=None*, *radius=<Quantity 1. m>*, *basis_factory=None*, ***kwargs*)
Bases: `poppy.wfe.WavefrontError`

Define an optical element in terms of its distortion as decomposed into a set of orthonormal basis functions (e.g. Zernikes, Hexikes, etc.). Included basis functions are normalized such that user-provided coefficients correspond to meters RMS wavefront aberration for that basis function.

coefficients
[iterable of numbers] The contribution of each term to the final distortion, in meters RMS wavefront error. The coefficients are interpreted as indices in the order of Noll et al. 1976: the first term corresponds to j=1, second to j=2, and so on.

radius
[float] Pupil radius, in meters. Defines the region of the input wavefront array over which the distortion terms will be evaluated. For non-circular pupils, this should be the circle circumscribing the actual pupil shape.

basis_factory
[callable] basis_factory will be called with the arguments `nterms`, `rho`, `theta`, and `outside`.

`nterms` specifies how many terms to compute, starting with the j=1 term in the Noll indexing convention for `nterms` = 1 and counting up.

`rho` and `theta` are square arrays holding the rho and theta coordinates at each pixel in the pupil plane. `rho` is normalized such that `rho` == 1.0 for pixels at `radius` meters from the center.

`outside` contains the value to assign pixels outside the radius `rho` == 1.0. (Always 0.0, but provided for compatibility with `zernike.zernike_basis` and `zernike.hexike_basis`.)

Methods Summary

`get_opd`(wave[, units])	Construct the optical path difference array for a wavefront error source as evaluated across the pupil for an input wavefront `wave`

Methods Documentation

get_opd(*wave*, *units='meters'*)
Construct the optical path difference array for a wavefront error source as evaluated across the pupil for an input wavefront `wave`

wave
[Wavefront] Wavefront object with a `coordinates` method that returns (y, x) coordinate arrays in meters in the pupil plane

units
['meters' or 'waves'] The units of optical path difference (Default: meters)

ZernikeWFE

class poppy.**ZernikeWFE**(*name='Zernike WFE'*, *coefficients=None*, *radius=None*, *aperture_stop=False*, ***kwargs*)
Bases: `poppy.wfe.WavefrontError`

Define an optical element in terms of its Zernike components by providing coefficients for each Zernike term contributing to the analytic optical element.

coefficients
[iterable of floats] Specifies the coefficients for the Zernike terms, ordered according to the convention of Noll et al. JOSA 1976. The coefficient is in meters of optical path difference (not waves).

radius
[float] Pupil radius, in meters, over which the Zernike terms should be computed such that rho = 1 at r = `radius`.

Methods Summary

`get_opd`(wave[, units])	Parameters
`get_transmission`(wave)	Note that this is the **amplitude** transmission, not the total intensity transmission.

Methods Documentation

get_opd(*wave*, *units='meters'*)

wave
[poppy.Wavefront (or float)] Incoming Wavefront before this optic to set wavelength and scale, or a float giving the wavelength in meters for a temporary Wavefront used to compute the OPD.

units
['meters' or 'waves'] Coefficients are supplied in `ZernikeWFE.coefficients` as meters of OPD, but the resulting OPD can be converted to waves based on the `Wavefront` wavelength or a supplied wavelength value.

get_transmission(*wave*)
Note that this is the **amplitude** transmission, not the total intensity transmission.

SineWaveWFE

class poppy.**SineWaveWFE**(*name='Sine WFE'*, *spatialfreq=1.0*, *amplitude=1e-06*, *phaseoffset=0*, ***kwargs*)
Bases: poppy.wfe.WavefrontError

A single sine wave ripple across the optic

Specified as a a spatial frequency in cycles per meter, an optional phase offset in cycles, and an amplitude.

By default the wave is oriented in the X direction. Like any AnalyticOpticalElement class, you can also specify a rotation parameter to rotate the direction of the sine wave.

(N.b. we intentionally avoid letting users specify this in terms of a spatial wavelength because that would risk potential ambiguity with the wavelength of light.)

