webbpsf Documentation

Release 0.8.0

Association of Universities for Research in Astronomy

Dec 14, 2018

Contents

WebbPSF is a Python package that computes simulated point spread functions (PSFs) for NASA's JWST and WFIRST observatories. WebbPSF transforms models of telescope and instrument optical state into PSFs, taking into account detector pixel scales, rotations, filter profiles, and point source spectra. It is *not* a full optical model of JWST, but rather a tool for transforming optical path difference (OPD) maps, created with some other tool, into the resulting PSFs as observed with JWST's or WFIRST's instruments.

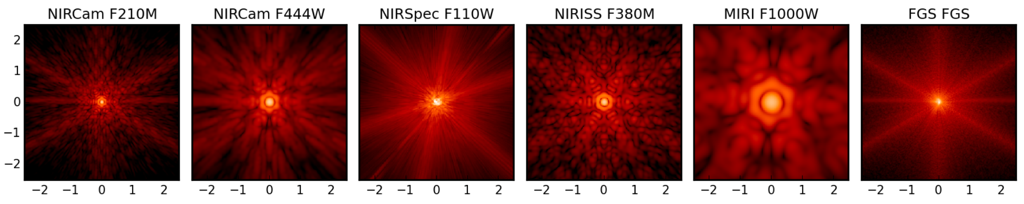


Fig. 1: Sample PSFs for JWST's instrument suite, all on the same angular scale and display stretch.

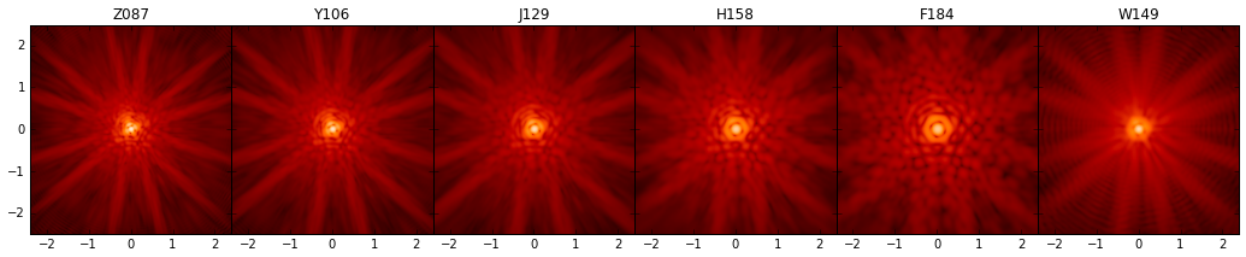


Fig. 2: Sample PSFs for the filters in the WFIRST WFI.

What this software does:

- Uses OPD maps precomputed by detailed optical simulations of JWST and WFIRST, and in the case of JWST based on instrument and telescope flight hardware cryo-vacuum test results.
- For JWST, computes PSF images with requested properties for any of JWST's instruments. Supports imaging, coronagraphy, and most spectrographic modes with all of JWST's instruments. IFUs are yet to come.
- For WFIRST, computes PSFs with the Wide Field Imager, based on recent GSFC optical models, including field- and wavelength-dependent aberrations. A preliminary version of the Coronagraph Instrument is also available.
- Provides a suite of tools for quantifying PSF properties such as FWHM, Strehl ratio, etc.

What this software does NOT do:

- Contain in itself any detailed thermal or optical model of JWST or WFIRST. For the results of end-to-end integrated simulations of JWST, see for instance Predicted JWST imaging performance (Knight, Lightsey, & Barto; Proc. SPIE 2012). For WFIRST modeling, see the WFIRST Reference Info page
- Model spectrally dispersed PSFs produced by any of the spectrograph gratings. It does, however, let you produce monochromatic PSFs in these modes, suitable for stitching together into spectra using some other software.
- Model most detector effects such as pixel MTF, intrapixel sensitivity variations, interpixel capacitance, or any noise sources. Add those separately with your favorite detector model code. (*Note, one particularly significant detector scattering for MIRI imaging has now been added.)

Contributors: WebbPSF has been developed by Marshall Perrin, Joseph Long, Neil Zimmerman, Robel Geda, Shannon Osborne, Marcio Melendez Hernandez, Lauren Chambers, and Keira Brooks, with contributions from Jarron Leisenring, Ewan Douglas, Charles Lajoie, Megan Sosey, and the developers of the astropy-helpers template framework.

Part I

Getting Started with WebbPSF

The WebbPSF software system is composed of two Python packages: a lower-level optical propagation library (`POPPY`) plus an implementation of the JWST instruments using that library (`WebbPSF`). This documentation explains the programming interfaces and graphical user interface of WebbPSF, as well as providing a *quick overview* of POPPY.

Quickstart Jupyter Notebook

This documentation is complemented by an Jupyter Notebook format quickstart tutorial. Downloading and running that notebook is a great way to get started using WebbPSF.

What's new in the latest release?

CHAPTER 1

Introduction

Conceptually, this simulation code has three layers of abstraction:

- A base package for wavefront propagation through generic optical systems (provided by `POPPY`)
- Models of the JWST instruments implemented on top of that base system (provided by `WebbPSF`)
- An optional *graphical user interface*

It is entirely possible (and indeed recommended for scripting) to just use the `WebbPSF` interface without the GUI, but the GUI can provide a quicker method for many simple interactive calculations.

1.1 Why WebbPSF?

For any space telescope, an ability to predict the properties of point spread functions (PSFs) is needed before launch for a wide range of preparatory science studies and tool development. Tools for producing high quality model PSFs must be easily accessible to the entire astronomical community. WebbPSF provides an easy-to-use tool for PSF simulations of JWST and WFIRST, in the style of the highly successful "Tiny Tim" PSF simulator for Hubble.

WebbPSF simulations are based on a mixture of observatory design parameters and as-built properties. The software provides a highly flexible and scriptable toolkit in Python for simulating a very wide range of observing modes and science scenarios, using efficient computational methods (including optional parallelization and use of GPUs). WebbPSF is a key building block in higher-level observatory simulators, including the JWST Exposure Time Calculator.

1.2 Algorithms Overview

Read on if you're interested in details of how the computations are performed. Otherwise, jump to *Quick Start*.

The problem at hand is to transform supplied, precomputed OPDs (derived from a detailed optomechanical model of the telescope) into observed PSFs as seen with one or more of JWST's various detectors. This requires knowledge of the location and orientation of the detector planes, the properties of relevant optics such as bandpass filters and/or coronagraphic image and pupil plane masks, and a model of light propagation between them.

Instrumental properties are taken from project documentation and the published literature as appropriate; see the *References* for detailed provenance information. Optics may be described either numerically (for instance, a FITS file containing a mask image for a Lyot plane or a FITS bintable giving a spectral bandpass) or analytically (for instance, a coronagraph occulter described as a circle of a given radius or a band-limited mask function with given free parameters).

WebbPSF computes PSFs under the assumption that JWST's instruments are well described by Fraunhofer diffraction, as implemented using the usual Fourier relationship between optical pupil and image planes (e.g. Goodman et al. 1996). Two specific types of 2D Fourier transform are implemented: a Fast Fourier Transform and a discrete Matrix Fourier Transform.

The familiar Fast Fourier Transform (FFT) algorithm achieves its speed at the cost of imposing a specific fixed relationship between pixel sampling in the pupil and image planes. As a result, obtaining finely sampled PSFs requires transforming very large arrays consisting mostly of zero-padding. A more computationally attractive method is to use a discrete matrix Fourier transform, which provides flexibility to compute PSFs on any desired output sampling without requiring any excess padding of the input arrays. While this algorithm's computational cost grows as `O(N^3)` versus `O(N log N)` for the FFT, the FFT's apparent advantage is immediately lost due to the need to resample the output onto the real pixel grid, which is an `O(N^2)` operation. By performing a matrix fourier transform directly to the desired output pixel scale, we can achieve arbitrarily fine sampling without the use of memory-intensive large padded arrays, and with lower overall computation time.

Further optimizations are available in coronagraphic mode using the semi-analytic coronagraphic propagation algorithm of Soummer et al. 2007. In this approach, rather than propagating the entire wavefront from pupil to image and back to pupil in order to account for the coronagraphic masks, we can propagate only the subset of the wavefront that is actually blocked by the image occulter and then subtract it from the rest of the wavefront at the Lyot plane. This relies on Babinet's principle to achieve the same final PSF with more computational efficiency, particularly for the case of highly oversampled image planes (as is necessary to account for fine structure in image plane occulter masks). See Soummer et al. 2007 for a detailed description of this algorithm.

1.2.1 Types of Fourier Transform Calculation in WebbPSF

- Any direct imaging calculation, any instrument: Matrix DFT
- NIRCam coronagraphy with circular occulters: Semi-Analytic Fast Coronagraphy and Matrix DFT
- NIRCam coronagraphy with wedge occulters: FFT and Matrix DFT
- MIRI Coronagraphy: FFT and Matrix DFT
- NIRISS NRM, GR799XD: Matrix DFT
- NIRSpec and NIRISS slit spectroscopy: FFT and Matrix DFT

See Optimizing Performance and Parallelization in the POPPY documentation for more details on calculation performance.

1.3 Getting WebbPSF

The WebbPSF software is installable through pip, but it depends on data files distributed through STScI. Since there is more than one way to install scientific Python software, the possibilities are covered in *Requirements & Installation*.

The AstroConda distribution includes WebbPSF and its associated data files. If you installed the packages into an environment named `astroconda`, the command to upgrade all STScI software (including WebbPSF) to the latest version would be `conda update --name astroconda stsci`.

For those who prefer to use pip (and have NumPy, SciPy, and matplotlib already installed) the command is:

```
$ pip install -U webbpsf
```

This command installs (or upgrades) WebbPSF to the latest version on PyPI. Before WebbPSF will run, you must *download the WebbPSF data files* and set the `WEBBPSF_DATA` environment variable to point to the place you extracted them. You may also want to *install Pysynphot*, an optional dependency, to improve PSF fidelity.

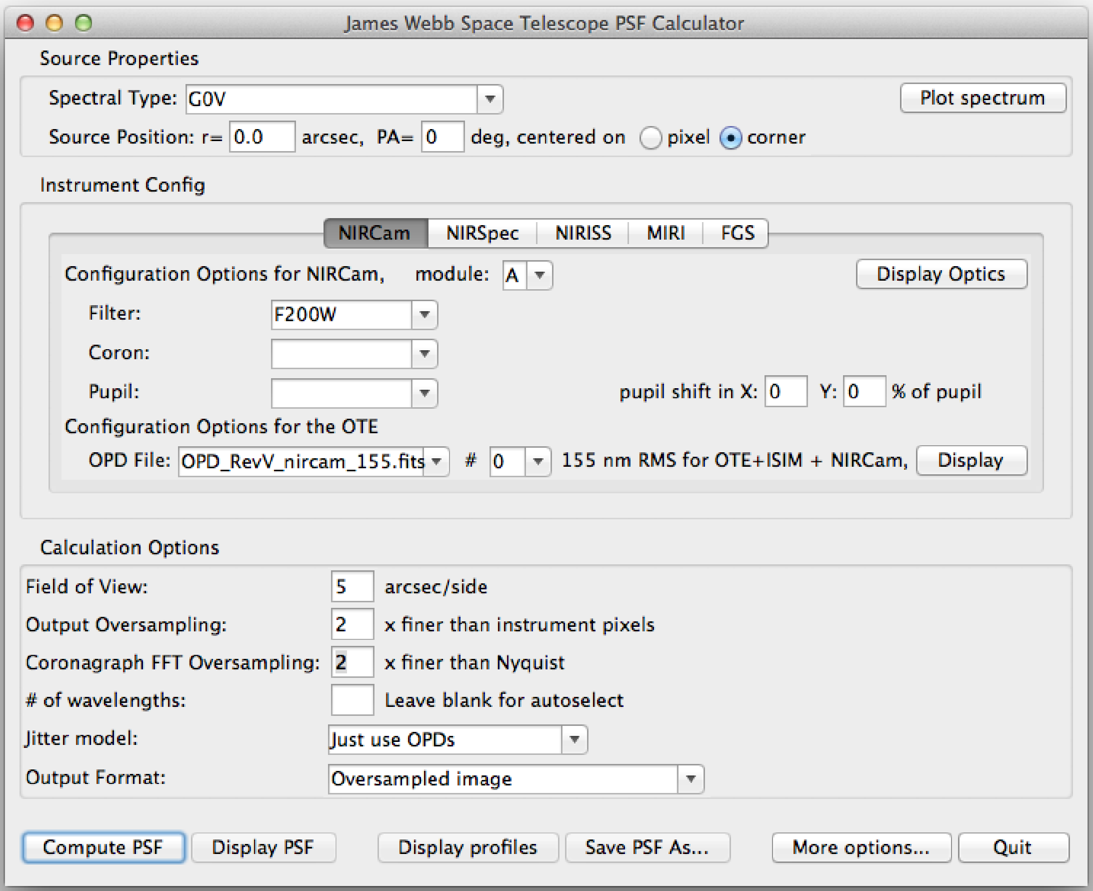
For detailed installation instructions, refer to *Requirements & Installation*. (This document also explains how to install optional dependencies, install supporting data files, *install from GitHub source*, etc.)

1.4 Quick Start

First, download and install the software. Then just start `python` and

```
>>> import webbpsf
>>> webbpsf.gui()
```

and you should be able to test drive things using the GUI:



James Webb Space Telescope PSF Calculator

Source Properties

Spectral Type: G0V

Source Position: r= 0.0 arcsec, PA= 0 deg, centered on pixel [✓ corner]

Instrument Config

Configuration Options for NIRCam, module: A

Filter: F200W

Coron:

Pupil:

pupil shift in X: 0 Y: 0 % of pupil

Configuration Options for the OTE

OPD File: OPD_RevV_nircam_155.fits # 0 155 nm RMS for OTE+ISIM + NIRCam,

Calculation Options

Field of View: 5 arcsec/side

Output Oversampling: 2 x finer than instrument pixels

Coronagraph FFT Oversampling: 2 x finer than Nyquist

# of wavelengths: Leave blank for autoselect

Jitter model: Just use OPDs

Output Format: Oversampled image

Fig. 1: The main window of the WebbPSF GUI when first launched.

Most controls should be self-explanatory, so feel free to experiment. The article *Using WebbPSF via the Graphical User Interface* provides a detailed explanation of the GUI options. (Please note, the GUI only provides *basic* access to WebbPSF for introductory purposes; the Python programming interface is the recommended best way to use WebbPSF.)

WebbPSF can save a detailed log of its calculations and results. This will by default be shown on screen but can also be saved to disk.

```
>>> webbpsf.setup_logging(filename='my_log_file.txt')
```

Log settings are persistent between sessions, so you can just set this once the very first time you start WebbPSF and logging will be enabled thereafter until you explicitly change it.

For further information, consult *Using WebbPSF via the Python API* or *Using WebbPSF via the Graphical User Interface*.

CHAPTER 2

Requirements & Installation

The latest released version of WebbPSF can be installed with the conda package management system or using pip.

2.1 Recommended best method: Installing via AstroConda

For ease of installation, we recommend using AstroConda, an astronomy-optimized software distribution for scientific Python built on Anaconda. Install AstroConda according to their instructions, then activate the environment with:

```
$ source activate astroconda
```

(Note: if you named your environment something other than `astroconda`, change the above command appropriately.)

Next, install WebbPSF (along with all its dependencies and required reference data) with:

```
(astroconda)$ conda install webbpsf
```

Optional: sign up to receive announcement of updates

This is entirely optional, but you may wish to sign up to the mailing list `webbpsf-users@stsci.edu`. This is a low-traffic moderated announce-only list, to which we will periodically post announcements of updates to this software.

To subscribe, visit the maillist.stsci.edu server

2.2 Installing with conda (but no AstroConda)

If you already use `conda`, but do not want to install the full suite of STScI software, you can simply add the AstroConda *channel* and install WebbPSF as follows (creating a new environment named `webbpsf-env`):

```
$ conda config --add channels http://ssb.stsci.edu/astroconda
$ conda create -n webbpsf-env webbpsf
$ source activate webbpsf-env
```

Upgrading to the latest version is done with `conda update -n webbpsf-env --all`.

Warning: You *must* install WebbPSF into a specific environment (e.g. `webbpsf-env`); our conda package will not work if installed into the default "root" environment.

2.3 Installing with pip

WebbPSF and its underlying optical library POPPY may be installed from the Python Package Index in the usual manner for Python packages.

```
$ pip install --upgrade webbpsf
[... progress report ...]

Successfully installed webbpsf
```

Note that `pip install webbpsf` only installs the program code. **If you install via pip, you must manually download and install the data files, as :ref:'described below <data_install>'.** To obtain source spectra for calculations, you should also follow *installation instructions for pysynphot*.

2.4 Installing or updating pysynphot

Pysynphot is an optional dependency, but is highly recommended. Pysynphot is best installed via AstroConda. Further installation instructions can be found here in the POPPY docs.

2.5 Installing the Required Data Files

If you install via pip or manually, you must install the data files yourself. If you install via Conda, the data files are automatically installed, in which case you can skip this section.

Files containing such information as the JWST pupil shape, instrument throughputs, and aperture positions are distributed separately from WebbPSF. To run WebbPSF, you must download these files and tell WebbPSF where to find them using the `WEBBPSF_PATH` environment variable.

1. Download the following file: webbpsf-data-0.8.0.tar.gz [approx. 240 MB]
2. Untar `webbpsf-data-0.8.0.tar.gz` into a directory of your choosing.
3. Set the environment variable `WEBBPSF_PATH` to point to that directory. e.g.

   export WEBBPSF_PATH=$HOME/data/webbpsf-data

for bash. (You will probably want to add this to your `.bashrc`.)

You should now be able to successfully `import webbpsf` in a Python session, or start the GUI with the command `webbpsfgui`.

Warning: If you have previously installed the data files for an earlier version of WebbPSF, and then update to a newer version, the software may prompt you that you must download and install a new updated version of the data files.

For STScI Users Only

Users at STScI may access the required data files from the Central Storage network. Set the following environment variables in your bash shell. (You will probably want to add this to your `.bashrc`.)

```
export WEBBPSF_PATH="/grp/jwst/ote/webbpsf-data"
export PYSYN_CDBS="/grp/hst/cdbs"
```

2.6 Software Requirements

Required Python version: WebbPSF 0.8 and above require Python 3.5 or higher.

Required Python packages:

- Recent versions of NumPy, SciPy and matplotlib, if not installed already.
- Astropy
- POPPY

Recommended Python packages:

- 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*. Pysynphot is recommended for most users.

Optional Python packages:

Some calculations with POPPY can benefit from the optional packages psutil and pyFFTW, but these are not needed in general. See the POPPY installation docs for more details. These optional packages are only worth adding for speed improvements if you are spending substantial time running calculations.

Additional packages are needed for the optional use of GPUs to accelerate calculations. See the POPPY documentation.

2.7 Installing a pre-release version or contributing to WebbPSF development

The WebbPSF source code repository is hosted at GitHub, as is the repository for POPPY. Users may clone or fork in the usual manner. Pull requests with code enhancements welcomed.

To install the current development version of WebbPSF, you can use `pip` to install directly from a `git` repository. To install WebbPSF and POPPY from `git`, uninstall any existing copies of WebbPSF and POPPY, then invoke pip as follows:

```
$ pip install -e git+https://github.com/spacetelescope/poppy.git#egg=poppy \
   -e git+https://github.com/spacetelescope/webbpsf.git#egg=webbpsf
```

This will create directories `./src/poppy` and `./src/webbpsf` in your current directory containing the cloned repository. If you have commit access to the repository, you may want to clone via ssh with a URL like `git+ssh://git@github.com:spacetelescope/webbpsf.git`. Documentation of the available options for installing directly from Git can be found in the pip documentation.

Remember to *install the required data files*, if you have not already installed them.

CHAPTER 3

Using WebbPSF via the Python API

This module provides the primary interface for programmers and for interactive non-GUI use. It provides five classes corresponding to the JWST instruments, with consistent interfaces. See *this page* for the detailed API; for now let's dive into some example code.