Methods Summary

get_opd(wave[, units])	Parameters

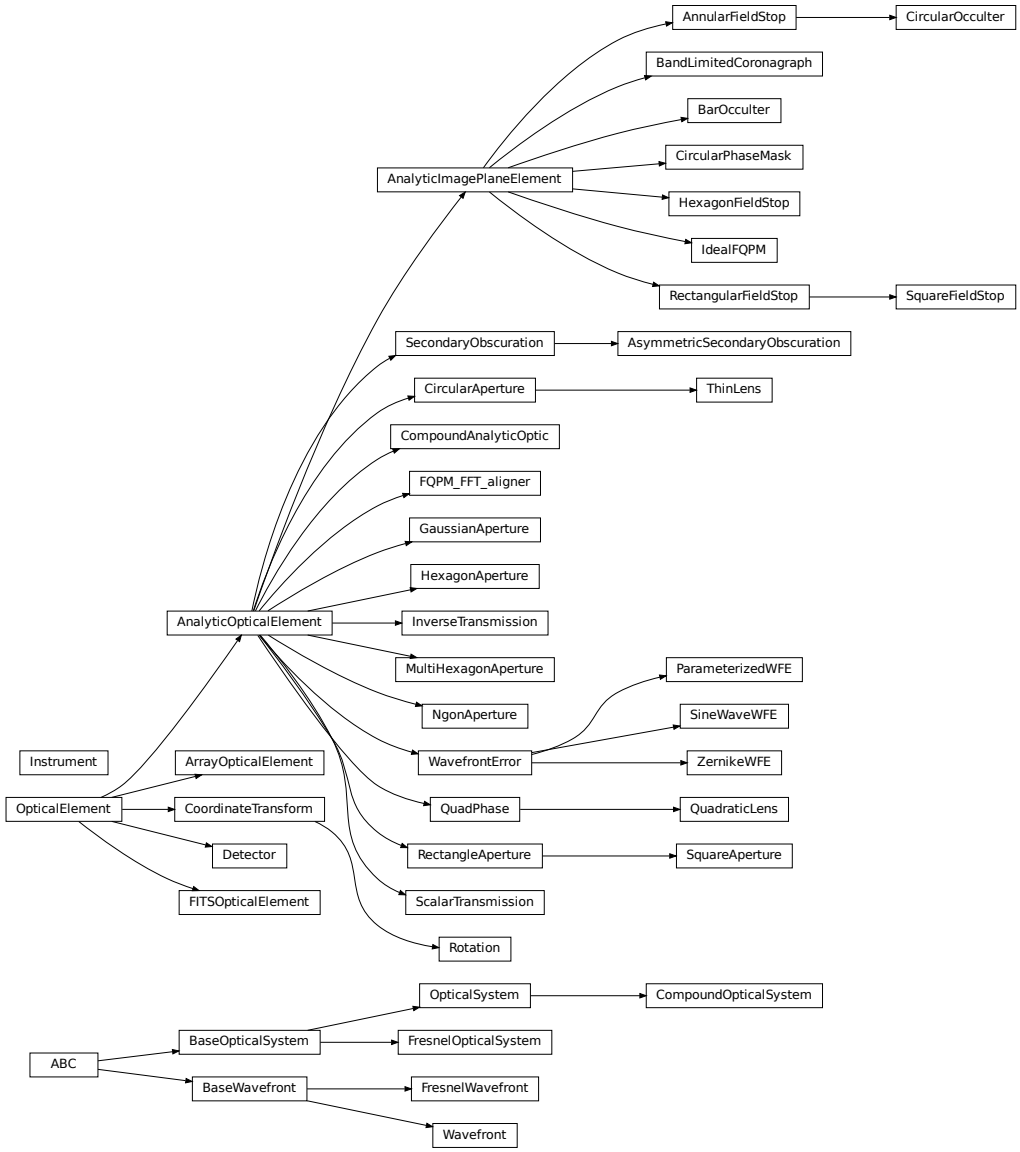
Methods Documentation

get_opd(*wave*, *units='meters'*)

wave
[poppy.Wavefront (or float)] Incoming Wavefront before this optic to set wavelength and scale, or a float giving the wavelength in meters for a temporary Wavefront used to compute the OPD.

units
['meters' or 'waves'] Coefficients are supplied as meters of OPD, but the resulting OPD can be converted to waves based on the Wavefront wavelength or a supplied wavelength value.

12.1.3 Class Inheritance Diagram



12.2 poppy.zernike Module

Zernike & Related Polynomials

This module implements several sets of orthonormal polynomials for measuring and modeling wavefronts:

- the classical Zernike polynomials, which are orthonormal over the unit circle.

- 'Hexikes', orthonormal over the unit hexagon
- **tools for creating a custom set orthonormal over a numerically supplied JWST pupil,** or other generalized pupil
- Segmented bases with piston, tip, & tilt of independent hexagonal segments.

For definitions of Zernikes and a basic introduction to why they are a useful way to

parametrize data, see e.g.
Hardy's 'Adaptive Optics for Astronomical Telescopes' section 3.5.1 or even just the Wikipedia page is pretty decent.

For definition of the hexagon and arbitrary pupil polynomials, a good reference to the

Gram-Schmidt orthonormalization process as applied to this case is
Mahajan and Dai, 2006. Optics Letters Vol 31, 16, p 2462:

12.2.1 Functions

R(n, m, rho)	Compute R[n, m], the Zernike radial polynomial
cached_zernike1	Compute Zernike based on Noll index *j*, using an LRU cache for efficiency.
hex_aperture([npix, rho, theta, vertical, ...])	Return an aperture function for a hexagon.
hexike_basis([nterms, npix, rho, theta, ...])	Return a list of hexike polynomials 1-N following the method of Mahajan and Dai 2006 for numerical orthonormalization
noll_indices(j)	Convert from 1-D to 2-D indexing for Zernikes or Hexikes.
opd_expand(opd[, aperture, nterms, basis])	Given a wavefront OPD map, return the list of coefficients in a given basis set (by default, Zernikes) that best fit the OPD map.
opd_expand_nonorthonormal(opd[, aperture, ...])	Modified version of opd_expand, for cases where the basis function is *not* orthonormal, for instance using the regular Zernike functions on obscured apertures.
opd_expand_segments(opd[, aperture, nterms, ...])	Expand OPD into a basis defined by segments, typically with piston, tip, & tilt of each.
opd_from_zernikes(coeffs[, basis, aperture, ...])	Synthesize an OPD from a set of coefficients
str_zernike(n, m)	Return analytic expression for a given Zernike in LaTeX syntax
zern_name(i)	Return a human-readable text name corresponding to some Zernike term as specified by j, the index
zernike(n, m[, npix, rho, theta, outside, ...])	Return the Zernike polynomial Z[m,n] for a given pupil.
zernike1(j, **kwargs)	Return the Zernike polynomial Z_j for pupil points {r,theta}.
zernike_basis([nterms, npix, rho, theta])	Return a cube of Zernike terms from 1 to N each as a 2D array showing the value at each point.
arbitrary_basis(aperture[, nterms, rho, ...])	Orthonormal basis on arbitrary aperture, via Gram-Schmidt

R

poppy.zernike.**R**(*n*, *m*, *rho*)
Compute R[n, m], the Zernike radial polynomial

n, m
[int] Zernike function degree

rho
[array] Image plane radial coordinates. `rho` should be 1 at the desired pixel radius of the unit circle

zernike.cached_zernike1

`zernike.`**`cached_zernike1`**
Compute Zernike based on Noll index *j*, using an LRU cache for efficiency. Refer to the `zernike1` docstring for details.

Note: all arguents should be plain ints, tuples, floats etc rather than Astropy Quantities.

hex_aperture

`poppy.zernike.`**`hex_aperture`**(*npix=1024*, *rho=None*, *theta=None*, *vertical=False*, *outside=0*)
Return an aperture function for a hexagon.

Note that the flat sides are aligned with the X direction by default. This is appropriate for the individual hex PMSA segments in JWST.

npix
[integer] Size, in pixels, of the aperture array. The hexagon will span the whole array from edge to edge in the direction aligned with its flat sides. (Ignored when `rho` and `theta` are supplied.)

rho, theta
[2D numpy arrays, optional] For some square aperture, rho and theta contain each pixel's coordinates in polar form. The hexagon will be defined such that it can be circumscribed in a rho = 1 circle.

vertical
[bool] Make flat sides parallel to the Y axis instead of the default X.

outside
[float] Value for pixels outside the hexagonal aperture. Default is `np.nan`, but you may also find it useful for this to be 0.0 sometimes.

hexike_basis

`poppy.zernike.`**`hexike_basis`**(*nterms=15*, *npix=512*, *rho=None*, *theta=None*, *aperture=None*, *vertical=False*, *outside=nan*)
Return a list of hexike polynomials 1-N following the method of Mahajan and Dai 2006 for numerical orthonormalization

This function orders the hexikes in a similar way as the Zernikes.

See also hexike_basis_wss for an alternative implementation.

nterms
[int] Number of hexike terms to compute, starting from piston. (e.g. `nterms=1` would return only the hexike analog to the Zernike piston term.) Default is 15.

npix
[int] Size, in pixels, of the aperture array. The hexagon will span the whole array from edge to edge in the direction aligned with its flat sides.

rho, theta
[2D numpy arrays, optional] For some square aperture, rho and theta contain each pixel's coordinates in polar form. The hexagon will be defined such that it can be circumscribed in a rho = 1 circle.

vertical
[bool] Make flat sides parallel to the Y axis instead of the default X. Default is False.

outside
[float] Value for pixels outside the hexagonal aperture. Default is `np.nan`, but you may also find it useful for this to be 0.0 sometimes.

aperture
[2D numpy array, optional] Aperture mask for which pixels are included within the aperture. All positive nonzero values are considered within the aperture; any pixels with zero, negative, or NaN values will be considered outside the aperture, and set equal to the 'outside' parameter value. If this parameter is not set, the aperture will be inferred from the provided rho and theta arrays.

noll_indices

poppy.zernike.**noll_indices**(*j*)
Convert from 1-D to 2-D indexing for Zernikes or Hexikes.

j
[int] Zernike function ordinate, following the convention of Noll et al. JOSA 1976. Starts at 1.

opd_expand

poppy.zernike.**opd_expand**(*opd*, *aperture=None*, *nterms=15*, *basis=<function zernike_basis>*, ***kwargs*)
Given a wavefront OPD map, return the list of coefficients in a given basis set (by default, Zernikes) that best fit the OPD map.

Note that this implementation of the function treats the Zernikes as an orthonormal basis, which is only true on the unobscured unit circle. See also `opd_expand_nonorthonormal` for an alternative approach for basis vectors that are not orthonormal, or `opd_expand_segments` for basis vectors defined over physically disjoint segments.

opd
[2D numpy.ndarray] The wavefront OPD map to expand in terms of the requested basis. Must be square.

aperture
[2D numpy.ndarray, optional] Aperture mask for which pixels are included within the aperture. NOTE - this is handed through to the basis function (see basis parameter) which is responsible for implementing this masking, if appropriate. All positive nonzero values are considered within the aperture; any pixels with zero, negative, or NaN values will be considered outside the aperture, and set equal to the 'outside' parameter value. If this parameter is not set, the aperture will be inferred from the finite (i.e. non-NaN) pixels in the OPD array.

nterms
[int] Number of terms to use. (Default: 15)

basis
[callable, optional] Callable (e.g. a function) that generates a sequence of basis arrays given arguments `nterms`, `npix`, and `outside`. Default is `poppy.zernike.zernike_basis`.

Additional keyword arguments to this function are passed through to the `basis` callable.

Note: Recovering coefficients used to generate synthetic/test data depends greatly on the sampling (as one might expect). Generating test data using zernike_basis with npix=256 and passing the result through opd_expand reproduces the input coefficients within <0.1%.

coeffs
[list] List of coefficients (of length `nterms`) from which the input OPD map can be constructed in the given basis. (No additional unit conversions are performed. If the input wavefront is in waves, coeffs will be in waves.) Note that the first coefficient (element 0 in Python indexing) corresponds to the Z=1 Zernike piston term, and so on.

opd_expand_nonorthonormal

poppy.zernike.**opd_expand_nonorthonormal**(*opd*, *aperture=None*, *nterms=15*, *basis=<functools._lru_cache_wrapper object>*, *iterations=5*, *verbose=False*, ***kwargs*)

Modified version of opd_expand, for cases where the basis function is *not* orthonormal, for instance using the regular Zernike functions on obscured apertures.

This version subtracts off each term as it is fit, to avoid over-fitting the same WFE multiple times. It also iterates the fitting multiple times by re-fitting the residuals, in order to allow for capturing any WFE which is missed by the first pass at fitting.

Based on various empirical experimentation for what is necessary to get reasonable behavior in this non-ideal case. Factors to consider:

1) Masking to use just pixels good in both the zernike unit circle and the asymmetric numerical aperture
2) Subtracting off the fit terms as you go, so as to not fit the same WFE multiple times
3) Iterating multiples by re-fitting the residual, to include as much WFE as possible.

opd
[2d ndarray] the OPD you want to fit

aperture
[2D numpy array, optional] Aperture mask for which pixels are included within the aperture. All positive nonzero values are considered within the aperture; any pixels with zero, negative, or NaN values will be considered outside the aperture, and set equal to the 'outside' parameter value. If this parameter is not set, the aperture will be inferred from the finite (i.e. non-NaN) pixels in the OPD array.

nterms
[int] number of terms to fit

basis
[function] which basis function to use. Defaults to Zernike

iterations
[int] Number of iterations for convergence. Default is 5

opd_expand_segments

poppy.zernike.**opd_expand_segments**(*opd*, *aperture=None*, *nterms=15*, *basis=None*, *iterations=2*, *verbose=False*, ***kwargs*)

Expand OPD into a basis defined by segments, typically with piston, tip, & tilt of each.

Similar algorithm as opd_expand_nonorthonormal, but adjusted slightly for spatially disjoint basis vectors, and also for different expected normalization of the piston and tip/tilt basis terms.

The segment_piston_basis and segment_ptt_basis functions are intended for use with this, but it should be generally applicable to higher order hexikes or zernikes defined per segment as well.