Additional code examples are available later in this documentation.

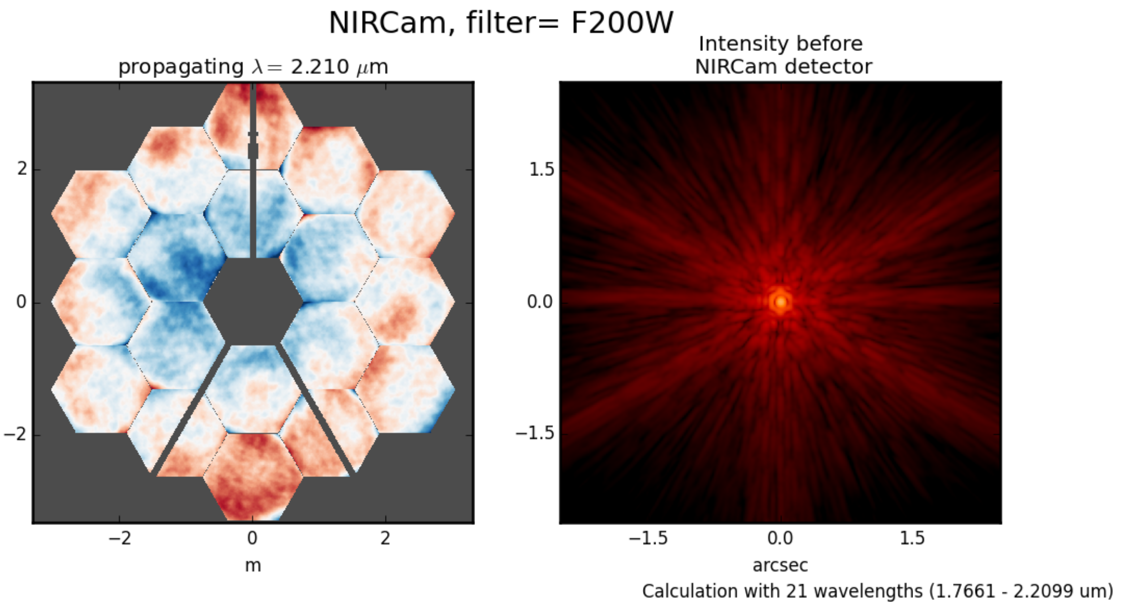
3.1 Usage and Examples

Simple PSFs are easily obtained:

```
>>> import webbpsf
>>> nc = webbpsf.NIRCam()
>>> nc.filter =  'F200W'
>>> psf = nc.calc_psf(oversample=4)      # returns an astropy.io.fits.HDUlist containing PSF and header
>>> plt.imshow(psf[0].data)              # display it on screen yourself, or
>>> webbpsf.display_psf(psf)             # use this convenient function to make a nice log plot with␣
↪labeled axes
>>>
>>> psf = nc.calc_psf(filter='F470N', oversample=4)    # this is just a shortcut for setting the filter,
↪ then computing a PSF
>>>
>>> nc.calc_psf("myPSF.fits", filter='F480M')         # you can also write the output directly to disk␣
↪if you prefer.
```

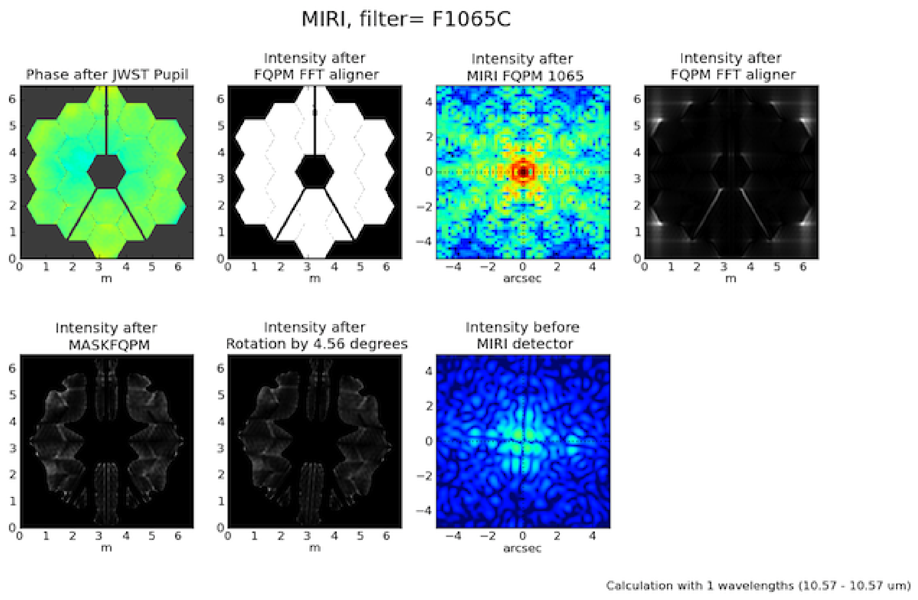
For interactive use, you can have the PSF displayed as it is computed:

```
>>> nc.calc_psf(display=True)                          # will make nice plots with matplotlib.
```



More complicated instrumental configurations are available by setting the instrument's attributes. For instance, one can create an instance of MIRI and configure it for coronagraphic observations, thus:

```
>>> miri = webbpsf.MIRI()
>>> miri.filter = 'F1065C'
>>> miri.image_mask = 'FQPM1065'
>>> miri.pupil_mask = 'MASKFQPM'
>>> miri.calc_psf('outfile.fits')
```



3.1.1 Input Source Spectra

WebbPSF attempts to calculate realistic weighted broadband PSFs taking into account both the source spectrum and the instrumental spectral response.

The default source spectrum is, if `pysynphot` is installed, a G2V star spectrum from Castelli & Kurucz 2004. Without `pysynphot`, the default is a simple flat spectrum such that the same number of photons are detected at each wavelength.

You may choose a different illuminating source spectrum by specifying a `source` parameter in the call to `calc_psf()`. The following are valid sources:

1. A `pysynphot.Spectrum` object. This is the best option, providing maximum ease and accuracy, but requires the user to have `pysynphot` installed. In this case, the `Spectrum` object is combined with a `pysynphot.ObsBandpass` for the selected instrument and filter to derive the effective stimulus in detected photoelectrons versus wavelength. This is binned to the number of wavelengths set by the `nlambda` parameter.
2. A dictionary with elements `source["wavelengths"]` and `source["weights"]` giving the wavelengths in meters and the relative weights for each. These should be numpy arrays or lists. In this case, the wavelengths and weights are used exactly as provided, without applying the instrumental filter profile.

```
>>> src = {'wavelengths': [2.0e-6, 2.1e-6, 2.2e-6], 'weights': [0.3, 0.5, 0.2]}
>>> nc.calc_psf(source=src, outfile='psf_for_src.fits')
```

3. A tuple or list containing the numpy arrays `(wavelength, weights)` instead.

As a convenience, webbpsf includes a function to retrieve an appropriate `pysynphot.Spectrum` object for a given stellar spectral type from the PHOENIX or Castelli & Kurucz model libraries.

```
>>> src = webbpsf.specFromSpectralType('G0V', catalog='phoenix')
>>> psf = miri.calc_psf(source=src)
```

3.1.2 Making Monochromatic PSFs

To calculate a monochromatic PSF, just use the `monochromatic` parameter. Wavelengths are always specified in meters.

```
>>> psf = miri.calc_psf(monochromatic=9.876e-6)
```

3.1.3 Adjusting source position, centering, and output format

A number of non-instrument-specific calculation options can be adjusted through the `options` dictionary attribute on each instrument instance. (For a complete listing of options available, consult `JWInstrument.options`.)

Input Source position offsets

The PSF may be shifted off-center by adjusting the offset of the stellar source. This is done in polar coordinates:

```
>>> instrument.options['source_offset_r'] = 0.3         # offset in arcseconds
>>> instrument.options['source_offset_theta'] = 45.     # degrees counterclockwise from instrumental +Y␣
↪in the science frame
```

If these options are set, the offset is applied relative to the central coordinates as defined by the output array size and parity (described just below).

For coronagraphic modes, the coronagraph occulter is always assumed to be at the center of the output array. Therefore, these options let you offset the source away from the coronagraph.

Simulating telescope jitter

Space-based observatories don't have to contend with the seeing limit, but imprecisions in telescope pointing can have the effect of smearing out the PSF. To simulate this with WebbPSF, the option names are `jitter` and `jitter_sigma`.

```
>>> instrument.options['jitter'] = 'gaussian'   # jitter model name or None
>>> instrument.options['jitter_sigma'] = 0.009  # in arcsec, default 0.007
```

Array sizes, star positions, and centering

Output array sizes may be specified either in units of arcseconds or pixels. For instance,

```
>>> mynircam = webbpsf.NIRCam()
>>> result = mynircam.calc_psf(fov_arcsec=7, oversample=2, filter='F250M')
>>> result2= mynircam.calc_psf(fov_pixels=512, oversample=2, filter='F250M')
```

In the latter example, you will in fact get an array which is 1024 pixels on a side: 512 physical detector pixels, times an oversampling of 2.

By default, the PSF will be centered at the exact center of the output array. This means that if the PSF is computed on an array with an odd number of pixels, the PSF will be centered exactly on the central pixel. If the PSF is computed on an array with even size, it will be centered on the "crosshairs" at the intersection of the central four pixels. If one of these is particularly desirable to you, set the parity option appropriately:

```
>>>  instrument.options['parity'] = 'even'
>>>  instrument.options['parity'] = 'odd'
```

Setting one of these options will ensure that a field of view specified in arcseconds is properly rounded to either odd or even when converted from arcsec to pixels. Alternatively, you may also just set the desired number of pixels explicitly in the call to calc_psf():

```
>>>  instrument.calc_psf(fov_npixels=512)
```

Note: Please note that these parity options apply to the number of *detector pixels* in your simulation. If you request oversampling, then the number of pixels in the output file for an oversampled array will be `fov_npixels` times `oversampling`. Hence, if you request an odd parity with an even oversampling of, say, 4, then you would get an array with a total number of data pixels that is even, but that correctly represents the PSF located at the center of an odd number of detector pixels.

Output format options for sampling

As just explained, WebbPSF can easily calculate PSFs on a finer grid than the detector's native pixel scale. You can select whether the output data should include this oversampled image, a copy that has instead been rebinned down to match the detector scale, or optionally both. This is done using the `options['output_mode']` parameter.

```
>>> nircam.options['output_mode'] = 'oversampled'
>>> psf = nircam.calc_psf()       # the 'psf' variable will be an oversampled PSF, formatted as a FITS␣
↪HDUlist
>>>
>>> nircam.options['output_mode'] = 'detector sampled'
>>> psf2 = nircam.calc_psf()      # now 'psf2' will contain the result as resampled onto the detector␣
↪scale.
```

(continues on next page)

(continued from previous page)

```
>>>
>>> nircam.options['output_mode'] = 'both'
>>> psf3 = nircam.calc_psf()      # 'psf3' will have the oversampled image as primary HDU, and
>>>                              # the detector-sampled image as the first image extension HDU.
```

Warning: The default behavior is both. Note that at some point in the future, this default is likely to change to detector sampling. To future-proof your code, set `options['output_mode']` explicitly.

3.1.4 Pixel scales, sampling, and oversampling

The derived instrument classes all know their own instrumental pixel scales. You can change the output pixel scale in a variety of ways, as follows. See the `JWInstrument.calc_psf` documentation for more details.

1. Set the `oversample` parameter to calc_psf(). This will produce a PSF with a pixel grid this many times more finely sampled. `oversample=1` is the native detector scale, `oversample=2` means divide each pixel into 2x2 finer pixels, and so forth. You can automatically obtain both the oversampled PSF and a version rebinned down onto the detector pixel scale by setting `rebin=True` in the call to calc_psf:

```
>>> hdulist = instrument.calc_psf(oversample=2, rebin=True)    # hdulist will contain a primary␣
↪HDU with the
>>>                                                            # oversampled data, plus an image␣
↪extension
>>>                                                            # with the PSF rebinned down to␣
↪regular sampling.
```

2. For coronagraphic calculations, it is possible to set different oversampling factors at different parts of the calculation. See the `calc_oversample` and `detector_oversample` parameters. This is of no use for regular imaging calculations (in which case `oversample` is a synonym for `detector_oversample`). Specifically, the `calc_oversample` keyword is used for Fourier transformation to and from the intermediate optical plane where the occulter (coronagraph spot) is located, while `detector_oversample` is used for propagation to the final detector. Note that the behavior of these keywords changes for coronagraphic modeling using the Semi-Analytic Coronagraphic propagation algorithm (not fully documented yet - contact Marshall Perrin if curious).

```
>>> miri.calc_psf(calc_oversample=8, detector_oversample=2)  # model the occulter with very fine␣
↪pixels, then save the
>>>                                                          # data on a coarser (but still␣
↪oversampled) scale
```

3. Or, if you need even more flexibility, just change the `instrument.pixelscale` attribute to be whatever arbitrary scale you require.

```
>>> instrument.pixelscale = 0.0314159
```

Note that the calculations performed by WebbPSF are somewhat memory intensive, particularly for coronagraphic observations. All arrays used internally are double-precision complex floats (16 bytes per value), and many arrays of size `(npixels * oversampling)^2` are needed (particularly if display options are turned on, since the matplotlib graphics library makes its own copy of all arrays displayed).

Your average laptop with a couple GB of RAM will do perfectly well for most computations so long as you're not too ambitious with setting array size and oversampling. If you're interested in very high fidelity simulations of large fields (e.g. 1024x1024 pixels oversampled 8x) then we recommend a large multicore desktop with >16 GB RAM.

3.1.5 PSF normalization

By default, PSFs are normalized to total intensity = 1.0 at the entrance pupil (i.e. at the JWST OTE primary). A PSF calculated for an infinite aperture would thus have integrated intensity =1.0. A PSF calculated on any smaller finite subarray will have some finite encircled energy less than one. For instance, at 2 microns a 10 arcsecond size FOV will enclose about 99% of the energy of the PSF. Note that if there are any additional obscurations in the optical system (such as coronagraph masks, spectrograph slits, etc), then the fraction of light that reaches the final focal plane will typically be significantly less than 1, even if calculated on an arbitrarily large aperture. For instance the NIRISS NRM mask has a throughput of about 15%, so a PSF calculated in this mode with the default normalization will have integrated total intensity approximately 0.15 over a large FOV.

If a different normalization is desired, there are a few options that can be set in calls to calc_psf:

```
>>> psf = nc.calc_psf(normalize='last')
```

The above will normalize a PSF after the calculation, so the output (i.e. the PSF on whatever finite subarray) has total integrated intensity = 1.0.

```
>>> psf = nc.calc_psf(normalize='exit_pupil')
```

The above will normalize a PSF at the exit pupil (i.e. last pupil plane in the optical model). This normalization takes out the effect of any pupil obscurations such as coronagraph masks, spectrograph slits or pupil masks, the NIRISS NRM mask, and so forth. However it still leaves in the effect of any finite FOV. In other words, PSFs calculated in this mode will have integrated total intensity = 1.0 over an infinitely large FOV, even after the effects of any obscurations.

Note: An aside on throughputs and normalization: Note that *by design* WebbPSF does not track or model the absolute throughput of any instrument. Consult the JWST Exposure Time Calculator and associated reference material if you are interested in absolute throughputs. Instead WebbPSF simply allows normalization of output PSFs' total intensity to 1 at either the entrance pupil, exit pupil, or final focal plane. When used to generate monochromatic PSFs for use in the JWST ETC, the entrance pupil normalization option is selected. Therefore WebbPSF first applies the normalization to unit flux at the primary mirror, propagates it through the optical system ignoring any reflective or transmissive losses from mirrors or filters (since the ETC throughput curves take care of those), and calculates only the diffractive losses from slits and stops. Any loss of light from optical stops (Lyot stops, spectrograph slits or coronagraph masks, the NIRISS NRM mask, etc.) will thus be included in the WebbPSF calculation. Everything else (such as reflective or transmissive losses, detector quantum efficiencies, etc., plus scaling for the specified target spectrum and brightness) is the ETC's job. This division of labor has been coordinated with the ETC team and ensures each factor that affects throughput is handled by one or the other system but is not double counted in both.

To support realistic calculation of broadband PSFs however, WebbPSF does include normalized copies of the relative spectral response functions for every filter in each instrument. These are included in the WebbPSF data distribution, and are derived behind the scenes from the same reference database as is used for the ETC. These relative spectral response functions are used to make a proper weighted sum of the individual monochromatic PSFs in a broadband calculation: weighted *relative to the broadband total flux of one another*, but still with no implied absolute normalization.

3.1.6 Controlling output log text

WebbPSF can output a log of calculation steps while it runs, which can be displayed to the screen and optionally saved to a file. This is useful for verifying or debugging calculations. To turn on log display, just run

```
>>> webbpsf.setup_logging(filename='webbpsf.log')
```

The setup_logging function allows selection of the level of log detail following the standard Python logging system (DEBUG, INFO, WARN, ERROR). To disable all printout of log messages, except for errors, set

```
>>> webbpsf.setup_logging(level='ERROR')
```

WebbPSF remembers your chosen logging settings between invocations, so if you close and then restart python it will automatically continue logging at the same level of detail as before. See `webbpsf.setup_logging()` for more details.

3.2 Advanced Usage: Output file format, OPDs, and more

This section serves as a catch-all for some more esoteric customizations and applications. See also the *More Examples* page.

3.2.1 Writing out only downsampled images

Perhaps you may want to calculate the PSF using oversampling, but to save disk space you only want to write out the PSF downsampled to detector resolution.

```
>>> result =  inst.calc_psf(args, ...)
>>> result['DET_SAMP'].writeto(outputfilename)
```

Or if you really care about writing it as a primary HDU rather than an extension, replace the 2nd line with

```
>>> pyfits.PrimaryHDU(data=result['DET_SAMP'].data, header=result['DET_SAMP'].header).
↪writeto(outputfilename)
```

3.2.2 Writing out intermediate images

Your calculation may involve intermediate pupil and image planes (in fact, it most likely does). WebbPSF / POPPY allow you to inspect the intermediate pupil and image planes visually with the display keyword argument to `calc_psf()`. Sometimes, however, you may want to save these arrays to FITS files for analysis. This is done with the `save_intermediates` keyword argument to `calc_psf()`.

The intermediate wavefront planes will be written out to FITS files in the current directory, named in the format `wavefront_plane_%03d.fits`. You can additionally specify what representation of the wavefront you want saved with the `save_intermediates_what` argument to `calc_psf()`. This can be `all`, `parts`, `amplitude`, `phase` or `complex`, as defined as in `poppy.Wavefront.asFITS()`. The default is to write `all` (intensity, amplitude, and phase as three 2D slices of a data cube).

If you pass `return_intermediates=True` as well, the return value of calc_psf is then `psf, intermediate_wavefronts_list` rather than the usual `psf`.

Warning: The `save_intermediates` keyword argument does not work when using parallelized computation, and WebbPSF will fail with an exception if you attempt to pass `save_intermediates=True` when running in parallel. The `return_intermediates` option has this same restriction.

3.2.3 Providing your own OPDs or pupils from some other source

It is straight forward to configure an Instrument object to use a pupil OPD file of your own devising, by setting the `pupilopd` attribute of the Instrument object:

```
>>> niriss = webbpsf.NIRISS()
>>> niriss.pupilopd = "/path/to/your/OPD_file.fits"
```

If you have a pupil that is an array in memory but not saved on disk, you can pass it in as a fits.HDUList object :

```
>>> myOPD = some_function_that_returns_properly_formatted_HDUList(various, function, args...)
>>> niriss.pupilopd = myOPD
```

Likewise, you can set the pupil transmission file in a similar manner by setting the `pupil` attribute:

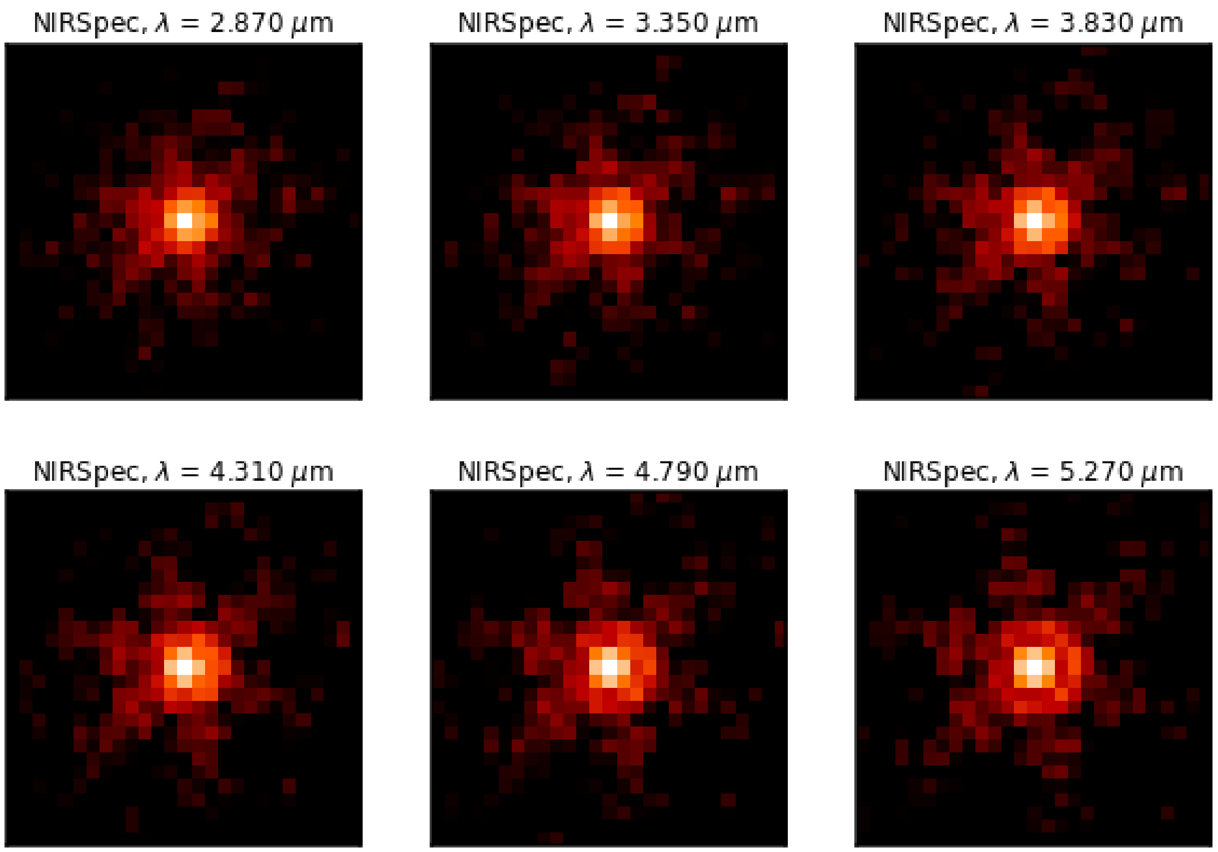
```
>>> niriss.pupil = "/path/to/your/OPD_file.fits"
```

Please see the documentation for `poppy.FITSOpticalElement` for information on the required formatting of the FITS file. In particular, you will need to set the `PUPLSCAL` keyword, and OPD values must be given in units of meters.

3.2.4 Calculating Data Cubes

Sometimes it is convenient to calculate many PSFs at different wavelengths with the same instrument config. You can do this just by iterating over calls to `calc_psf`, but there's also a function to automate this: `calc_datacube`. For example, here's something loosely like the NIRSpec IFU in F290LP:

```
# Set up a NIRSpec instance
nrs = webbpsf.NIRSpec()
nrs.image_mask = None # No MSA for IFU mode
nl = np.linspace(2.87e-6, 5.27e-6, 6)

# Calculate PSF datacube
cube = nrs.calc_datacube(wavelengths=nl, fov_pixels=27, oversample=4)

# Display the contents of the data cube
fig, axes = plt.subplots(nrows=2, ncols=3, figsize=(10,7))
for iy in range(2):
    for ix in range(3):
        ax=axes[iy,ix]
        i = iy*3+ix
        wl = cube[0].header['WAVELN{:02d}'.format(i)]

        # Note that when displaying datacubes, you have to set the "cube_slice" parameter
        webbpsf.display_psf(cube, ax=ax, cube_slice=i,
                            title="NIRSpec, $\lambda$ = {:.3f} $\mu$m".format(wl*1e6),
                            vmax=.2, vmin=1e-4, ext=1, colorbar=False)
        ax.xaxis.set_visible(False)
        ax.yaxis.set_visible(False)
```



3.2.5 Subclassing a JWInstrument to add additional functionality

Perhaps you want to modify the OPD used for a given instrument, for instance to add a defocus. You can do this by subclassing one of the existing instrument classes to override the `JWInstrument._addAdditionalOptics()` function. An `OpticalSystem` is basically a list so it's straightforward to just add another optic there. In this example it's a lens for defocus but you could just as easily add another `FITSOpticalElement` instead to read in a disk file.

Note, we do this as an example here to show how to modify an instrument class by subclassing it, which can let you add arbitrary new functionality. There's an easier way to add defocus specifically; see below.

```
>>> class FGS_with_defocus(webbpsf.FGS):
>>>     def __init__(self, *args, **kwargs):
>>>         webbpsf.FGS.__init__(self, *args, **kwargs)
>>>         # modify the following as needed to get your desired defocus
>>>         self.defocus_waves = 0
>>>         self.defocus_lambda = 4e-6
>>>     def _addAdditionalOptics(self, optsys, *args, **kwargs):
>>>         optsys = webbpsf.FGS._addAdditionalOptics(self, optsys, *args, **kwargs)
>>>         lens = poppy.ThinLens(
>>>             name='FGS Defocus',
>>>             nwaves=self.defocus_waves,
>>>             reference_wavelength=self.defocus_lambda
>>>         )
>>>         lens.planetype = poppy.PUPIL  # tell propagation algorithm which this is
>>>         optsys.planes.insert(1, lens)
>>>         return optsys
```

(continues on next page)

(continued from previous page)

```
>>>
>>> fgs2 = FGS_with_defocus()
>>> # apply 4 waves of defocus at the wavelength
>>> # defined by FGS_with_defocus.defocus_lambda
>>> fgs2.defocus_waves = 4
>>> psf = fgs2.calc_psf()
>>> webbpsf.display_psf(psf)
```

3.2.6 Defocusing an instrument

The instrument options dictionary also lets you specify an optional defocus amount. You can specify both the wavelength at which it should be applied, and the number of waves of defocus (at that wavelength, specified as waves peak-to-valley over the circumscribing circular pupil of JWST).

```
>>> nircam.options['defocus_waves'] = 3.2
>>> nircam.options['defocus_wavelength'] = 2.0e-6
```

CHAPTER 4

JWST Instrument Model Details

The following describes specific details for the various JWST instrument classes. See also *the references page* for information on data sources.

One general note is that the `webbpsf` class interfaces do not attempt to exactly model the implementation details of all instrument mechanisms, particularly for NIRCam and NIRISS that each have multiple wheels. The `filter` attribute of a given class is used to select any and all filters, even if as a practical matter a given filter is physically installed in a "pupil" wheel instead of a "filter" wheel. Likewise any masks that affect the aperture shape are selected via the `pupil_mask` attribute even if physically an optic is in a so-called "filter" wheel.

All classes share some common attributes:

- `filter` which is the string name of the currently selected filter. The list of available filters is provided as the `filter_list` attribute.
- `image_mask` which is the name of a selected image plane element such as a coronagraph mask or spectrograph slit, or `None` to indicate no such element is present. The list of available options is provided as the `image_mask_list` attribute.
- `pupil_mask` likewise allows specification of any optic that modifies the pupil plane subsequent to the image mask, such as a coronagraphic Lyot stop or spectrograph grating stop. The list of available options is provided as the `pupil_mask_list` attribute.
- Each SI has a `detector` attribute that can be used to select among its multiple detectors (if more than one are present in that SI), and a `detector_position` attribute which is a 2-tuple giving the pixel coordinates on that detector for the center location in any calculated output PSF. Note that the `detector_position` value should be specified using the Python (Y,X) axes order convention.

Warning: WebbPSF provides some sanity checking on user inputs, but does not strive to strictly forbid users from trying to simulate instrument configurations that may not be achievable in practice. Users are responsible for knowing the available modes well enough to be aware if they are trying to simulate an inconsistent or physically impossible configuration.

4.1 Optical Telescope Element (OTE)

The JWST Optical Telescope Element consists of the telescope optics that serve all the science instruments and the fine guidance sensor. Most notably, this means the primary, secondary, tertiary, and fast steering mirrors. The OTE contributes to the overall wavefront error (and therefore the aberrations in instrument PSFs) in a few ways:

- The limits of precisely manufacturing the mirrors introduce tiny high spatial frequency bumps and ripples of optical path difference
- During commissioning, the telescope mirror segments will be aligned and phased as precisely as possible, but small errors in the final aligned configuration will still contribute to WFE
- The WFE will vary with field position, which is inherent in the OTE optical design even if perfectly aligned
- Aberrations can be introduced by pupil shear or other misalignments between the OTE and each science instrument

These effects are simulated at high fidelity in models maintained by Ball Aerospace, which in turn were used to create the OPD map files for the JWST instruments included in WebbPSF. Specifically, WebbPSF uses information derived from the as-built OTE optical model Revision G (for the static surface figures of each segments) and the overall JWST optical error budget Revision W (for OTE to ISIM misalignments, WFSC residuals, stability, and budgeted uncertainties for both the OTE and SI contributions).

JWST's optical system has been extremely precisely engineered and assembled. Individual mirrors typically have below 30 nm r.m.s. WFE, and the overall OTE system including alignment tolerances and dynamics is expected to deliver wavefronts of roughly 100 to 150 nm r.m.s. WFE to each of the instruments. This corresponds to Strehl ratios of 90% or better for wavelengths beyond 2 microns.

Further information on JWST's predicted optical performance is available in "Status of the optical performance for the James Webb Space Telescope", Lightsey *et al.*, (2014) and "Predicted JWST imaging performance", Knight *et al.* (2012).

For each science instrument, if you examine `inst.opd_list` (where `inst` is an instance of an instrument model), you will see the filenames for a "predicted" OPD map and a "requirements" OPD map. For example:

```
>>> nc = webbpsf.NIRCam()
>>> nc.opd_list
['OPD_RevW_ote_for_NIRCam_predicted.fits.gz',
 'OPD_RevW_ote_for_NIRCam_requirements.fits.gz']
```

By default, WebbPSF selects the (slightly more conservative) `requirements` OPD map:

```
>>> nc.pupilopd
'OPD_RevW_ote_for_NIRCam_requirements.fits.gz'
```

Performance predictions for a large active deployable space telescope are inherently probabilistic, and Monte Carlo methods have been used to derive overall probability distributions based on the individual error budget terms. The "predicted" OPD maps provided with WebbPSF correspond to the median values from such simulations, and provide a reasonable approximation of current performance expectations. However, performance at such levels is not guaranteed. The "requirements" OPD maps are more conservative, set to the slightly higher levels of residual wavefront error that we can be confident will be achieved in practice. Both the predicted and required values contain maximal budgeted contributions from OTE temporal drifts and dynamics (roughly 55 nm of low and mid frequency error); i.e. they correspond to times well after a wavefront control and shortly before a next set of control moves might be issued.

To select the `predicted` map, simply assign it to the `pupilopd` attribute before calculating the PSF:

```
>>> nc.pupilopd = 'OPD_RevW_ote_for_NIRCam_predicted.fits.gz'
```

For both the required and predicted cases, the OPD files contain 10 Monte Carlo realizations of the telescope. You can select one of these by specifying the plane number in a tuple:

```
>>> nc.pupilopd = ('OPD_RevW_ote_for_NIRCam_predicted.fits.gz', 7)
```

Note that these represent 10 distinct, totally independent realizations of JWST and its optical error budget. They do not represent any sort of time series or wavefront drift. The average levels of WFE from the telescope itself used in the OPD files are as follows.

Instrument	Predicted	Requirements
NIRCam	90 nm rms	117 nm rms
NIRSpec	163 nm rms	188 nm rms
NIRISS	108 nm rms	145 nm rms
MIRI	204 nm rms	258 nm rms

While different OPD maps are used for each SI, these OPD maps do not include wavefront error contributions from optics internal to the science instrument. Additional details on the SI-specific wavefront error models are given under each instrument model section below.

4.2 NIRCam

4.2.1 Imaging

NIRCam is one of the more complicated classes in `webbpsf`, and has several unique selectable options to model the two copies of NIRCam each with two channels.

The `detector` attribute can be used to select between any of the ten detectors, A1-A5 and B1-B5. Additional attributes are then automatically set for `channel` ("short" or "long") and module ("A" or "B") but these cannot be set directly; just set the desired detector and the channel and module are inferred automatically.

The choice of `filter` also impacts the channel selection: If you choose a long-wavelength filter such as F460M, then the detector will automatically switch to the long-wave detector for the current channel. For example, if the detector was previously set to A2, and the user enters `nircam.filter = "F460M"` then the detector will automatically change to A5. If the user later selects `nircam.filter = "F212N"` then the detector will switch to A1 (and the user will need to manually select if a different short wave detector is desired). This behavior on filter selection can be disabled by setting `nircam.auto_channel = False`.

NIRCam class automatic pixelscale changes

The `pixelscale` will automatically toggle to the correct scale for LW or SW based on user inputs for either detector or filter. If you set the `detector` to NRCA1-4 or NRCB1-4, the scale will be set for SW, otherwise for NRCA5 or NRCB5 the pixel scale will be for LW. If you set the `filter` attribute to a filter in the short wave channel, the pixel scale will be set for SW, otherwise for a filter in the long wave challen the scale will be set for LW.

The intent is that the user should in general automatically get a PSF with the appropriate pixelscale for whatever instrument config you're trying to simulate, with no extra effort needed by the user to switch between NIRCam's two channels.

Note that this behavior is *not* invoked for monochromatic calculations; you can't just iterate over calc_psf calls at different wavelengths and expect it to toggle between SW and LW at some point. The workaround is simple, just set either the filter or detector attribute whenever you want to toggle between SW or LW channels.

4.2.2 Coronagraph Masks

The coronagraph image-plane masks and pupil-plane Lyot masks are all included as options. These are based on the nominal design properties as provided by the NIRCam team, not on any specific measurements of the as-built masks. The simulations of the occulting mask fields also include the nearby neutral density squares for target acquisitions.

WebbPSF won't prevent users from simulating configuration using a coronagraph image mask without the Lyot stop, but that's not something that can be done for real with NIRCam.

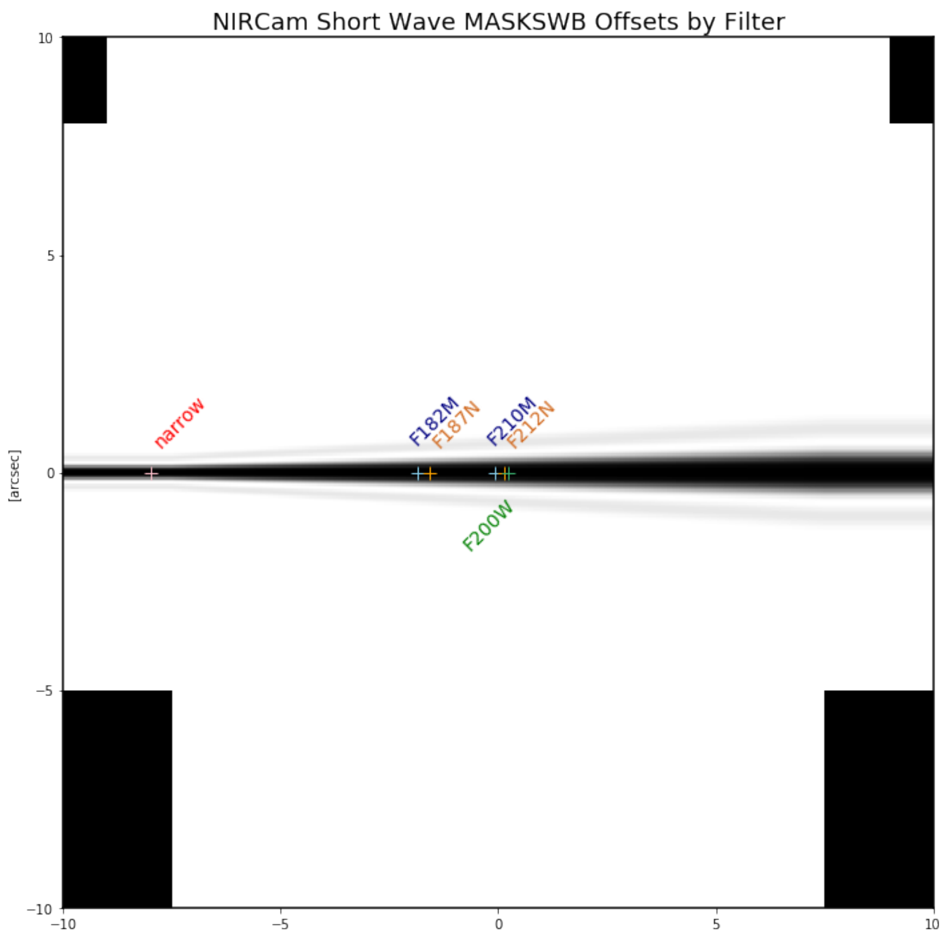
Note, the Lyot masks have multiple names for historical reasons: The names 'CIRCLYOT' and 'WEDGELYOT' have been used since early in WebbPSF development, and can still be used, but the same masks can also be referred to as "MASKRND" and "MASKSWB" or "MASKLWB", the nomenclature that was eventually adopted for use in APT and other JWST documentation. Both ways work and will continue to do so.

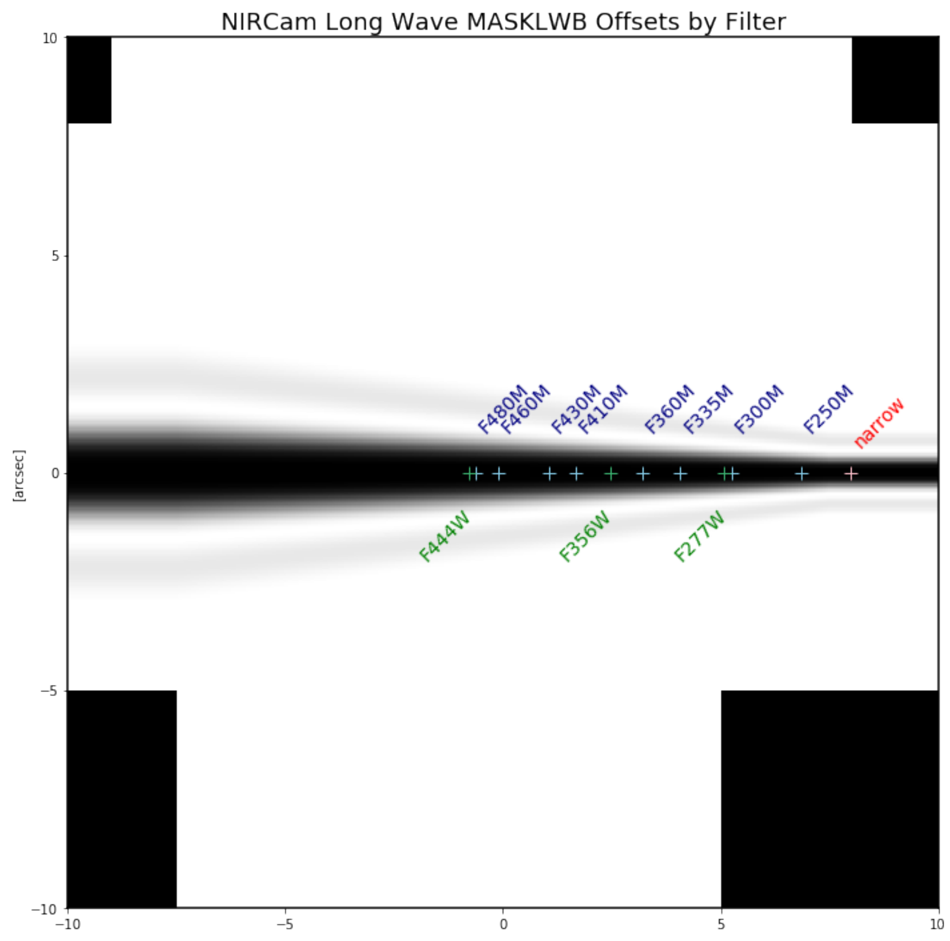
Offsets along the MASKLWB and MASKSWB masks:

Each allowable filter has its own default location along one of these masks. The appropriate offset is automatically selected in WebbPSF based on the currently selected filter name. If you want to do something different, you can set the `bar_offset` option:

```
>>> nc.options['bar_offset'] = 2.0    # Offsets 2 arcseconds in +X along the mask
or
>>> nc.options['bar_offset'] = 'F480M'  # Use the position for F480M regardless of the currently␣
↪selected filter
```

Note that just because you can simulate such arbitrary position in WebbPSF does not mean you can easily actually achieve that pointing with the flight hardware.