Rather than supplying directly e.g. the segment_ptt_basis function with its default parameters, you will likely want to provide a custom basis wrapper function that sets the number of segments, segment size, etc. as appropriate for your chosen segmented aperture.

opd
[2D numpy.ndarray] The wavefront OPD map to expand in terms of the requested basis. Must be square.

aperture
[2D numpy.ndarray, optional] Aperture mask for which pixels are included within the aperture. NOTE - this is handed through to the basis function (see basis parameter) which is responsible for implementing this masking, if appropriate. All positive nonzero values are considered within the aperture; any pixels with zero, negative, or NaN values will be considered outside the aperture, and set equal to the 'outside' parameter value. If this parameter is not set, the aperture will be inferred from the finite (i.e. non-NaN) pixels in the OPD array.

nterms
[int] Number of terms to use. (Default: 15)

basis
[callable, optional] Callable (e.g. a function) that generates a sequence of basis arrays given arguments `nterms`, `npix`, and `outside`. This should be an instance of Segment_Piston_Basis() or Segment_PTT_Basis(), or an equivalent.

opd_from_zernikes

poppy.zernike.**opd_from_zernikes**(*coeffs*, *basis=<functools._lru_cache_wrapper object>*, *aperture=None*, *outside=nan*, ***kwargs*)
Synthesize an OPD from a set of coefficients

coeffs
[list or ndarray] Coefficients for the Zernike terms

basis
[callable] Which basis set. Defaults to Zernike

aperture
[2D ndarray, optional] Aperture mask for which pixels are included within the aperture. All positive nonzero values are considered within the aperture; any pixels with zero, negative, or NaN values will be considered outside the aperture, and set equal to the 'outside' parameter value. If this parameter is not set, the aperture will be default to the 2D unit circle.

outside
[float] Value for pixels outside the specified aperture. Default is `np.nan`, but you may also find it useful for this to be 0.0 sometimes.

Other parameters are supported via **kwargs, in particular setting the size of the OPD via npix.

opd = opd_from_zernikes([0,0,-5,1,0,4,0,8], npix=512)

str_zernike

poppy.zernike.**str_zernike**(*n*, *m*)
Return analytic expression for a given Zernike in LaTeX syntax

zern_name

poppy.zernike.**zern_name**(*i*)

Return a human-readable text name corresponding to some Zernike term as specified by j, the index

Only works up to term 22, i.e. 5th order spherical aberration.

zernike

poppy.zernike.**zernike**(*n*, *m*, *npix=100*, *rho=None*, *theta=None*, *outside=nan*, *noll_normalize=True*, ***kwargs*)

Return the Zernike polynomial Z[m,n] for a given pupil.

For this function the desired Zernike is specified by 2 indices m and n. See zernike1 for an equivalent function in which the polynomials are ordered by a single index.

You may specify the pupil in one of two ways:

zernike(n, m, npix)
where npix specifies a pupil diameter in pixels. The returned pupil will be a circular aperture with this diameter, embedded in a square array of size npix*npix.

zernike(n, m, rho=r, theta=theta)
Which explicitly provides the desired pupil coordinates as arrays r and theta. These need not be regular or contiguous.

The expressions for the Zernike terms follow the normalization convention of Noll et al. JOSA 1976 unless the `noll_normalize` argument is False.

n, m
[int] Zernike function degree

npix
[int] Desired diameter for circular pupil. Only used if `rho` and `theta` are not provided.

rho, theta
[array_like] Image plane coordinates. `rho` should be 0 at the origin and 1.0 at the edge of the circular pupil. `theta` should be the angle in radians.

outside
[float] Value for pixels outside the circular aperture (rho > 1). Default is `np.nan`, but you may also find it useful for this to be 0.0 sometimes.

noll_normalize
[bool] As defined in Noll et al. JOSA 1976, the Zernike definition is modified such that the integral of Z[n, m] * Z[n, m] over the unit disk is pi exactly. To omit the normalization constant, set this to False. Default is True.

zern
[2D numpy array] Z(m,n) evaluated at each (rho, theta)

zernike1

poppy.zernike.**zernike1**(*j*, ***kwargs*)

Return the Zernike polynomial Z_j for pupil points {r,theta}.

For this function the desired Zernike is specified by a single index j. See zernike for an equivalent function in which the polynomials are ordered by two parameters m and n.

Note that there are multiple contradictory conventions for labeling Zernikes with one single index. We follow that of Noll et al. JOSA 1976.

j
[int] Zernike function ordinate, following the convention of Noll et al. JOSA 1976

Additional arguments are defined as in `poppy.zernike.zernike`.

zern
[2D numpy array] Z_j evaluated at each (rho, theta)

zernike_basis

poppy.zernike.**zernike_basis**(*nterms=15*, *npix=512*, *rho=None*, *theta=None*, ***kwargs*)
Return a cube of Zernike terms from 1 to N each as a 2D array showing the value at each point. (Regions outside the unit circle on which the Zernike is defined are initialized to zero.)

nterms
[int, optional] Number of Zernike terms to return, starting from piston. (e.g. `nterms=1` would return only the Zernike piston term.) Default is 15.

npix
[int] Desired pixel diameter for circular pupil. Only used if `rho` and `theta` are not provided.

rho, theta
[array_like] Image plane coordinates. `rho` should be 0 at the origin and 1.0 at the edge of the circular pupil. `theta` should be the angle in radians.

Other parameters are passed through to `poppy.zernike.zernike` and are documented there.

arbitrary_basis

poppy.zernike.**arbitrary_basis**(*aperture*, *nterms=15*, *rho=None*, *theta=None*, *outside=nan*)
Orthonormal basis on arbitrary aperture, via Gram-Schmidt

Return a cube of Zernike-like terms from 1 to N, calculated on an arbitrary aperture, each as a 2D array showing the value at each point. (Regions outside the unit circle on which the Zernike is defined are initialized to zero.)

This implements Gram-Schmidt orthonormalization numerically, starting from the regular Zernikes, to generate an orthonormal basis on some other aperture

aperture
[2D array_like] 2D array representing the arbitrary aperture. All positive nonzero values are considered within the aperture; any pixels with zero, negative, or NaN values will be considered outside the aperture, and set equal to the 'outside' parameter value.

nterms
[int, optional] Number of Zernike terms to return, starting from piston. (e.g. `nterms=1` would return only the Zernike piston term.) Default is 15.

rho, theta
[array_like, optional] Image plane coordinates. `rho` should be 0 at the origin and 1.0 at the edge of the pupil. `theta` should be the angle in radians.

outside
[float] Value for pixels outside the specified aperture. Default is `np.nan`, but you may also find it useful for this to be 0.0 sometimes.

12.2.2 Classes

Segment_Piston_Basis([rings, flattoflat, ...])	Eigenbasis of segment pistons, tips, tilts.
Segment_PTT_Basis([rings, flattoflat, gap, ...])	Eigenbasis of segment pistons, tips, tilts.

Segment_Piston_Basis

class poppy.zernike.**Segment_Piston_Basis**(*rings=2*, *flattoflat=<Quantity 1. m>*, *gap=<Quantity 1. cm>*, *center=False*, *pupil_diam=None*)

Bases: poppy.zernike.Segment_PTT_Basis

Eigenbasis of segment pistons, tips, tilts. (Or of pistons only using the Segment_Piston_Basis subclass.)