NIRCam Short Wave MASKSWB Offsets by Filter
[arcsec]
narrow
F182M
F187N
F210M
F212N
F200W
10
5
0
−5
−10
−10
−5
0
5
10



NIRCam class automatic detector position setting for coronagraphy

Each coronagraphic mask is imaged onto a specific area of a specific detector. Setting the image mask attribute to a coronagraphic mask (e.g. MASKLWB or MASK335R) will automatically configure the `detector` and `detector_position` attributes appropriately for that mask's field point. Note, this will also invoke the automatic pixelscale functionality to get the right scale for SW or LW, too.

Warning: Coronagraph field point WFE not yet included

The coronagraph field points are far off axis, and this comes with significant WFE added compared to the inner portion of the NIRCam field of view. This is not yet included in WebbPSF, but is work in progress for the next version expected in 2019.

4.2.3 Weak Lenses for Wavefront Sensing

WebbPSF includes models for the three weak lenses used for wavefront sensing, including the pairs of lenses that can be used together simultaneously.

The convention is such that the "negative" 8 waves lens is concave, the "positive" two lenses are convex. Thus positive weak lenses move best focus in front of the detector, or equivalently the electric field imaged on the detector becomes behind or beyond best focus. Negative weak lenses move best focus behind the detector, or equivalently the image on the detector is moved closer to the OTE exit pupil than best focus.

Note that the weak lenses are in the short wave channel only; webbpsf won't stop you from simulating a LW image with a weak lens, but that's not a real configuration that can be acheived with NIRCam.

4.2.4 SI WFE

(Not yet available)

The SI internal WFE measurements are distinct for each of the modules and channels. When enabled, these are added to the final pupil of the optical train, i.e. after the coronagraphic image planes.

4.2.5 Wavelength-Dependent Focus Variations

TODO Add documentation here for the focus variations vs wavelength and how webbpsf models those.

4.3 NIRSpec

4.3.1 Imaging and spectroscopy

webbpsf models the optics of NIRSpec, mostly in **imaging** mode or for monochromatic PSFs that can be assembled into spectra using other tools.

This is not a substitute for a spectrograph model, but rather a way of simulating a PSF as it would appear with NIRSpec in imaging mode (e.g. for target acquisition). It can also be used to produce monochromatic PSFs appropriate for spectroscopic modes, but other software must be used for assembling those monochromatic PSFs into a spectrum.

Slits: webbpsf includes models of each of the fixed slits in NIRSpec (S200A1, S1600A1, and so forth), plus a few patterns with the MSA: (1) a single open shutter, (2) three adjacent open shutters to make a mini-slit, and (3) all shutters open at once. Other MSA patterns could be added if requested by users.

By default the `pupil_mask` is set to the "NIRSpec grating" pupil mask. In this case a rectangular pupil mask 8.41x7.91 m as projected onto the primary is added to the optical system at the pupil plane after the MSA. This is an estimate of the pupil stop imposed by the outer edge of the grating clear aperture, estimated based on optical modeling by Erin Elliot and Marshall Perrin.

4.3.2 SI WFE

(Not yet available)

SI WFE will most likely be added to the entrance pupil, prior to the MSA image plane. This model is still under development.

4.4 NIRISS

4.4.1 Imaging and AMI

WebbPSF models the direct imaging and nonredundant aperture masking interferometry modes of NIRISS in the usual manner.

Note that long wavelength filters (>2.5 microns) are used with a pupil obscuration which includes the pupil alignment reference fixture. This is called the "CLEARP" pupil.

Based on the selected filter, webbpsf will automatically toggle the `pupil_mask` between "CLEARP" and the regular clear pupil (i.e. `pupil_mask = None`).

4.4.2 Slitless Spectroscopy

webbpsf provides preliminary support for the single-object slitless spectroscopy ("SOSS") mode using the GR700XD cross-dispersed grating. Currently this includes the clipping of the pupil due to the undersized grating and its mounting hardware, and the cylindrical lens that partially defocuses the light in one direction.

Warning: Prototype implementation - Not yet fully tested or verified.

Note that WebbPSF does not model the spectral dispersion in any of NIRISS' slitless spectroscopy modes. For wide-field slitless spectroscopy, this can best be simulated by using webbpsf output PSFs as input to the aXe spectroscopy code. Contact Van Dixon at STScI for further information. For SOSS mode, contact Loic Albert at Universite de Montreal.

The other two slitless spectroscopy grisms use the regular pupil and do not require any special support in WebbPSF; just calculate monochromatic PSFs at the desired wavelengths and assemble them into spectra using tools such as aXe.

4.4.3 Coronagraph Masks

NIRISS includes four coronagraphic occulters, machined as features on its pick-off mirror. These were part of its prior incarnation as TFI, and are not expected to see much use in NIRISS. However they remain a part of the physical instrument and we retain in webbpsf the capability to simulate them.

4.4.4 SI WFE

(Not yet available)

The SI internal WFE measurements are distinct for each of the modules and channels. When enabled, these are added to the final pupil of the optical train, i.e. after the coronagraphic image planes.

4.5 MIRI

4.5.1 Imaging

webbpsf models the MIRI imager; currently there is no specific support for MRS, however monochromatic PSFS computed for the imager may be used as a reasonable proxy for PSF properties at the entrance to the MRS slicers.

4.5.2 Coronagraphy

webbpsf includes models for all three FQPM coronagraphs and the Lyot coronagraph. In practice, the wavelength selection filters and the Lyot stop are co-mounted. webbpsf models this by automatically setting the `pupil_mask` element to one of the coronagraph masks or the regular pupil when the `filter` is changed. If you want to disable this behavior, set `miri.auto_pupil = False`.

4.5.3 LRS Spectroscopy

webbpsf includes models for the LRS slit and the subsequent pupil stop on the grism in the wheels. Users should select `miri.image_mask = "LRS slit"` and `miri.pupil_mask = 'P750L LRS grating'`. That said, the LRS simulations have not been extensively tested yet; feedback is appreciated about any issues encountered.

4.5.4 SI WFE

(Not yet available)

The SI internal WFE measurements, when enabled, are added to the final pupil of the optical train, i.e. after the coronagraphic image planes.

4.5.5 Minor Field-Dependent Pupil Vignetting

TODO Add documentation here of this effect and how WebbPSF models it.

A fold mirror at the MIRI Imager's internal cold pupil is used to redirect light from the MIRI calibration sources towards the detector, to enable flat field calibrations. For a subset of field positions, this fold mirror slightly obscures a small portion of the pupil. This is a small effect with little practical consequence for MIRI PSFs, but WebbPSF does model it.

4.6 FGS

The FGS class does not have any selectable optical elements (no filters or image or pupil masks of any kind). Only the detector is selectable, between either 'FGS1' or 'FGS2'.

4.6.1 SI WFE

(Not yet available)

CHAPTER 5

WebbPSF for WFIRST

WebbPSF provides a framework for instrument PSF calculations that is easily extensible to other instruments and observatories. The `webbpsf.wfirst` module was developed to enable simulation of WFIRST's instruments, the *Wide Field Instrument (WFI)* and *Coronagraph Instrument (CGI)*.

5.1 Wide Field Instrument (WFI)

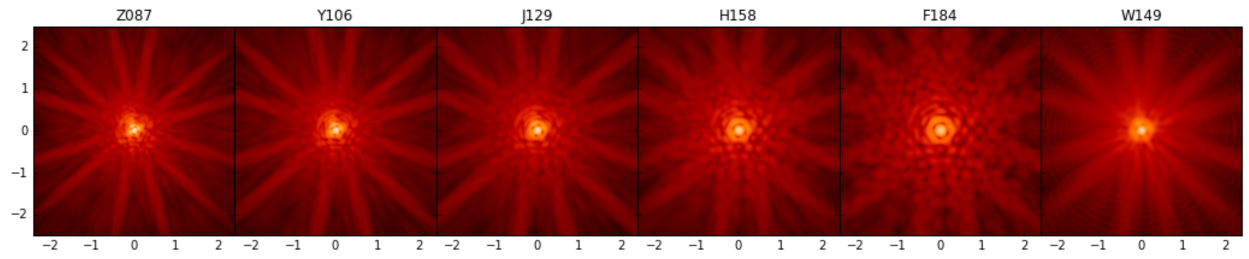


Fig. 1: Sample PSFs for the filters in the WFIRST WFI. Angular scale in arcseconds, log-scaled intensity.

The WFI model is based on the Cycle 6 instrument reference information from the WFIRST team at Goddard Space Flight Center.

At this time, the only instrument simulated is the WFI, but that may change in the future. To work with the WFI model, import and instantiate it as follows:

```
>>> import webbpsf
>>> from webbpsf import wfirst
>>> wfi = wfirst.WFI()
```

Usage of the WFI model class is, for the most part, just like any other WebbPSF instrument model. For help setting things like filters, position offsets, and sampling refer back to *Using WebbPSF via the Python API*.

The WFI model includes a model for field dependent PSF aberrations. With as large a field of view as the WFI is designed to cover, there will be variation in the PSF from one end of the field of view to the other. WebbPSF's WFI model faithfully reproduces the field dependent aberrations calculated from the Goddard WFIRST team's Cycle 6 WFI design. This provides a toolkit for users to assess the impact of inter-SCA and intra-SCA PSF variations on science cases of interest.

Quickstart IPython Notebook

This documentation is complemented by an IPython Notebook format quickstart tutorial. Downloading and run that notebook to use the beta notebook GUI for the WFI model, and to explore code samples for common tasks interactively.

Caution: Note that unlike most JWST modes, WFIRST WFI is *significantly* undersampled relative to Nyquist. Undersampled data is inherently lossy with information, and subject to aliasing. Measurements of properties such as encircled energy, FWHM, Strehl ratio, etc cannot be done precisely on undersampled data.

In flight, we will use dithering and similar strategies to reconstruct better-sampled images. The same can be done in simulation using WebbPSF. **Only measure PSF properties such as FWHM or encircled energy on well-sampled data**. That means either simulating dithered undersampled data at multiple subpixel steps and drizzling them back together, or else performing your measurements on oversampled calculation outputs. (I.e. in webbpsf, set `wfi.oversample=4` or more, and perform your measurements on extension 0 of the returned FITS file.)

5.1.1 Field dependence in the WFI model

Field points are specified in a WebbPSF calculation by selecting a detector and pixel coordinates within that detector. A newly instantiated WFI model already has a default detector and position.

```
>>> wfi.detector
'SCA01'
>>> wfi.detector_position
(2048, 2048)
```

The WFI field of view is laid out as shown in the figure. To select a different detector, assign its name to the `detector` attribute:

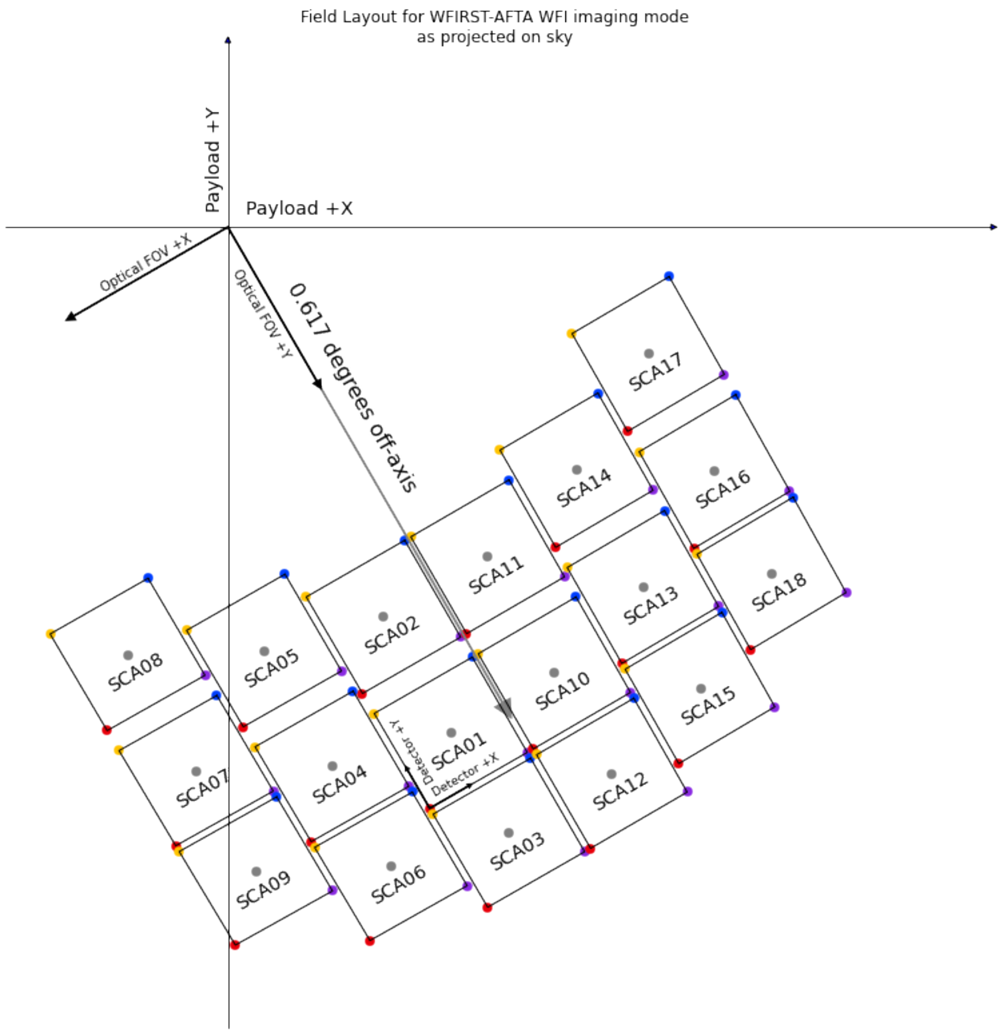
```
>>> wfi.detector_list
['SCA01', 'SCA02', 'SCA03', 'SCA04', 'SCA05', 'SCA06', 'SCA07', 'SCA08', 'SCA09', 'SCA10', 'SCA11',
↪'SCA12', 'SCA13', 'SCA14', 'SCA15', 'SCA16', 'SCA17', 'SCA18']
>>> wfi.detector = 'SCA03'
```

The usable region of the 4096 by 4096 pixel detectors specified for the Wide Field Instrument will range from (4, 4) to (4092, 4092), accounting for the 4 pixel wide bands of reference pixels. To change the position at which to calculate a PSF, simply assign an (X, Y) tuple:

```
>>> wfi.detector_position = (4, 400)
```

Warning: WebbPSF will not prevent you from setting an out of range detector position, but an error will be raised if you try to calculate a PSF with one.

```
>>> wfi.detector_position = (1, 1)
>>> wfi.calc_psf()
[ ... traceback omitted ... ]
RuntimeError: Attempted to get aberrations for an out-of-bounds field point
```


Field Layout for WFIRST-AFTA WFI imaging mode
as projected on sky
Payload +Y
Payload +X
Optical FOV +X
Optical FOV +Y
0.617 degrees off-axis
SCA01
SCA02
SCA03
SCA04
SCA05
SCA06
SCA07
SCA08
SCA09
SCA10
SCA11
SCA12
SCA13
SCA14
SCA15
SCA16
SCA17
SCA18
Detector +Y
Detector +X

The reference information available gives the field dependent aberrations in terms of Zernike polynomial coefficients from Z_1 to Z_{22}. These coefficients were calculated for five field points on each of 18 detectors, each at 16 unique wavelengths providing coverage from 0.76 μm to 2.0 μm (that is, the entire wavelength range of the WFI). WebbPSF interpolates the coefficients in position and wavelength space to allow the user to simulate PSFs at any valid pixel position and wavelength.

Bear in mind that the pixel position you set does not automatically set the **centering** of your calculated PSF. As with other models in WebbPSF, an `options` dictionary key can be set to specify 'even' (center on crosshairs between four pixels) or 'odd' (center on pixel center) parity.

```
>>> wfi.options['parity'] = 'even'  # best case for dividing PSF core flux
>>> wfi.options['parity'] = 'odd'  # worst case for PSF core flux landing in a single pixel
```

5.1.2 Example: Computing the PSF difference between opposite corners of the WFI field of view

This example shows the power of WebbPSF to simulate and analyze field dependent variation in the model. About a dozen lines of code are all that's necessary to produce a figure showing how the PSF differs between the two extreme edges of the instrument field of view.

```
>>> wfi = wfirst.WFI()
>>> wfi.filter = 'J129'
>>> wfi.detector = 'SCA09'
>>> wfi.detector_position = (4, 4)
>>> psf_sca09 = wfi.calc_psf()
>>> wfi.detector = 'SCA17'
>>> wfi.detector_position = (4092, 4092)
>>> psf_sca17 = wfi.calc_psf()
>>> fig, (ax_sca09, ax_sca17, ax_diff) = plt.subplots(1, 3, figsize=(16, 4))
>>> webbpsf.display_psf(psf_sca09, ax=ax_sca09, imagecrop=2.0, title='WFI SCA09, bottom left - J129')
>>> webbpsf.display_psf(psf_sca17, ax=ax_sca17, imagecrop=2.0, title='WFI SCA17, top right - J129')
>>> webbpsf.display_psf_difference(psf_sca09, psf_sca17, vmax=5e-3, title='(SCA09) - (SCA17)',␣
↪imagecrop=2.0, ax=ax_diff)
```

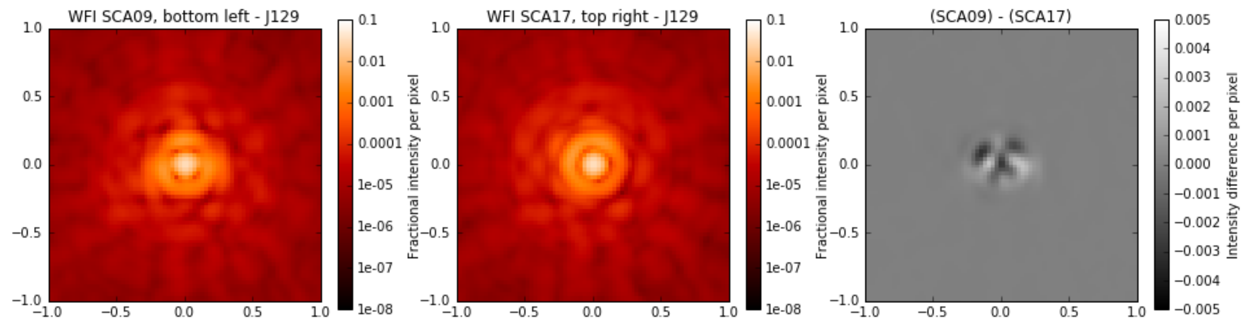


Fig. 2: This figure shows oversampled PSFs in the J129 filter at two different field points, and the intensity difference image between the two.

5.2 Coronagraph Instrument (CGI)

We have begun developing a Coronagraph Instrument (CGI) simulation module. The goal is to provide an open source modeling package for CGI for use by the science centers and science teams, to complement the existing in-house optical modeling capabilities at JPL.

Currently a prototype implementation is available for the shaped pupil coronagraph modes only, for both the CGI imager and IFS. Future releases will incorporate realistic aberrations, both static and dynamic, to produce realistic speckle fields. We also plan to add the hybrid Lyot modes.

Warning: Current functionality is limited to the Shaped Pupil Coronagraph (SPC) observing modes, and these modes are only simulated with static, unaberrated wavefronts, without relay optics and without DM control. The design respresented here is an approximation to a baseline concept, and will be subject to change based on ongoing trades studies and technology development.

A hands-on tutorial in using the CGI class is available in this Jupyter Notebook. Here we briefly summarize the key points, but see that for more detail.

The CGI class has attributes for `filter`, etc., like other instrument classes, but since these masks are designed to be used in specific combinations, a `mode` attribute exists that allows easy specification of all those attributes at once. For example, setting

::

>> cgi = wfirst.CGI() >> cgi.mode='CHARSPC_F770'

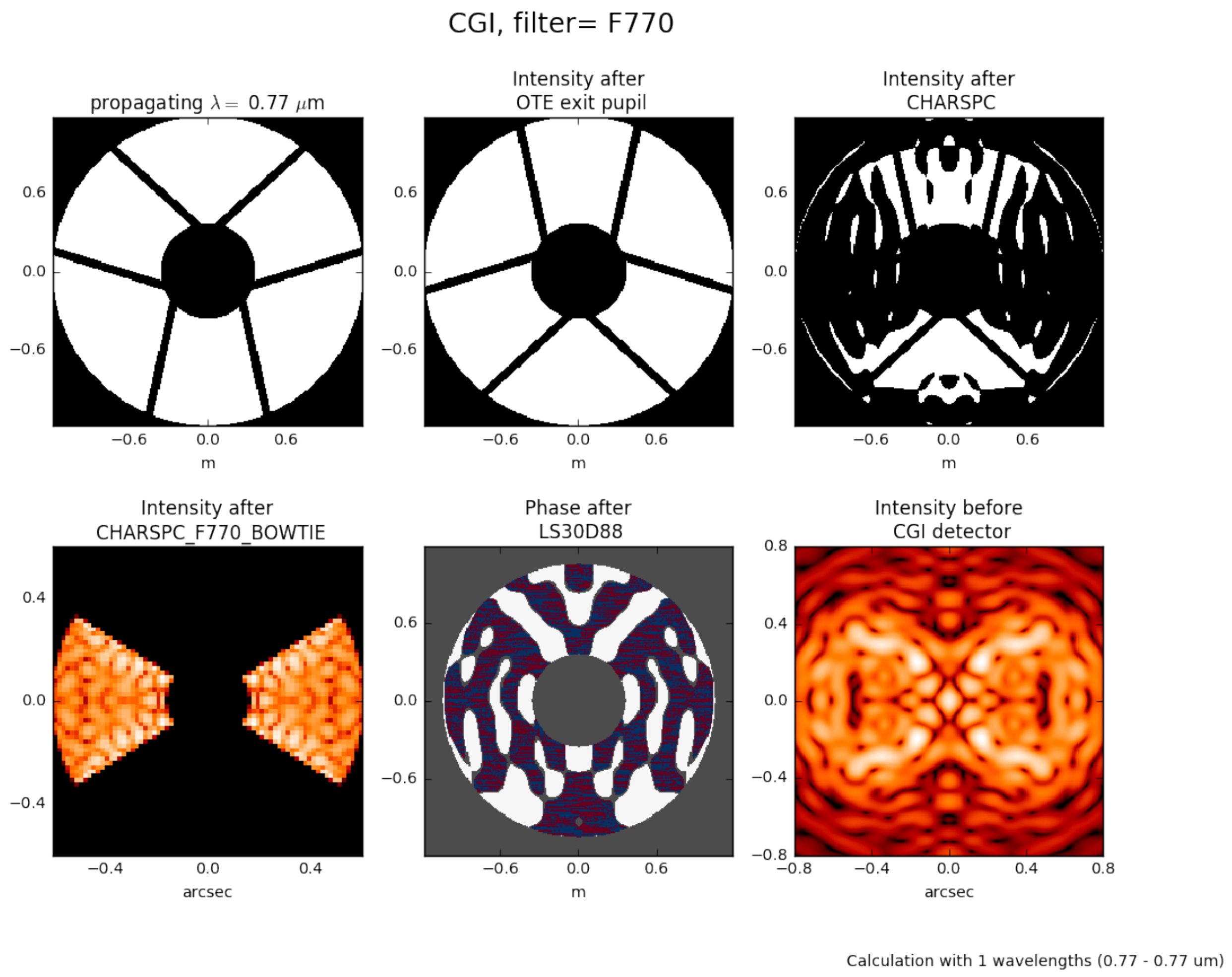
is equivalent to:

```
>> cgi.camera = 'IFS'
>> cgi.filter = 'F770'
>> cgi.apodizer = 'CHARSPC'
>> cgi.fpm = 'CHARSPC_F770_BOWTIE'
>> cgi.lyotstop = 'LS30D88'
```

There are `_list` attributes that tell you the allowed values for each attribute, including a `mode_list` for all the available meta-modes.

Calculations are invoked similarly to any other instrument class:

```
>> mono_char_spc_psf = cgi.calc_psf(nlambda=1, fov_arcsec=1.6, display=True)
```


CGI, filter= F770
propagating $\lambda =$ 0.77 μm
Intensity after OTE exit pupil
Intensity after CHARSPC
Intensity after CHARSPC_F770_BOWTIE
Phase after LS30D88
Intensity before CGI detector
m
arcsec
Calculation with 1 wavelengths (0.77 - 0.77 um)

CHAPTER 6

Using PSF Grids

WebbPSF includes functionality designed to work with the Photutils package to enable precise PSF-fitting photometry and astrometry. This makes use of the `GriddedPSFModel` class (available in Photutils > 0.6), which implements a version of the empirical or effective PSF ("ePSF") modeling framework pioneered by Jay Anderson, Ivan King, and collaborators. This approach has been highly successful with HST and other space observatories, and we expect it will also be productive with JWST. In practice we will want to use ePSF models derived from real observations, but for now we can make them in simulation.

The first step is to create a grid of fiducial PSFs spanning the instrument/detector of choice. This can be done using the `psf_grid()` method which will output a (list of or single) photutils GriddedPSFModel object(s). Users can then use photutils to apply interpolation to the grid to simulate a spatially dependent PSF anywhere on the instrument, without having to perform a full PSF calculation at each location. This faster approach is critical if you're dealing with potentially tens of thousands of stars scattered across many megapixels of detector real estate.

Jupyter Notebook

See this Gridded PSF Library tutorial notebook for more details and example code.

6.1 Example PSF grid

PSF grid calculations are useful for visualizing changes in the PSF across instrument fields of view. Here's one example of that.

```
nrc = webbpsf.NIRCam()
nrc.filter='F212N'
nrc.detector='NRCA3'
grid = nrc.psf_grid(num_psfs=36, all_detectors=False)
webbpsf.gridded_library.display_psf_grid(grid)
```

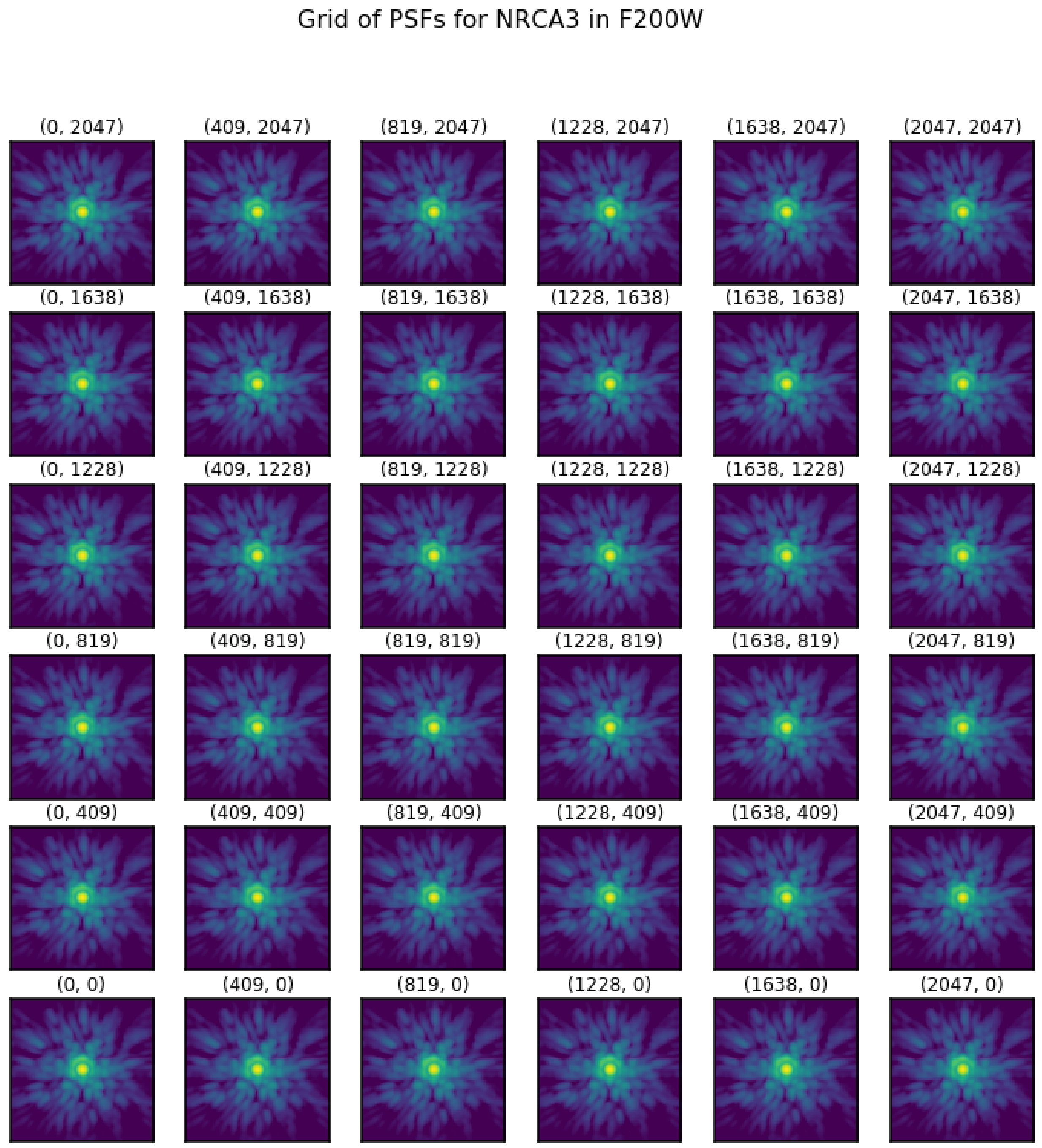


Fig. 1: An example of grid calculated across the NRCA3 detector in NIRCam. These PSFs are all very similar.

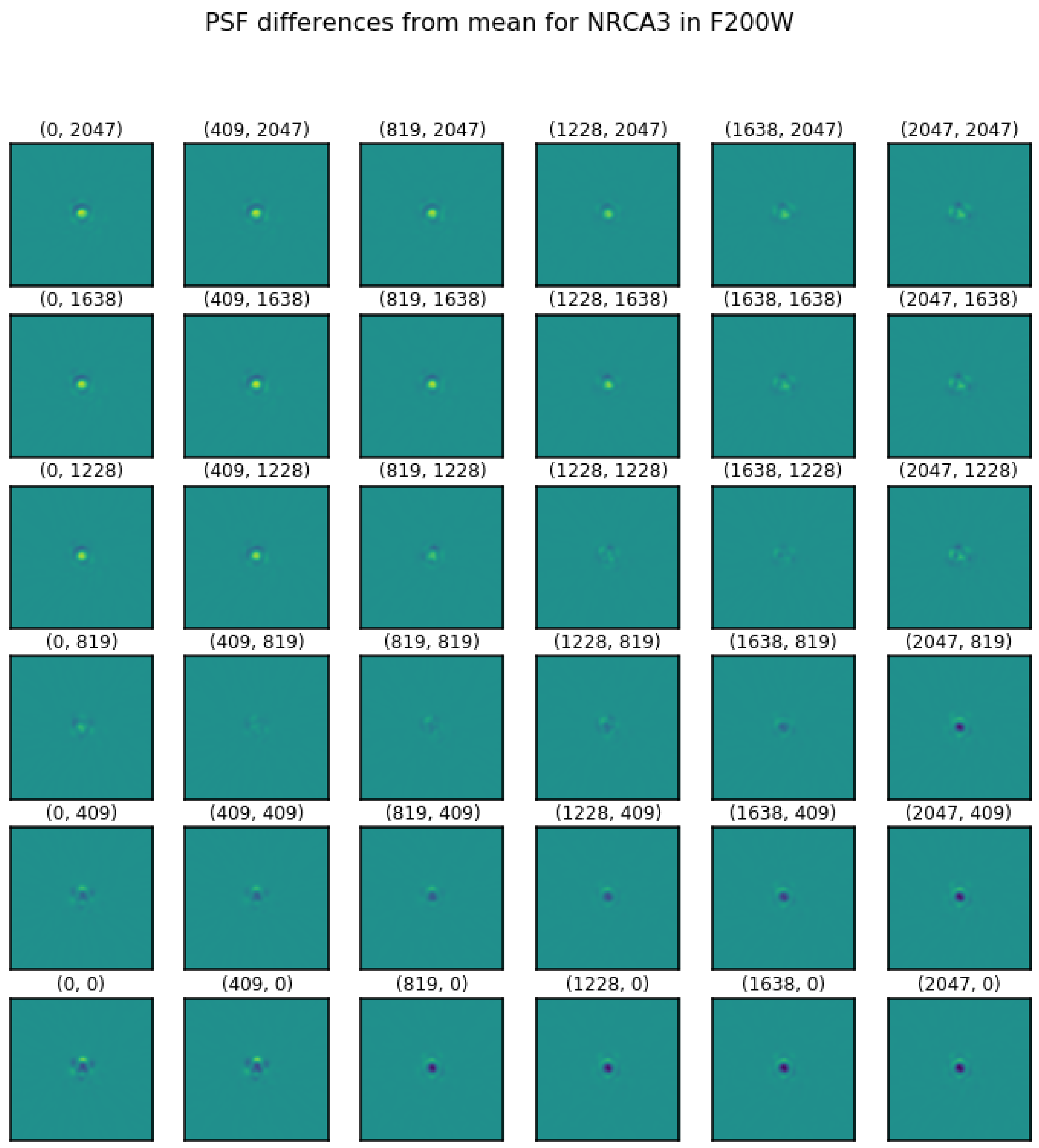


Fig. 2: By subtracting off the average PSF, the subtle differences from point to point become clear. The PSF is sharpest in the upper left corner of this detector.

CHAPTER 7

Using WebbPSF via the Graphical User Interface

The WebbPSF GUI provides an easy interface to much, but not all, of the functionality of WebbPSF. (Many advanced settings in the class attributes and `options` structure for `webbpsf.JWInstrument` are not exposed in the GUI. The programming API is also much better suited for scripting calculations.)

Attention: The GUI is mostly not actively developed any more, and is not recommended for more than basic use cases. It's a nice way to play around, but we encourage almost everyone to use the Python API as their primary way of interacting with WebbPSF. The GUI may be deprecated in a future release.

7.1 Using the Graphical Interface

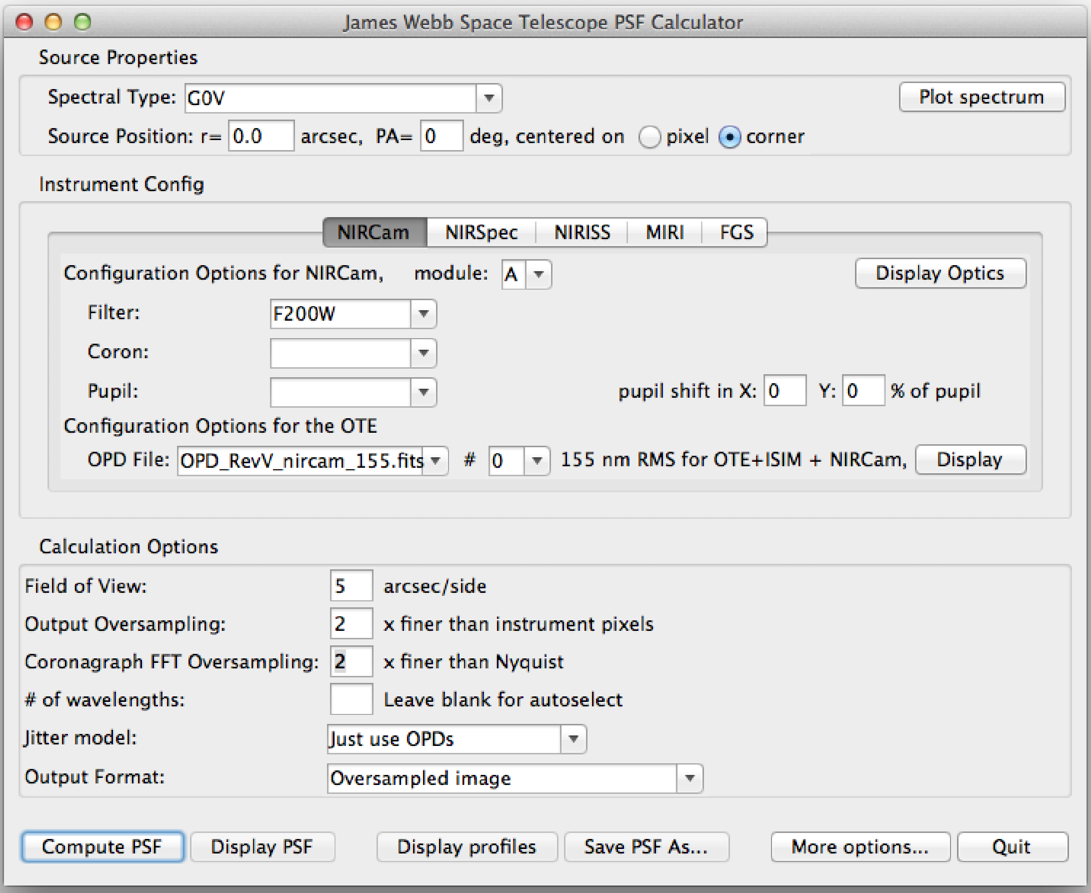
Once you have *installed WebbPSF*, you should have the launcher script `webbpsfgui` available. (If not, verify that the WebbPSF installation directory is on your system `$PATH`.)

Alternatively, you may start the GUI from an interactive session:

```
>>> import matplotlib
>>> matplotlib.use('TkAgg')
>>> import webbpsf
>>> webbpsf.gui()
```

The main window is divided into three regions:

- The top region allows control of the source spectral type and position. (Selecting a source spectral type requires installing the optional dependency *pysynphot*.)
- The central, main region allows selection of instrument and configuration of instrument options. The options available here largely correspond to attributes of the `webbpsf.JWInstrument` classes.
- The lower region contains options for the PSF calculation itself such as pixel sampling and wavelengths. These correspond to parameters of the `webbpsf.JWInstrument.calc_psf()` function call.



Source Properties

Spectral Type: G0V

Source Position: r= 0.0 arcsec, PA= 0 deg, centered on ( ) pixel [✓ corner]

Instrument Config

Configuration Options for NIRCam, module: A

Filter: F200W

Coron:

Pupil: pupil shift in X: 0 Y: 0 % of pupil

Configuration Options for the OTE

OPD File: OPD_RevV_nircam_155.fits # 0 155 nm RMS for OTE+ISIM + NIRCam,

Calculation Options

Field of View: 5 arcsec/side

Output Oversampling: 2 x finer than instrument pixels

Coronagraph FFT Oversampling: 2 x finer than Nyquist

# of wavelengths: Leave blank for autoselect

Jitter model: Just use OPDs

Output Format: Oversampled image

Fig. 1: The main window of `webbpsfgui` when first launched.