The aperture geometry is specified identically to the MultiHexagonAperture class. Set that when creating an instance of this class, then you can call the resulting function object to generate a basis set with the desired sampling, or pass it to the opd_from_zernikes or opd_expand_segments functions.

The basis is generated over a square array that exactly circumscribes the hexagonal aperture.

rings
[int] Number of rings of segments

flattoflat
[float or astropy.Quantity length] Size of a single segment

gap
[float or astropy.Quantity length] Gap between adjacent segments

center
[bool] Include the center segment?

pupil_diam
[float oar astropy.Quantity length] Diameter of the array on which to generate the basis; by default this is chosen to circumscribe the multihex aperture given the specified segment and gap sizes and number of segments.

Methods Summary

__call__([nterms, npix, outside])	Generate piston-only basis ndarray for the specified aperture

Methods Documentation

__call__(*nterms=None*, *npix=512*, *outside=nan*)
Generate piston-only basis ndarray for the specified aperture

nterms
[int] Number of terms. Set to 3x the number of segments.

npix
[int] Size, in pixels, of the aperture array.

outside
[float] Value for pixels outside the specified aperture. Default is `np.nan`, but you may also find it useful for this to be 0.0 sometimes.

Segment_PTT_Basis

class poppy.zernike.**Segment_PTT_Basis**(*rings=2*, *flattoflat=<Quantity 1. m>*, *gap=<Quantity 1. cm>*, *center=False*, *pupil_diam=None*)

Bases: object

Eigenbasis of segment pistons, tips, tilts. (Or of pistons only using the Segment_Piston_Basis subclass.)

The aperture geometry is specified identically to the MultiHexagonAperture class. Set that when creating an instance of this class, then you can call the resulting function object to generate a basis set with the desired sampling, or pass it to the opd_from_zernikes or opd_expand_segments functions.

The basis is generated over a square array that exactly circumscribes the hexagonal aperture.

rings
[int] Number of rings of segments

flattoflat
[float or astropy.Quantity length] Size of a single segment

gap
[float or astropy.Quantity length] Gap between adjacent segments

center
[bool] Include the center segment?

pupil_diam
[float oar astropy.Quantity length] Diameter of the array on which to generate the basis; by default this is chosen to circumscribe the multihex aperture given the specified segment and gap sizes and number of segments.

Methods Summary

__call__([nterms, npix, outside])	Generate PTT basis ndarray for the specified aperture
aperture([npix])	Return the overall aperture across all segments

Methods Documentation

__call__(*nterms=None*, *npix=512*, *outside=nan*)
Generate PTT basis ndarray for the specified aperture

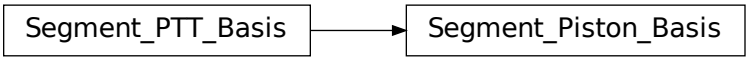
nterms
[int] Number of terms. Set to 3x the number of segments.

npix
[int] Size, in pixels, of the aperture array.

outside
[float] Value for pixels outside the specified aperture. Default is np.nan, but you may also find it useful for this to be 0.0 sometimes.

aperture(*npix=512*)
Return the overall aperture across all segments

12.2.3 Class Inheritance Diagram



CHAPTER 13

About POPPY

13.1 Acknowledging or citing POPPY

If you use this package for work or research presented in a publication (whether directly, or as a dependency to another package), we would be grateful if you could include the following acknowledgment:

> This research made use of POPPY, an open-source optical propagation Python package originally developed for the James Webb Space Telescope project (Perrin, 2012).

You may also cite the software code record at the Astrophysics Source Code Library.

13.2 The Team

POPPY is developed primarily by a team of astronomers at the Space Telescope Science Insitute, but is open to contributions from scientists and software developers around the world. Development takes place on Github at http://github.com/spacetelescope/poppy . See that page for the most up-to-date list of contributors.

13.2.1 Direct contributors to POPPY code

- Marshall Perrin (@mperrin)
- Joseph Long (@josephoenix)
- Ewan Douglas (@douglase)
- Shannon Osborne (@shanosborne)
- Neil Zimmerman (@neilzim)
- Anand Sivaramakrishnan (@anand0xff)
- Maciek Grochowicz (@maciekgroch)
- Phillip Springer (@daphil)

- Ted Corcovilos (@corcoted)
- Kyle Douglass (@kmdouglas)
- Christine Slocum (@cslocum)
- Matt Mechtley (@mmechtley)
- Scott Will (@sdwill)

13.2.2 Indirect Contributors (algorithms, advice, ideas, inspirations)

- Erin Elliot
- Remi Soummer
- Laurent Pueyo
- Charles Lajoie
- Mark Sienkiewicz
- Perry Greenfield

13.2.3 Contributors via Astropy Affiliated Package Template

- Mike Droetboom
- Erik Bray
- Erik Tollerud
- Tom Robitaille
- Matt Craig
- Larry Bradley
- Kyle Barbary
- Christoph Deil
- Adam Ginsburg
- Wolfgang Kerzendorf
- Hans Moritz Gunther
- Benjamin Weaver
- Brigitta Sipocz

This work was supported in part by the JWST mission, a joint effort of NASA, ESA, and CSA, and by the WFIRST Phase-A mission development project. STScI is operated on behalf of NASA by the Association of Universities for Research in Astronomy.

13.3 License

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. The name of AURA and its representatives may not be used to endorse or promote products derived from this software without specific prior written permission.

CHAPTER 14

Appendix A: Optimizing Performance and Parallelization

Performance optimization on modern multi-core machines is a complex subject.

14.1 Know your Linear Algebra Library

The optical propagation calculations in POPPY are dominated by a small number of matrix algebra calls, primarily `np.dot` with a side order of `np.outer`. `numpy` relies heavily on ATLAS (A Tuned LaPACK and BLAS library) to perform its linear algebra calculations, but for reasons of portability many distributed copies of numpy do not have an ATLAS that has been compiled with all the CPU optimization tricks turned on.

Whether or not `numpy` is linked against an optimized linear algebra library can make a **huge** difference in execution speed, with speedups of an **order of magnitude or more**. You definitely want to make sure that your numpy is using such a library:

- Apple's Accelerate framework (a.k.a. vecLib) provides a highly tuned copy of BLAS and LAPACK on any Mac, right out of the box.
- OpenBLAS is recommended on Linux.
- The Intel Math Kernel Library (MKL) Optimizations is available as an add-on from Continuum Analytics to their Anaconda Python distribution. This requires a commercial license for a small fee.

Numpy is statically linked at compile time against a given copy of BLAS. Switching backends generally requires recompiling numpy. (Note that if you use MacPorts on Mac OS to install numpy, it automatically uses Apple's Accelerate framework for you. Nice!)

Various stackoverflow, quora, and twitter posts suggest that OpenBLAS, MKL, and Accelerate all have very similar performance, so as long as your `numpy` is using one of those three you should be in good shape.

14.2 Parallelized Calculations

POPPY can parallelize calculations in two different ways:

1. Using Python's built-in `multiprocessing` package to launch many additional Python processes, each of which calculates a different wavelength.
2. Using the FFTW library for optimized accellerated Fourier transform calculations. FFTW is capable of sharing load across multiple processes via multiple threads.

One might think that using both of these together would result in the fastest possible speeds. However, in testing it appears that FFTW does not work reliably with multiprocessing for some unknown reason; this instability manifests itself as occasional fatal crashes of the Python process. For this reason it is recommended that one use *either multiprocessing or FFTW, but not both*.

For most users, the recommended choice is using multiprocessing.

FFTW only helps for FFT-based calculations, such as some coronagraphic or spectroscopic calculations. Calculations that use only discrete matrix Fourier transforms are not helped by FFTW. Furthermore, baseline testing indicates that in many cases, just running multiple Python processes is in fact significantly faster than using FFTW, even for coronagraphic calculations using FFTs.

There is one slight tradeoff in using multiprocessing: When running in this mode, POPPY cannot display plots of the calculation's work in progress, for instance the intermediate optical planes. (This is because the background Python processes can't write to any Matplotlib display windows owned by the foreground process.) One can still retrieve the intermediate optical planes after the multiprocess calculation is complete and examine them then; you just can't see plots displayed on screen as the calculation is proceeding. Of course, the calculation ought to proceed faster overall if you're using multiple processes!

Warning: On Mac OS X, for Python < 2.7, multiprocessing is not compatible with Apple's Accelerate framework mentioned above, due to the non-POSIX-compliant manner in which multiprocessing forks new processes. See https://github.com/spacetelescope/poppy/issues/23 and https://github.com/numpy/numpy/issues/5752 for discussion. Python 3.4 provides an improved method of starting new processes that removes this limitation.

If you want to use multiprocessing with Apple's Accelerate framework, you must upgrade to Python 3.4+. POPPY will raise an exception if you try to start a multiprocess calculation and numpy is linked to Accelerate on earlier versions of Python.

This is likely related to the intermittent crashes some users have reported with multiprocessing and FFTW; that combination may also prove more stable on Python 3.4 but this has not been extensively tested yet.

The configuration options to enable multiprocessing live under `poppy.conf`, and use the Astropy configuration framework. Enable them as follows:

```
>>> import poppy
>>> poppy.conf.use_multiprocessing = True
>>> poppy.conf.use_fftw = False
```

One caveat with running multiple processes is that the memory demands can become substantial for large oversampling factors. For instance, a 1024-pixel-across pupil with `oversampling=4` results in arrays that are 256 MB each. Several such arrays are needed in memory per calculation, with peak memory utilization reaching ~ 1 GB per process for `oversampling=4` and over 4 GB per process for `oversamping=8`.

Thus, if running on a 16-core computer, ensure at least 32 GB of RAM are available before using one process per core. If you are constrained by the amount of RAM available, you may experience better performance using fewer processes than the number of processor cores in your computer.

By default, POPPY attempts to automatically choose a number of processes based on available CPUs and free memory that will optimize computation speed without exhausting available RAM. This is implemented in the function `poppy.utils.estimate_optimal_nprocesses()`.

If desired, the number of processes can be explicitly specified:

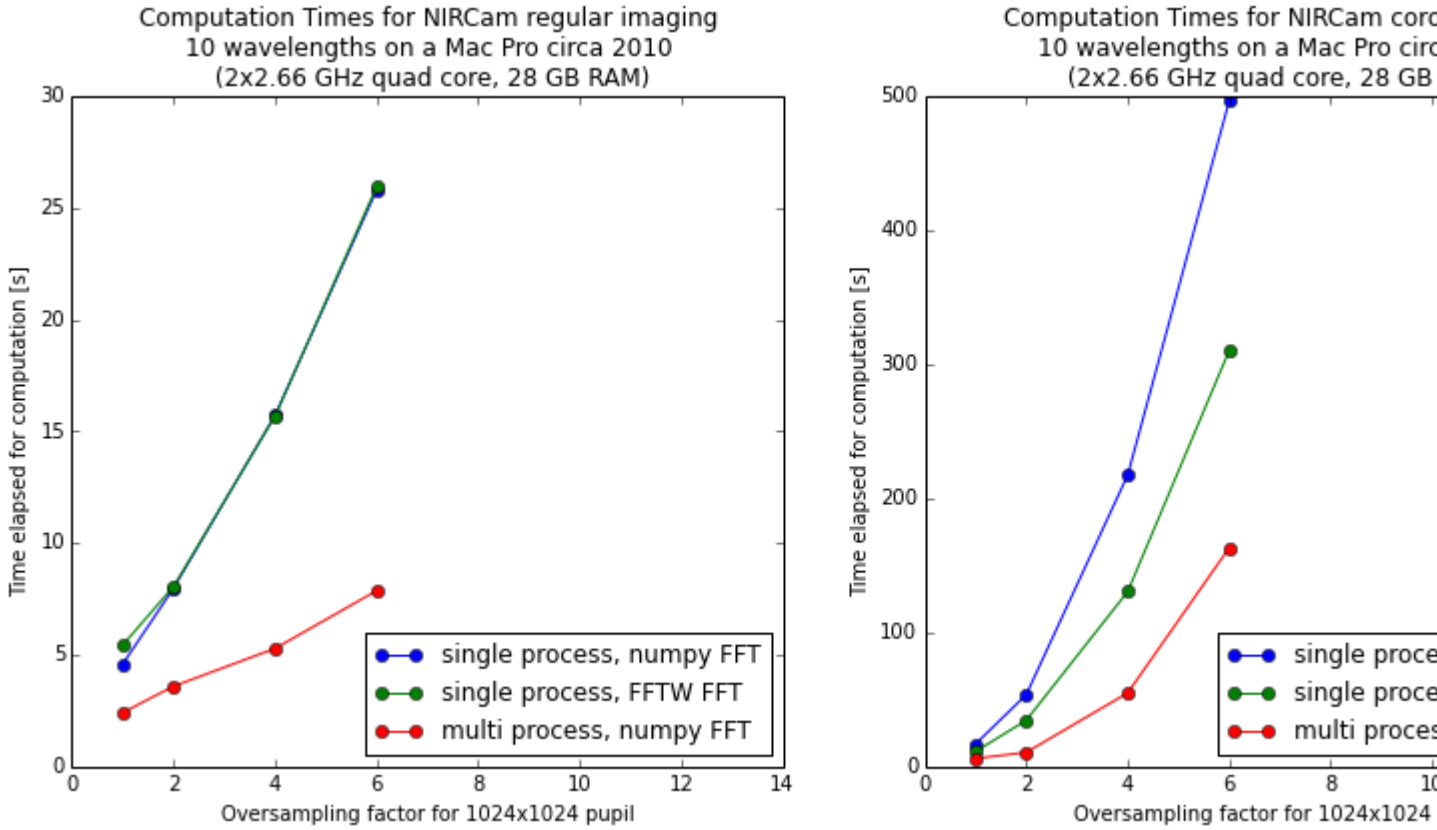
```
>>> poppy.conf.n_processes = 5
```

Set this to zero to enable automatic selection via the `estimate_optimal_nprocesses()` function.

14.3 Comparison of Different Parallelization Methods

The following figure shows the comparison of single-process, single-process with FFTW, and multi-process calculations on a relatively high end 16-core Mac Pro. The calculations were done with WebbPSF, a PSF simulator for JWST that uses POPPY to perform computations.

The horizontal axis shows increasing detail of calculation via higher oversampling, while the vertical axis shows computation time. Note the very different Y-axis scales for the two figures; coronagraphic calculations take much longer than direct imaging!



Using multiple Python processes is the clear winner for most workloads. Explore the options to find what works best for your particular calculations and computer setup.

CHAPTER 15

Appendix B: Optimizing FFT Performance with FFTW

Optimizing numerical performance of FFTs is a complicated subject. Just using the FFTW library is no guarantee of optimal performance; you need to know how to configure it.

Note: The following tests were performed using the older PyFFTW3 package, and have not yet been updated for the newer pyFFTW package. However, performance considerations are expected to be fairly similar for both packages since the underlying FFTW library is the same.

See discussion and test results at https://github.com/spacetelescope/webbpsf/issues/10

This is probably fairly sensitive to hardware details. The following benchmarks were performed on a Mac Pro, dual quad-core 2.66 GHz Xeon, 12 GB RAM.

- Unlike many of the array operations in `numpy`, the `fft` operation is not threaded for execution across multiple processors. It is thus slow and inefficient.
- Numpy and Scipy no longer include FFTW, but luckily there is an independently maintained pyfftw3 module. See https://launchpad.net/pyfftw/
- Using pyfftw3 can result in a 3-4x speedup for moderately large arrays. However, there are two significant gotchas to be aware of:

1) the pyfftw3.Plan() function has a default parameter of nthreads=1. You have to explicitly tell it to use multiple threads if you wish to reap the rewards of multiple processors. By default with nthreads=1 it is in fact a bit slower than numpy.fft!
2) The FFTW3 documentation asserts that greater speed can be achieved by using arrays which are aligned in memory to 16-byte boundaries. There is a fftw3.create_aligned_array() function that created numpy arrays which have this property. While I expected using this would make the transforms faster, in fact I see significantly better performance when using unaligned arrays. (The speed difference becomes larger as array size increases, up to 2x!) This is unexpected and not understood, so it may vary by machine and I suggest one ought to test this on different machines to see if it is reliable.

15.1 Planning in FFTW3

- Performing plans can take a *long* time, especially if you select exhaustive or patient modes:
- The default option is 'estimate' which turns out to be really a poor choice.
- It appears that you can get most of the planning benefit from using the 'measure' option.
- Curiously, the really time-consuming planning only appears to take place if you do use aligned arrays. If you use regular unaligned arrays, then a very abbreviated planning set is performed, and yet you still appear to reap most of the benefits of

15.2 A comparison of different FFT methods

This test involves, in each iteration, allocating a new numpy array filled with random values, passing it to a function, FFTing it, and then returning the result. Thus it is a fairly realistic test but takes longer per iteration than some of the other tests presented below on this page. This is noted here in way of explanation for why there are discrepant values for how long an optimized FFT of a given size takes.

Test results:

```
Doing complex FFT with array size = 1024 x 1024
   for         numpy fft, elapsed time is: 0.094331 s
   for             fftw3, elapsed time is: 0.073848 s
   for    fftw3 threaded, elapsed time is: 0.063143 s
   for fftw3 thr noalign, elapsed time is: 0.020411 s
   for fftw3 thr na inplace, elapsed time is: 0.017340 s
Doing complex FFT with array size = 2048 x 2048
   for         numpy fft, elapsed time is: 0.390593 s
   for             fftw3, elapsed time is: 0.304292 s
   for    fftw3 threaded, elapsed time is: 0.224193 s
   for fftw3 thr noalign, elapsed time is: 0.061629 s
   for fftw3 thr na inplace, elapsed time is: 0.047997 s
Doing complex FFT with array size = 4096 x 4096
   for         numpy fft, elapsed time is: 2.190670 s
   for             fftw3, elapsed time is: 1.911555 s
   for    fftw3 threaded, elapsed time is: 1.414653 s
   for fftw3 thr noalign, elapsed time is: 0.332999 s
   for fftw3 thr na inplace, elapsed time is: 0.293531 s
```

Conclusions: It appears that the most efficient algorithm is a non-aligned in-place FFT. Therefore, this is the algorithm adopted into POPPY.

In this case, it makes sense that avoiding the alignment is beneficial, since it avoids a memory copy of the entire array (from regular python unaligned into the special aligned array). Another set of tests (results not shown here) indicated that there is no gain in performance from FFTing from an unaligned input array to an aligned output array.