7.2 GUI Controls

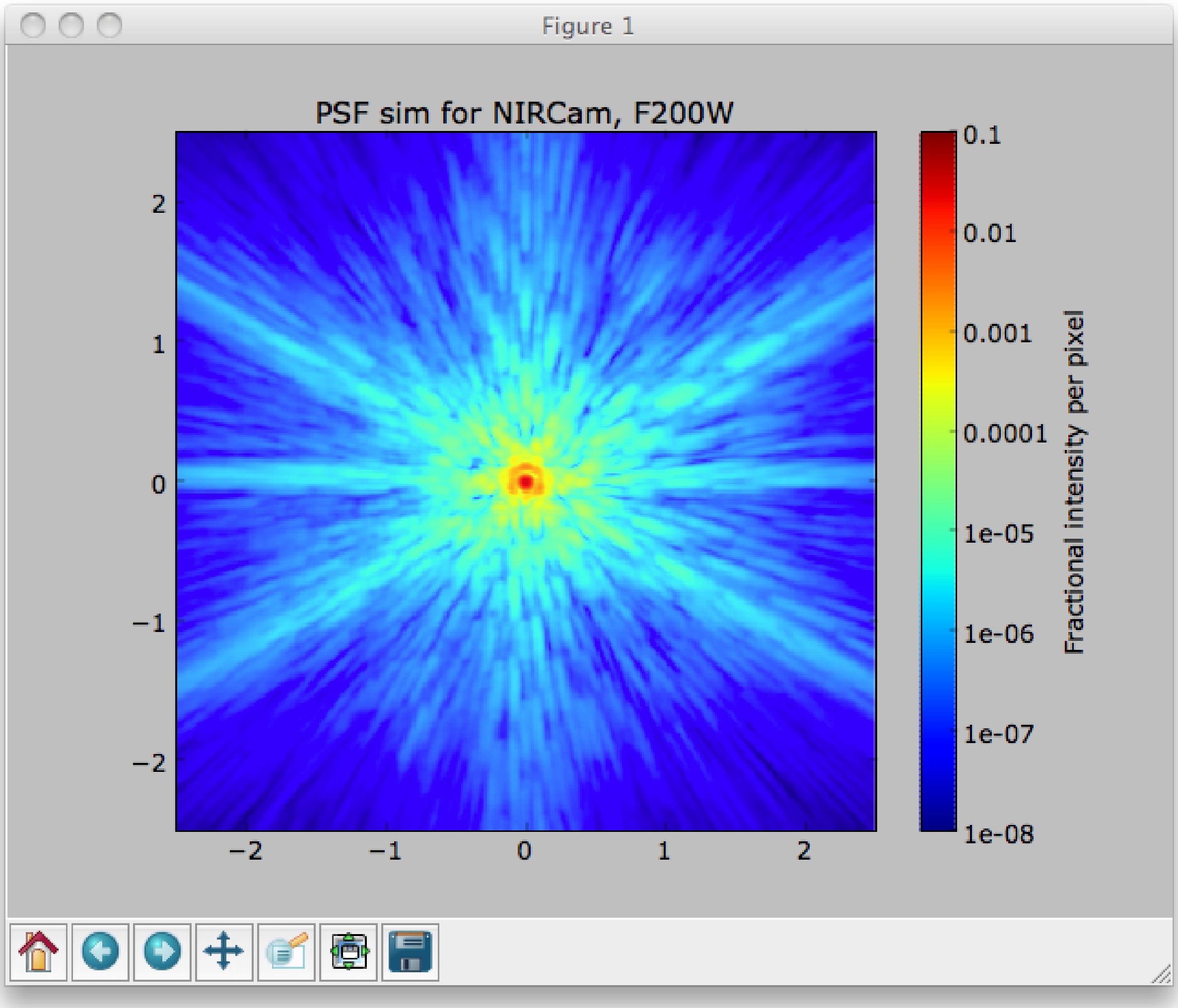
The GUI buttons invoke actions as follows:

7.2.1 Compute PSF

This invokes a PSF calculation with the given options. Each wavelength will be displayed in turn as it is computed, and finally the summed broadband PSF. This resulting PSF is stored in memory for use by the next three buttons.

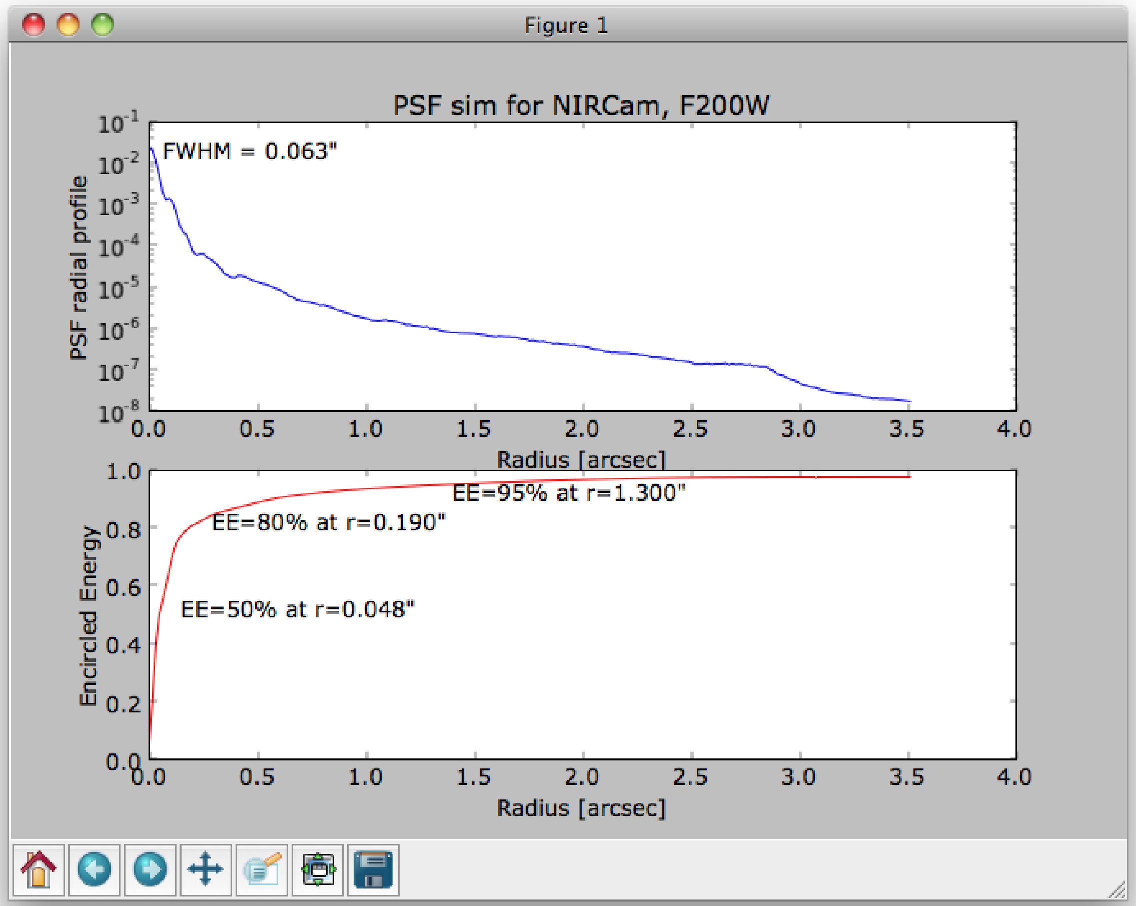
7.2.2 Display PSF

This button will redisplay the PSF if the window has closed or something else has been displayed.



7.2.3 Display Profiles

This will display the PSF's radial profile and encircled energy profile.

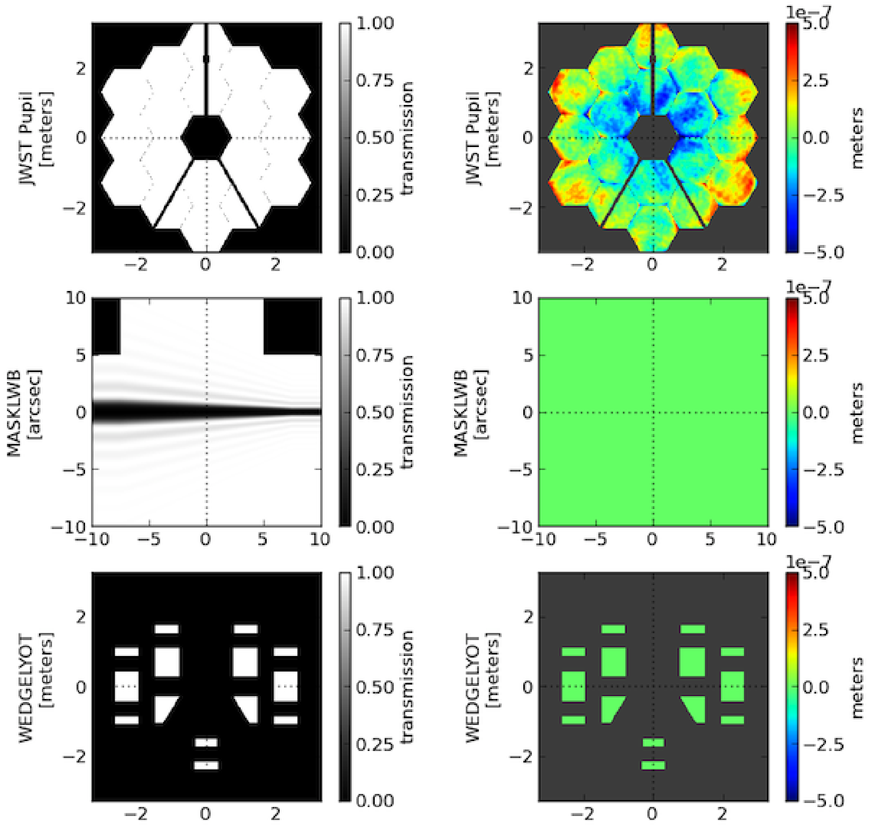


7.2.4 Save PSF As…

This will invoke a standard File Save dialog box allowing you to save your new PSF to a FITS file.

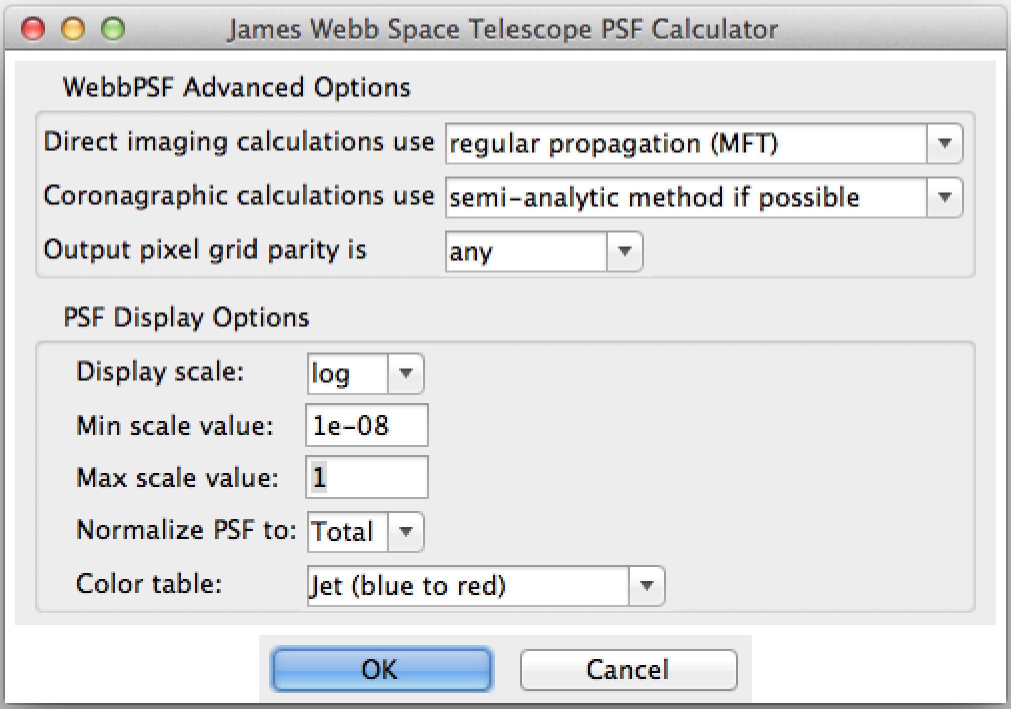
7.2.5 Display Optics

This will display a graphical representation of the optical train for the current instrument configuration.



7.2.6 More Options…

The 'More Options…' button on the toolbar will bring up a window that allows you to select options controlling the computation of the PSF (e.g. which Fourier transform algorithm is used) or display of the PSF (e.g. which color map to use).



WebbPSF Advanced Options

Direct imaging calculations use: regular propagation (MFT)

Coronagraphic calculations use: semi-analytic method if possible

Output pixel grid parity is: any

PSF Display Options

Display scale: log

Min scale value: 1e-08

Max scale value: 1

Normalize PSF to: Total

Color table: Jet (blue to red)

7.3 Troubleshooting

Caution: Matplotlib Back End Issues

On macOS, some users have encountered problems running the GUI due to incompatibilities with Matplotlib backends. If you see a severe error when trying to start the gui, try switching the backend to “TkAgg” rather than the default “MacOSX”. This needs to be done immediately after starting IPython, prior to any attempt to use the GUI, and ideally before even importing webbpsf:

```
import matplotlib
matplotlib.use('TkAgg')
import webbpsf
```

CHAPTER 8

More Examples

Any user of Webbpsf is invited to submit snippets of example code for sharing here.

This code is also available as a Jupyter notebook. This version of the page is kept for convenience but may be slightly out of date in a few places.

Examples are organized by topic:

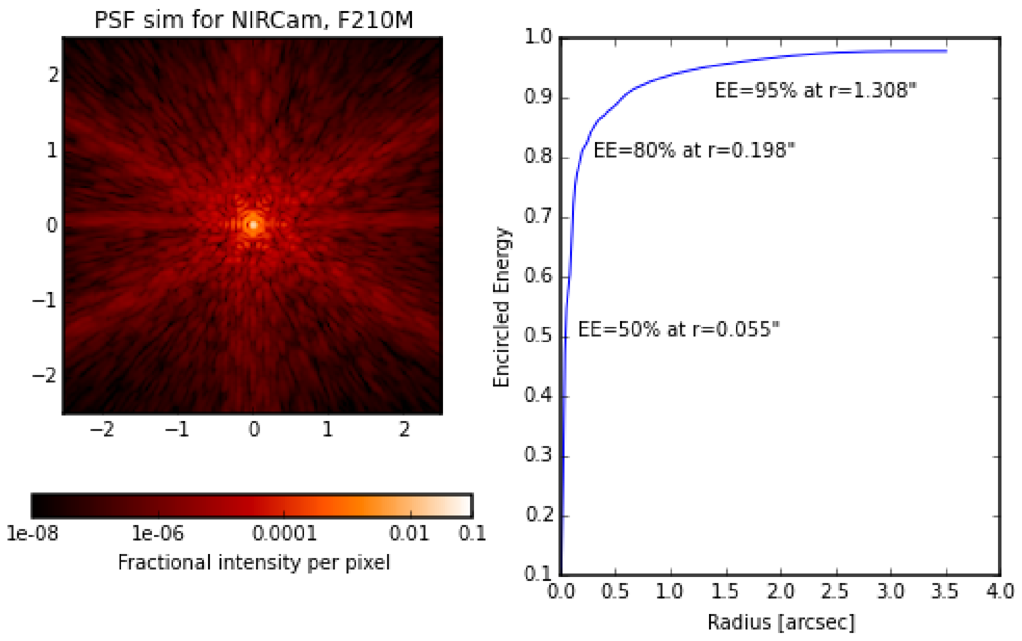
- *Typical Usage Cases*
- *Spectroscopic PSFs, Slit and Slitless*
- *Coronagraphy and Complications*

The notebook version of this page includes a fourth section providing examples of all the major SI modes for each of the JWST instruments.

8.1 Typical Usage Cases

8.1.1 Displaying a PSF as an image and as an encircled energy plot

```
#create a NIRCam instance and calculate a PSF for F210M
nircam = webbpsf.NIRCam()
nircam.filter = 'F210M'
psf210 = nircam.calc_psf(oversample=2)

# display the PSF and plot the encircled energy
plt.subplot(1,2,1)
webbpsf.display_psf(psf210, colorbar_orientation='horizontal')
axis2 = plt.subplot(1,2,2)
webbpsf.display_ee(psf210, ax=axis2)

psf210.writeto('nircam_F210M.fits')
plt.savefig('plot_nircam_f210m.pdf')
```



8.1.2 Iterating over multiple OPDs and filters

Perhaps you want to calculate PSFs for all filters of a given instrument, using all 10 available simulated OPDs:

```
def niriss_psfs():
    niriss = webbpsf.NIRISS()

    opdname = niriss.pupilopd

    for i in range(10):
        niriss.pupilopd = (opdname,i)
        for filtname in niriss.filter_list:
            niriss.filter=filtname
            fov=18
            outname = "PSF_NIRISS_%scen_wfe%d.fits" % (filtname, i)
            psf = niriss.calc_psf(outname, nlambda=1, oversample=4, fov_arcsec=fov, rebin=True,␣
↪display=True)
```

8.1.3 Create monochromatic PSFs across an instrument’s entire wavelength range

Monochromatic PSFs with steps of 0.1 micron from 5-28.3 micron.

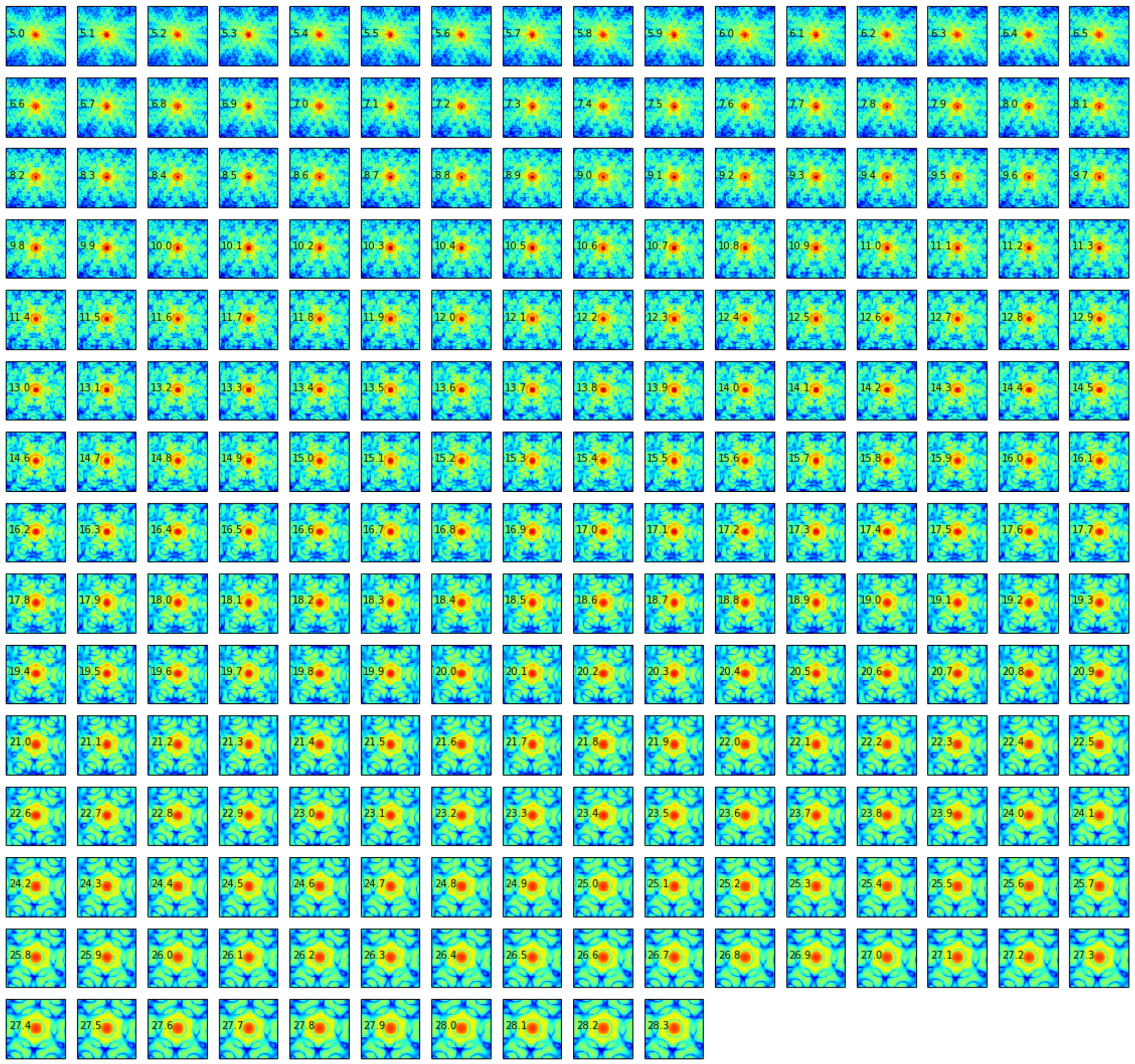
```
m = webbpsf.MIRI()
m.pupilopd = 'OPD_RevW_ote_for_MIRI_requirements.fits.gz'       # select an OPD
                                            # looks inside $WEBBPSF_DATA/MIRI/OPD by default
                                             # or you can specify a full path name.
m.options['parity'] = 'odd'                 # please make an output PSF with its center
                                             # aligned to the center of a single pixel

waves = np.linspace(5.0, 28.3, 234)*1e-6     # iterate over wavelengths in meters
```

(continues on next page)

(continued from previous page)

```
#waves = np.linspace(5.0, 28.3, 20)*1e-6     # iterate over wavelengths in meters

for iw, wavelength in enumerate(waves):
    psffile = 'psf_MIRI_mono_%.1fum_opd1.fits' % (wavelength*1e6)
    psf = m.calc_psf(fov_arcsec=30, oversample=4, rebin=True, monochromatic=wavelength, display=False,
                outfile=psffile)
    ax = plt.subplot(16,16,iw+1)
    webbpsf.display_psf(psffile, ext='DET_SAMP', colorbar=False, imagecrop=8)
    ax.set_title('')
    ax.xaxis.set_visible(False)
    ax.yaxis.set_visible(False)
    ax.text(-3.5, 0, '{0:.1f}'.format(wavelength*1e6))
```

Click to enlarge:

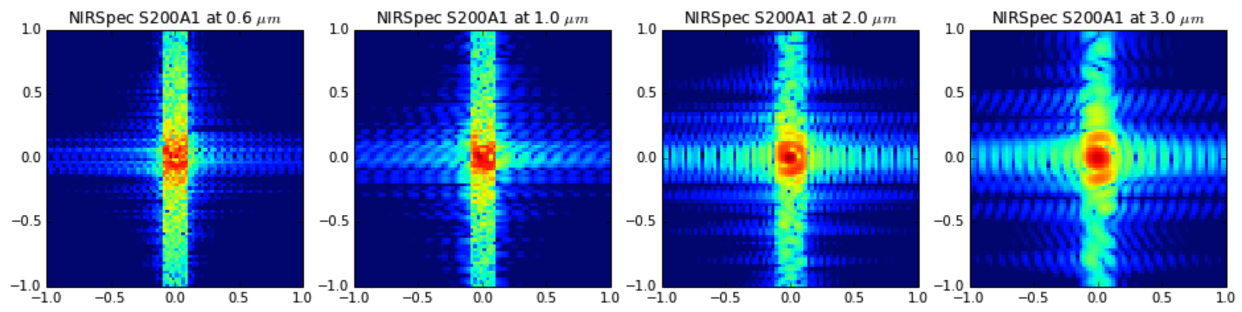


8.2 Spectroscopic PSFs, Slit and Slitless

Note that WebbPSF does not yet compute *dispersed* spectroscopic PSFs, but you can compute monochromatic PSFs and combine them yourself with an appropriate dispersion model.

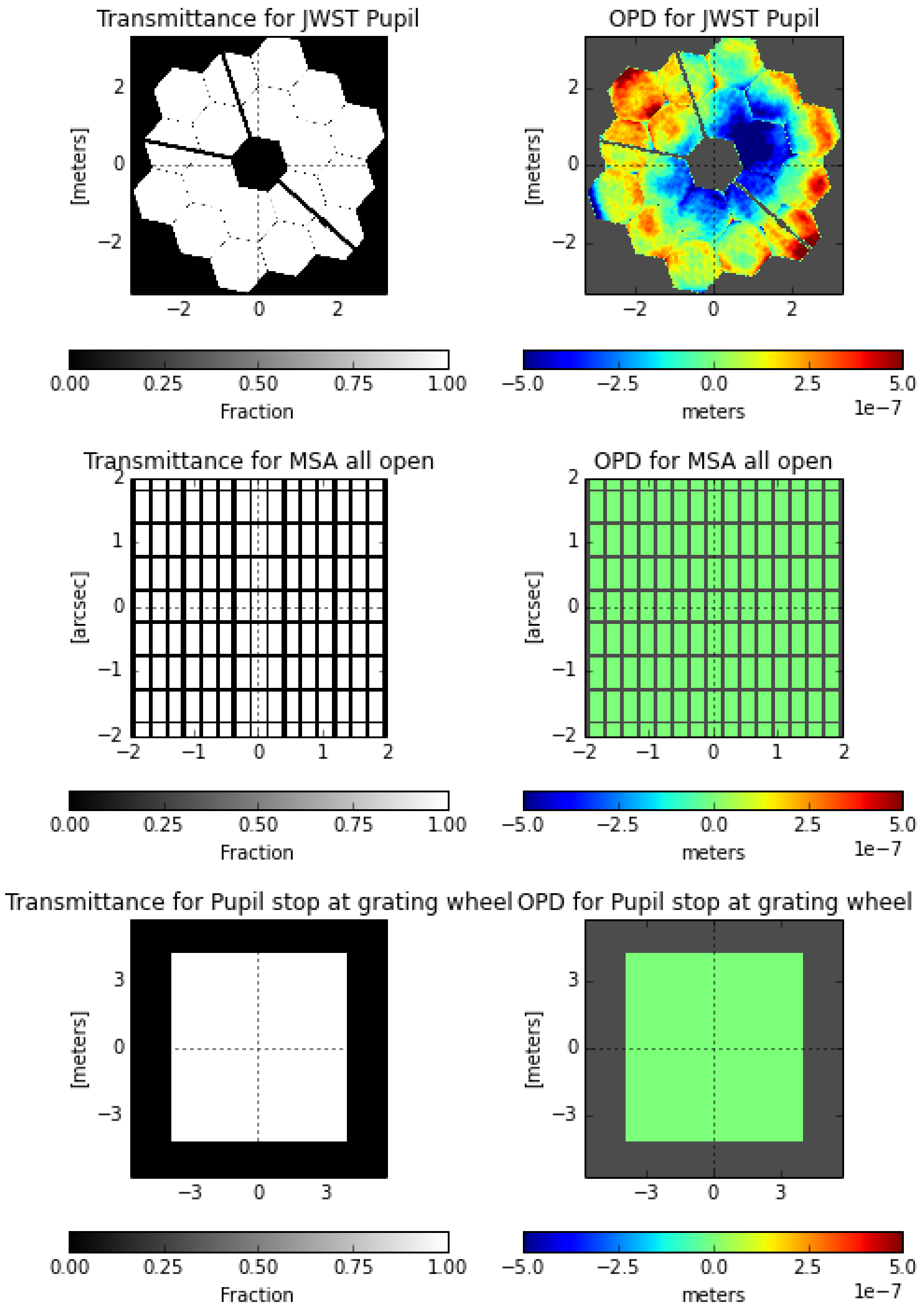
8.2.1 NIRSpec fixed slits

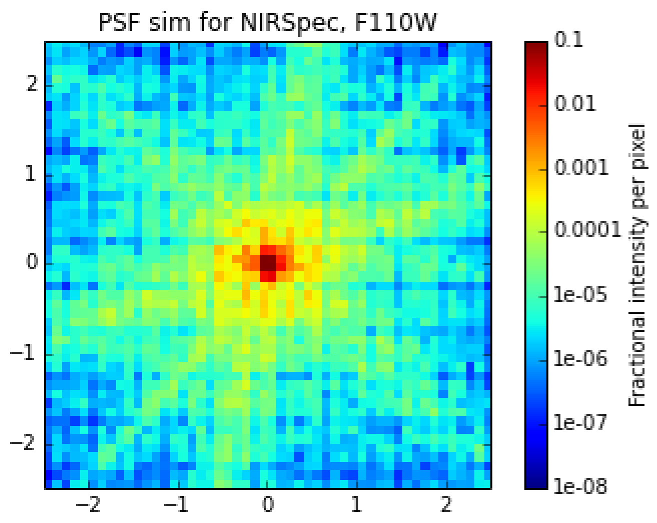
```
plt.figure(figsize=(8, 12))
nspec = webbpsf.NIRSpec()
nspec.image_mask = 'S200A1' # 0.2 arcsec slit

psfs = {}
for wave in [0.6e-6, 1e-6, 2e-6, 3e-6]:
    psfs[wave] = nspec.calc_psf(monochromatic=wave, oversamp=4)

for i, wave in enumerate([0.6e-6, 1e-6, 2e-6, 3e-6]):
    plt.subplot(1, 4, i+1)
    webbpsf.display_psf(psfs[wave], colorbar=False, imagecrop=2, title='NIRSpec S200A1 at {0:.1f} $\mu m
↪$'.format(wave*1e6))
plt.savefig('example_nirspec_slitpsf.png')
```



8.2.2 NIRSpec MSA

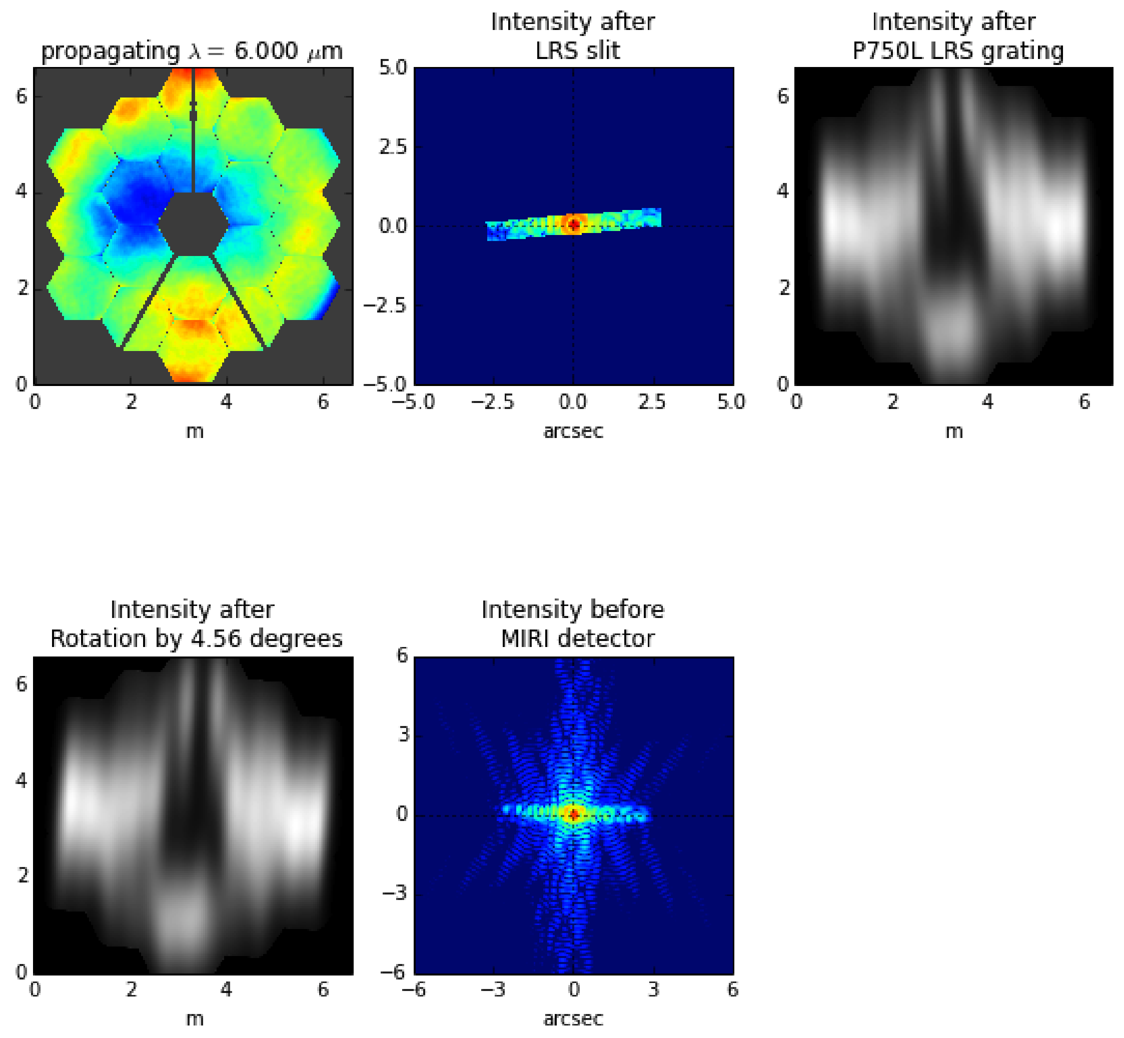
```
plt.figure(figsize=(8, 12))
ns = webbpsf.NIRSpec()
ns.image_mask='MSA all open'
ns.display()
plt.savefig('example_nirspec_msa_optics.png')
msapsf = ns.calc_psf(monochromatic=2e-6, oversample=8, rebin=True)
webbpsf.display_psf(msapsf, ext='DET_SAMP')
```


Transmittance for JWST Pupil
OPD for JWST Pupil
[meters]
Fraction
meters
1e−7
Transmittance for MSA all open
OPD for MSA all open
[arcsec]
Transmittance for Pupil stop at grating wheel
OPD for Pupil stop at grating wheel



8.2.3 MIRI LRS

```
miri = webbpsf.MIRI()
miri.image_mask = 'LRS slit'
miri.pupil_mask = 'P750L LRS grating'
psf = miri.calc_psf(monochromatic=6.0e-6, display=True)
```



8.3 Coronagraphy and Complications

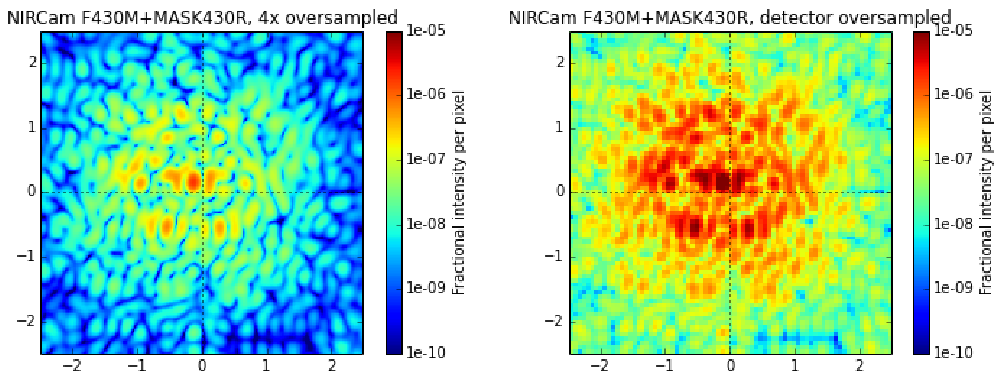
8.3.1 NIRCam coronagraphy with an offset source

```
nc = webbpsf.NIRCam()
nc.filter='F430M'
nc.image_mask='MASK430R'
nc.pupil_mask='CIRCLYOT'
nc.options['source_offset_r'] = 0.20       # source is 200 mas from center of coronagraph
                                           # (note that this is MUCH larger than expected acq
                                           # offsets. This size displacement is just for show)
nc.options['source_offset_theta'] = 45     # at a position angle of 45 deg
nc.calc_psf('coronagraphic.fits', oversample=4, clobber=True)   # create highly oversampled output image


plt.figure(figsize=(12,4))
plt.subplot(1,2,1)
```

(continues on next page)

(continued from previous page)

```
webbpsf.display_psf('coronagraphic.fits', vmin=1e-10, vmax=1e-5,
    ext='OVERSAMP', title='NIRCam F430M+MASK430R, 4x oversampled', crosshairs=True)
plt.subplot(1,2,2)
webbpsf.display_psf('coronagraphic.fits', vmin=1e-10, vmax=1e-5,
    ext='DET_SAMP', title='NIRCam F430M+MASK430R, detector oversampled', crosshairs=True)

plt.savefig('example_nircam_coron_resampling.png')
```



8.3.2 Simulate NIRCam coronagraphic acquisition images

```
def compute_psfs():
    nc = webbpsf.NIRCam()

    # acq filter, occulting mask, lyot, coords of acq ND square
    sets = [('F182M', 'MASKSWB', 'WEDGELYOT', -10,  7.5),
            ('F182M', 'MASK210R', 'CIRCLYOT', -7.5, 7.5),
            ('F335M', 'MASKLWB', 'WEDGELYOT',  7.5, 7.5),
            ('F335M', 'MASK335R', 'CIRCLYOT', -10,  7.5)]

    nlambda = 9
    oversample = 2

    calc_oversample=4

    fov_arcsec = 25

    for param in sets:
        nc.filter = param[0]
        nc.image_mask = param[1]
        nc.pupil_mask = param[2]
        source_offset_x = param[3]
        source_offset_y = param[4]


        source_offset_r = np.sqrt(source_offset_x**2+ source_offset_y**2)
        source_offset_theta = np.arctan2(source_offset_x, source_offset_y)*180/np.pi
        nc.options['source_offset_r'] = source_offset_r
        nc.options['source_offset_theta'] = source_offset_theta
```

(continues on next page)

(continued from previous page)

```
        filename = "PSF_NIRCam_%s_%s_%s_offset.fits" % (param[0], param[1], param[2])
        result = nc.calc_psf(nlambda=nlambda,
            oversample=oversample, calc_oversample=calc_oversample,
            fov_arcsec=fov_arcsec, outfile=filename, display=False)
```

8.3.3 Iterate a calculation over all MIRI coronagraphic modes

```
def miri_psfs_coron():
    miri = webbpsf.MIRI()

    for filtwave in [1065, 1140, 1550, 2300]:

        miri.filter='F%4dC' % filtwave
        if filtwave<2000:
            miri.image_mask='FQPM%4d' % filtwave
            miri.pupil_mask='MASKFQPM'
            fov=24
        else:
            miri.image_mask='LYOT2300'
            miri.pupil_mask='MASKLYOT'
            fov=30


        offset_x = 0.007 # arcsec
        offset_y = 0.007 # arcsec

        miri.options['source_offset_r'] = np.sqrt(offset_x**2+offset_y**2) # offset in arcsec
        miri.options['source_offset_theta'] = np.arctan2(-offset_x, offset_y)*180/np.pi # PA in deg


        outname = "PSF_MIRI_%s_x%+05.3f_y%+05.3f.fits" % (miri.image_mask, offset_x, offset_y)
        psf = miri.calc_psf(outname, oversample=4, fov_arcsec=fov, display=True)
```

8.3.4 Make plots of encircled energy in PSFs at various wavelengths

```
def miri_psfs_for_ee():
    miri = webbpsf.MIRI()

    opdname = miri.pupilopd

    for i in range(10):
        miri.pupilopd = (opdname,i)
        for wave in [5.0, 7.5, 10, 14]:

            fov=18

            outname = "PSF_MIRI_%.1fum_wfe%d.fits" % (wave, i)
            psf = miri.calc_psf(outname, monochromatic=wave*1e-6,
                    oversample=4, fov_arcsec=fov, rebin=True, display=True)
```

(continues on next page)

(continued from previous page)

```
def plot_ee_curves():
    plt.clf()
    for iw, wave in enumerate([5.0, 7.5, 10, 14]):

        ees60 = []
        ees51 = []
        ax = plt.subplot(2,2,iw+1)
        for i in range(10):
            name = "PSF_MIRI_%.1fum_wfe%d.fits" % (wave, i)
            webbpsf.display_ee(name, ax=ax, mark_levels=False)

            eefn = webbpsf.measure_ee(name)
            ees60.append(eefn(0.60))
            ees51.append(eefn(0.51))

        ax.text(1, 0.6, 'Mean EE inside 0.60": %.3f' % np.asarray(ees60).mean())
        ax.text(1, 0.5, 'Mean EE inside 0.51": %.3f' % np.asarray(ees51).mean())

        ax.set_title("Wavelength = %.1f $\mu$m" % wave)

        ax.axvline(0.6, ls=":", color='k')
        ax.axvline(0.51, ls=":", color='k')


    plt.tight_layout()
```

8.3.5 Simulate coronagraphy with pupil shear, saving the wavefront intensity in the Lyot pupil plane

This is an example of a much more complicated calculation, including code to generate publication-quality plots.