15.3 A test comparing all four planning methods

This test involves creating one single input array (specifically, a large circle in the central half of the array) and then repeatedly FFTing that same array. Thus it is pretty much the best possible case and the speeds are very fast.

```
For arrays of size 512x512
Building input circular aperture
        that took 0.024070 s
 Plan method= estimate
        Array alignment True            False
        Planning took   0.041177        0.005638 s
        Executing took  0.017639        0.017181 s
 Plan method= measure
        Array alignment True            False
        Planning took   0.328468        0.006960 s
        Executing took  0.001991        0.002741 s
 Plan method= patient
        Array alignment True            False
        Planning took   39.816985       0.020944 s
        Executing took  0.002081        0.002475 s
 Plan method= exhaustive
        Array alignment True            False
        Planning took   478.421909      0.090302 s
        Executing took  0.004974        0.002467 s
```

15.4 A comparison of 'estimate' and 'measure' for different sizes

This test involves creating one single input array (specifically, a large circle in the central half of the array) and then repeatedly FFTing that same array. Thus it is pretty much the best possible case and the speeds are very fast.

```
For arrays of size 1024x1024
Building input circular aperture
        that took 0.120378 s
 Plan method= estimate
        Array alignment True            False
        Planning took   0.006557        0.014652 s
        Executing took  0.041282        0.041586 s
 Plan method= measure
        Array alignment True            False
        Planning took   1.434870        0.015797 s
        Executing took  0.008814        0.011852 s

For arrays of size 2048x2048
Building input circular aperture
        that took 0.469819 s
 Plan method= estimate
        Array alignment True            False
        Planning took   0.006753        0.032270 s
        Executing took  0.098976        0.098925 s
 Plan method= measure
        Array alignment True            False
        Planning took   5.347839        0.033213 s
        Executing took  0.028528        0.047729 s

For arrays of size 4096x4096
Building input circular aperture
        that took 2.078152 s
 Plan method= estimate
        Array alignment True            False
        Planning took   0.007102        0.056571 s
```

(continues on next page)

(continued from previous page)

```
        Executing took  0.395048        0.326832 s
 Plan method= measure
        Array alignment True            False
        Planning took   17.890278       0.057363 s
        Executing took  0.126414        0.133602 s

For arrays of size 8192x8192
Building input circular aperture
        that took 93.043509 s
 Plan method= estimate
        Array alignment True            False
        Planning took   0.245359        0.425931 s
        Executing took  2.800093        1.426851 s
 Plan method= measure
        Array alignment True            False
        Planning took   41.203768       0.235688 s
        Executing took  0.599916        0.526022 s
```

15.5 Caching of plans means that irunning the same script a second time is much faster

Immediately after executing the above, I ran the same script again. Now the planning times all become essentially negligible.

Oddly, the exection time for the largest array gets longer. I suspect this has something to do with memory or system load.

```
For arrays of size 1024x1024
Building input circular aperture
        that took 0.115704 s
 Plan method= estimate
        Array alignment True            False
        Planning took   0.005147        0.015813 s
        Executing took  0.006883        0.011428 s
 Plan method= measure
        Array alignment True            False
        Planning took   0.009078        0.012562 s
        Executing took  0.007057        0.010706 s

For arrays of size 2048x2048
Building input circular aperture
        that took 0.421966 s
 Plan method= estimate
        Array alignment True            False
        Planning took   0.004888        0.032564 s
        Executing took  0.026869        0.043273 s
 Plan method= measure
        Array alignment True            False
        Planning took   0.019813        0.032273 s
        Executing took  0.027532        0.045452 s

For arrays of size 4096x4096
Building input circular aperture
        that took 1.938918 s
```

(continues on next page)

(continued from previous page)

```
 Plan method= estimate
        Array alignment True            False
        Planning took   0.005327        0.057813 s
        Executing took  0.123481        0.131502 s
 Plan method= measure
        Array alignment True            False
        Planning took   0.030474        0.057851 s
        Executing took  0.119786        0.134453 s

For arrays of size 8192x8192
Building input circular aperture
        that took 78.352433 s
 Plan method= estimate
        Array alignment True            False
        Planning took   0.020330        0.325254 s
        Executing took  0.593469        0.530125 s
 Plan method= measure
        Array alignment True            False
        Planning took   0.147264        0.227571 s
        Executing took  4.640368        0.528359 s
```

15.6 The Payoff: Speed improvements in POPPY

For a monochromatic propagation through a 1024x1024 pupil, using 4x oversampling, using FFTW results in about a 3x increase in performance.

```
Using FFTW:         FFT time elapsed:      0.838939 s
Using Numpy.fft:    FFT time elapsed:      3.010586 s
```

This leads to substantial savings in total computation time:

```
Using FFTW:          TIME 1.218268 s for propagating one wavelength
Using Numpy.fft:     TIME 3.396681 s for propagating one wavelength
```

Users are encouraged to try different approaches to optimizing performance on their own machines. To enable some rudimentary benchmarking for the FFT section of the code, set `poppy.conf.enable_speed_tests=True` and configure your logging display to show debug messages. (i.e. `webbpsf.configure_logging('debug')`). Measured times will be printed in the log stream, for instance like so:

```
poppy     : INFO     Calculating PSF with 1 wavelengths
poppy     : INFO      Propagating wavelength = 1e-06 meters  with weight=1.00
poppy     : DEBUG    Creating input wavefront with wavelength=0.000001, npix=511, pixel scale=0.007828␣
↪meters/pixel
poppy     : DEBUG      Wavefront and optic Optic from fits.HDUList object already at same plane type,␣
↪no propagation needed.
poppy     : DEBUG      Multiplied WF by phasor for Pupil plane: Optic from fits.HDUList object
poppy     : DEBUG    normalizing at first plane (entrance pupil) to 1.0 total intensity
poppy     : DEBUG      Propagating wavefront to Image plane: -empty- (Analytic).
poppy     : DEBUG    conf.use_fftw is True
poppy     : INFO     using numpy FFT of (511, 511) array
poppy     : DEBUG    using numpy FFT of (511, 511) array, direction=forward
poppy     : DEBUG       TIME 0.051085 s  for the FFT                                    # This line
poppy     : DEBUG      Multiplied WF by phasor for Image plane: -empty- (Analytic)
poppy     : DEBUG       TIME 0.063745 s for propagating one wavelength                  # and this one
```

(continues on next page)

(continued from previous page)

```
poppy     : INFO       Calculation completed in 0.082 s
poppy     : INFO     PSF Calculation completed.
```

Part III

Getting Help

POPPY is developed and maintained by *Marshall Perrin and collaborators*. Questions, comments, and pull requests always welcome, either via the Github repository or email to help@stsci.edu.

Python Module Index

p

Index

Symbols

A

B

C

D

F

G

H

I

M

N

O

P

Q

R

S

T

W

Z