There are two functions here, one that creates a simulated PSF for a given amount of shear, and one that makes some nice plots of it.

```
def miri_psf_sheared(shearx=0, sheary=0, nopds = 1, display=True, overwrite=False, \*\*kwargs):
    """ Compute MIRI coronagraphic PSFs assuming pupil shear between the MIRI lyot mask and the OTE

    Parameters
    ------------
    shearx, sheary: float
        Shear across the pupil expressed in percent,
        i.e. shearx=3 means the coronagraph pupil is sheared by 3% of the primary.

    """
    miri = webbpsf.MIRI()

    miri.options['pupil_shift_x'] = shearx/100 # convert shear amount to float between 0-1
    miri.options['pupil_shift_y'] = sheary/100

    opdname = miri.pupilopd         # save default OPD name for use in iterating over slices

    filtsets = [('F1065C', 'FQPM1065', 'MASKFQPM'), ('F2300C','LYOT2300','MASKLYOT')]

    fov=10

```

(continues on next page)

(continued from previous page)

```
    for i in range(nopds):
        miri.pupilopd = (opdname,i)
        for filt, im_mask, pup_mask in filtsets:
            print("Now computing OPD %d for %s, %s, %s" % (i, filt, im_mask, pup_mask))
            miri.filter=filt
            miri.image_mask = im_mask
            miri.pupil_mask = pup_mask


            outname = "PSF_MIRI_%s_wfe%d_shx%.1f_shy%.1f.fits" % (filt, i, shearx, sheary)
            outname_lyot = outname.replace("PSF_", 'LYOTPLANE_')


            if os.path.exists(outname) and not overwrite:
                print ("File %s already exists. Skipping and continuing for now... "
                       " Set overwrite=True to recalculate" % outname)
                return

            psf, intermediates = miri.calc_psf(oversample=4, fov_arcsec=fov,
                    rebin=True, display=display, return_intermediates=True, \*\*kwargs)

            lyot_intensity = intermediates[4]

            psf.writeto(outname, clobber=True)
            lyot_intensity.writeto(outname_lyot, clobber=True, includepadding=False)


def plot_sheared_psf(shearx=1.0, sheary=0, lyotmax=1e-5, psfmax = 1e-3, diffmax=10):
    i = 0
    filtsets = [('F1065C', 'FQPM1065', 'MASKFQPM')]#, ('F2300C','LYOT2300','MASKLYOT')]

    plt.clf()
    plt.subplots_adjust(left=0.02, right=0.98, wspace=0.3)
    for filt, im_mask, pup_mask in filtsets:
        perfectname = "PSF_MIRI_%s_wfe%d_shx%.1f_shy%.1f.fits" % (filt, i, 0,0)
        perfectname_lyot = perfectname.replace("PSF_", 'LYOTPLANE_')


        outname = "PSF_MIRI_%s_wfe%d_shx%.1f_shy%.1f.fits" % (filt, i, shearx, sheary)
        outname_lyot = outname.replace("PSF_", 'LYOTPLANE_')

        if not os.path.exists(outname):
            print "File %s does not exist, skipping" % outname
            return False


        #psf = pyfits.open(outname)
        #perfpsf = pyfits.open(perfectname)
        lyot = pyfits.open(outname_lyot)
        perflyot = pyfits.open(perfectname_lyot)

        wzero = np.where(lyot[0].data == 0)
        wzero = np.where(lyot[0].data < 1e-15)
        lyot[0].data[wzero] = np.nan
        wzero = np.where(perflyot[0].data == 0)
        perflyot[0].data[wzero] = np.nan
```

(continues on next page)

(continued from previous page)

```
        cmap = matplotlib.cm.jet
        cmap.set_bad('gray')



        # plot comparison perfect case Lyot Intensity
        ax = plt.subplot(231)
        plt.imshow(perflyot[0].data, vmin=0, vmax=lyotmax, cmap=cmap)
        plt.title("Lyot plane, no shear")
        ax.yaxis.set_ticklabels("")
        ax.xaxis.set_ticklabels("")

        wg = np.where(np.isfinite(perflyot[0].data))
        ax.set_xlabel("Residual flux = %.1f%%" % (perflyot[0].data[wg].sum()*100))

        # plot shifted pupil Lyot intensity
        ax = plt.subplot(234)
        plt.imshow(lyot[0].data, vmin=0, vmax=lyotmax, cmap=cmap)
        plt.title("Lyot plane, shear (%.1f, %.1f)" % (shearx, sheary))
        ax.yaxis.set_ticklabels("")
        ax.xaxis.set_ticklabels("")
        wg = np.where(np.isfinite(lyot[0].data))
        ax.set_xlabel("Residual flux = %.1f%%" % (lyot[0].data[wg].sum()*100))



        # Radial profile plot
        plt.subplot(233)

        radius, profperf = webbpsf.radial_profile(perfectname, ext=1)
        radius2, profshear = webbpsf.radial_profile(outname, ext=1)

        # normalize all radial profiles to peak=1 for an unocculted source
        radiusu, profunocc = webbpsf.radial_profile('PSF_MIRI_F1065C_wfe0_noshear_unocculted.fits',
            ext=1, center=(43.3, 68.6)) # center is in pixel coords

        peakunocc = profunocc.max()
        profperf /= peakunocc
        profshear/= peakunocc
        profunocc/= peakunocc


        plt.semilogy(radius, profperf, label="No shear")
        plt.semilogy(radius2, profshear, label="shear (%.1f, %.1f)" % (shearx, sheary))
        plt.semilogy(radiusu, profunocc, label="Unocculted", ls=":" )


        plt.xlabel("Separation [arcsec]")
        plt.ylabel("Relative Intensity")
        plt.legend(loc='upper right')
        plt.gca().set_xlim(0,6)


        # plot comparison perfect case PSF - detector sampled
        plt.subplot(232)
        webbpsf.display_psf(perfectname, ext=1, vmax=psfmax)
        plt.title("PSF, no shear")
```

(continues on next page)

(continued from previous page)

```
        # plot shifted pupil PSF - detector sampled
        plt.subplot(235)
        webbpsf.display_psf(outname, ext=1, vmax=psfmax)
        plt.title("PSF, shear (%.1f, %1.f)" % (shearx, sheary))
        plt.xlabel("Separation [arcsec]")
        # difference PSf
        plt.subplot(236)
        webbpsf.display_psf_difference(outname, perfectname, ext1=1,
            ext2=1, vmax=diffmax, vmin=-0.1, normalize_to_second=True)
        plt.title('Relative PSF increase')
        plt.xlabel("Separation [arcsec]")


        return True
```

CHAPTER 9

Overview of POPPY (Physical Optics Propagation in Python)

POPPY, which stands for Physical Optics Propagation in Python, 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.)

Note: This is an *abbreviated* version of the documentation for POPPY, included here as a brief summary relevant for WebbPSF. For more comprehensive documentation for POPPY please see the full POPPY documentation

9.1 Introduction

The POPPY functionality lives under the package name `poppy`, which is available separately from WebbPSF and contains general functionality for Fraunhofer domain optical simulation. WebbPSF uses POPPY under the hood to perform calculations, and indeed POPPY began its life as part of WebbPSF.

POPPY 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.

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.

9.2 The Object-Oriented Optical Model

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 `OpticalElements` is chained together in order in an `OpticalSystem` class. That class is capable of generating `Wavefronts` (another class) suitable for propagation through the desired elements (with correct array size and sampling), and then performing the optical propagation 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. All of the WebbPSF instruments are implemented by subclassing `poppy.Instrument`.

9.3 Algorithms, Approximations, and Performance

POPPY presently assumes that optical propagation can be modeled using Fraunhofer diffraction (far-field), such that the relationship between pupil and image plane optics is given by two-dimensional Fourier transforms. Fresnel propagation is not currently supported.

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.

Getting Help

For help using or installing webbpsf, you can contact the STScI Help Desk, help@stsci.edu. Note that WebbPSF is included in the Astroconda python distribution, as well as being installable via *standard Python packaging tools*..

Part II

Advanced Usage

CHAPTER 10

Detailed API Reference

10.1 webbpsf Package

10.1.1 WebbPSF: Simulated Point Spread Functions for the James Webb Space Telescope

WebbPSF produces simulated PSFs for the James Webb Space Telescope, NASA's next flagship infrared space telescope. WebbPSF can simulate images for any of the four science instruments plus the fine guidance sensor, including both direct imaging and coronagraphic modes.

Developed by Marshall Perrin and collaborators at STScI, 2010-2018.

Documentation can be found online at https://webbpsf.readthedocs.io/

10.1.2 Functions

`Instrument`(name)	This is just a convenience function, allowing one to access instrument objects based on a string.
`display_ee`([HDUlist_or_filename, ext, . . .])	Display Encircled Energy curve for a PSF
`display_profiles`([HDUlist_or_filename, ext, . . .])	Produce two plots of PSF radial profile and encircled energy
`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
`enable_adjustable_ote`(instr[, jsc])	Set up a WebbPSF instrument instance to have a modifiable OTE wavefront error OPD via an OTE linear optical model (LOM).
`gui`([preferred])	Start the WebbPSF GUI with the selected interface
`measure_centroid`([HDUlist_or_filename, ext, . . .])	Measure the center of an image via center-of-mass

Continued on next page

Table 1 – continued from previous page

`measure_ee`([HDUlist_or_filename, ext, . . .])	measure encircled energy vs radius and return as an interpolator
`measure_fwhm`(HDUlist_or_filename[, ext, . . .])	Improved version of measuring FWHM, without any binning of image data.
`measure_radial`([HDUlist_or_filename, ext, . . .])	measure azimuthally averaged radial profile of a PSF.
`measure_sharpness`([HDUlist_or_filename, ext])	Compute image sharpness, the sum of pixel squares.
`measure_strehl`([HDUlist_or_filename, ext, . . .])	Estimate the Strehl ratio for a PSF.
`radial_profile`([hdulist_or_filename, ext, . . .])	Compute a radial profile of the image.
`restart_logging`([verbose])	Restart logging using the same settings as the last WebbPSF session, as stored in the configuration system.
`setup_logging`([level, filename])	Allows selection of logging detail and output locations (screen and/or file)
`show_notebook_interface`(instrumentname)	Show Jupyter notebook widget interface
`specFromSpectralType`(sptype[, return_list, . . .])	Get Pysynphot Spectrum object from a user-friendly spectral type string.
`system_diagnostic`()	return various helpful/informative information about the current system.
`test`([package, test_path, args, plugins, . . .])	Run the tests using py.test.
`tkgui`([fignum])	

Instrument

`webbpsf.`**`Instrument`**(*name*)

This is just a convenience function, allowing one to access instrument objects based on a string. For instance,

```
>>> t = Instrument('NIRISS')
```

name
[string] Name of the instrument class to return. Case insensitive.

enable_adjustable_ote

`webbpsf.`**`enable_adjustable_ote`**(*instr*, *jsc=False*)

Set up a WebbPSF instrument instance to have a modifiable OTE wavefront error OPD via an OTE linear optical model (LOM).

inst
[WebbPSF Instrument instance] an instance of one of the WebbPSF instrument classes.

jsc
[bool] Use ACF pupil for JSC pass and a half test configuration

a modified copy of that instrument set up to use the LOM, and the associated instance of the LOM.

gui

`webbpsf.`**`gui`**(*preferred='wx'*)

Start the WebbPSF GUI with the selected interface

preferred
[string] either 'wx' or 'ttk' to indicate which GUI toolkit should be started.

measure_strehl

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

Estimate the Strehl ratio for a PSF.

This requires computing a simulated PSF with the same properties as the one under analysis.

Note that this calculation will not be very accurate unless both PSFs are well sampled, preferably several times better than Nyquist. See Roberts et al. 2004 SPIE 5490 for a discussion of the various possible pitfalls when calculating Strehl ratios.

WARNING: This routine attempts to infer how to calculate a perfect reference PSF based on FITS header contents. It will likely work for simple direct imaging cases with WebbPSF but will not work (yet) for more complicated cases such as coronagraphy, anything with image or pupil masks, etc. Code contributions to add such cases are welcomed.

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

center
[tuple] center to compute around. Default is image center. If the center is on the crosshairs between four pixels, then the mean of those four pixels is used. Otherwise, if the center is in a single pixel, then that pixel is used.

verbose, display
[bool] control whether to print the results or display plots on screen.

cache_perfect
[bool] use caching for perfect images? greatly speeds up multiple calcs w/ same config

strehl
[float] Strehl ratio as a floating point number between 0.0 - 1.0

restart_logging

`webbpsf.`**`restart_logging`**(*verbose=True*)

Restart logging using the same settings as the last WebbPSF session, as stored in the configuration system.

verbose
[boolean] Should this function print the new logging targets to standard output?

setup_logging

`webbpsf.`**`setup_logging`**(*level='INFO'*, *filename=None*)

Allows selection of logging detail and output locations (screen and/or file)

This is a convenience wrapper to Python's built-in logging package, as used by `webbpsf` and poppy. By default, this sets up log messages to be written to the screen, but the user can also request logging to a file.

Editing the WebbPSF config file to set `autoconfigure_logging = True` (and any of the logging settings you wish to persist) instructs WebbPSF to apply your settings on import. (This is not done by default in case you have configured `logging` yourself and don't wish to overwrite your configuration.)

For more advanced log handling, see the Python logging module's own documentation.

level
[str] Name of log output to show. Defaults to 'INFO', set to 'DEBUG' for more extensive messages, or to 'WARN' or 'ERROR' for fewer.

filename
[str, optional] Filename to write the log output to. If not set, output will just be displayed on screen. (Default: None)

```
>>> webbpsf.setup_logging(filename='webbpsflog.txt')
```

This will save all log messages to 'webbpsflog.txt' in the current directory. If you later start another copy of webbpsf in a different directory, that session will also write to 'webbpsflog.txt' in *that* directory. Alternatively you can specify a fully qualified absolute path to save all your logs to one specific file.

```
>>> webbpsf.setup_logging(level='WARN')
```

This will show only WARNING or ERROR messages on screen, and not save any logs to files at all (since the filename argument is None)

show_notebook_interface

`webbpsf.`**`show_notebook_interface`**`(`*instrumentname*`)`
Show Jupyter notebook widget interface

instrumentname
[string] one of 'NIRCam','NIRSpec','NIRISS','FGS','MIRI' or 'WFI'

system_diagnostic

`webbpsf.`**`system_diagnostic`**`()`
return various helpful/informative information about the current system. For instance versions of python & available packages.

Mostly undocumented function…

test

`webbpsf.`**`test`**`(`*package=None*, *test_path=None*, *args=None*, *plugins=None*, *verbose=False*, *pastebin=None*, *remote_data=False*, *pep8=False*, *pdb=False*, *coverage=False*, *open_files=False*, ***kwargs*`)`
Run the tests using py.test. A proper set of arguments is constructed and passed to pytest.main.

package
[str, optional] The name of a specific package to test, e.g. 'io.fits' or 'utils'. If nothing is specified all default tests are run.

test_path
[str, optional] Specify location to test by path. May be a single file or directory. Must be specified absolutely or relative to the calling directory.

args
[str, optional] Additional arguments to be passed to pytest.main in the `args` keyword argument.

plugins
[list, optional] Plugins to be passed to pytest.main in the `plugins` keyword argument.

verbose
: [bool, optional] Convenience option to turn on verbose output from py.test. Passing True is the same as specifying `'-v'` in `args`.

pastebin
: [{'failed','all',None}, optional] Convenience option for turning on py.test pastebin output. Set to `'failed'` to upload info for failed tests, or `'all'` to upload info for all tests.

remote_data
: [bool, optional] Controls whether to run tests marked with @remote_data. These tests use online data and are not run by default. Set to True to run these tests.

pep8
: [bool, optional] Turn on PEP8 checking via the pytest-pep8 plugin and disable normal tests. Same as specifying `'--pep8 -k pep8'` in `args`.

pdb
: [bool, optional] Turn on PDB post-mortem analysis for failing tests. Same as specifying `'--pdb'` in `args`.

coverage
: [bool, optional] Generate a test coverage report. The result will be placed in the directory htmlcov.

open_files
: [bool, optional] Fail when any tests leave files open. Off by default, because this adds extra run time to the test suite. Works only on platforms with a working `lsof` command.

parallel
: [int, optional] When provided, run the tests in parallel on the specified number of CPUs. If parallel is negative, it will use the all the cores on the machine. Requires the pytest-xdist plugin installed. Only available when using Astropy 0.3 or later.

kwargs
: Any additional keywords passed into this function will be passed on to the astropy test runner. This allows use of test-related functionality implemented in later versions of astropy without explicitly updating the package template.

tkgui

`webbpsf.`**`tkgui`**(*fignum=1*)

10.1.3 Classes

`CGI`([mode, pixelscale, fov_arcsec, ...])	WFIRST Coronagraph Instrument
`Conf`	Configuration parameters for `webbpsf`.
`FGS`()	A class modeling the optics of the FGS.
`JWInstrument`(*args, **kwargs)	Superclass for all JWST instruments
`MIRI`()	A class modeling the optics of MIRI, the Mid-InfraRed Instrument.
`NIRCam`()	A class modeling the optics of NIRCam.
`NIRISS`([auto_pupil])	A class modeling the optics of the Near-IR Imager and Slit Spectrograph
`NIRSpec`()	A class modeling the optics of NIRSpec, in **imaging** mode.

Continued on next page

Table 2 – continued from previous page

UnsupportedPythonError	
WFI([set_pupil_mask_on])	WFI represents to the to-be-named wide field imager for the WFIRST mission

CGI

class webbpsf.**CGI**(*mode=None*, *pixelscale=None*, *fov_arcsec=None*, *apply_static_opd=False*)

Bases: webbpsf.wfirst.WFIRSTInstrument

WFIRST Coronagraph Instrument

Simulates the PSF of the WFIRST coronagraph.

Current functionality is limited to the Shaped Pupil Coronagraph (SPC) observing modes, and these modes are only simulated with static, unaberrated wavefronts, without relay optics and without DM control. The design respresented here is an approximation to a baseline concept, and will be subject to change based on trades studies and technology development.

mode
[str] CGI observing mode. If not specified, the __init__ function will set this to a default mode 'CHAR-SPC_F660'

pixelscale
[float] Detector pixelscale. If not specified, the pixelscale will default to 0.02 arcsec for configurations usint the IMAGER camera and 0.025 arcsec for the IFS.

fov_arcsec
[float] Field of view in arcseconds. If not specified, the field of view will default to 3.20 arcsec for the IMAGER camera and 1.76 arcsec for the IFS.

Attributes Summary

apodizer	Currently selected apodizer name
apodizer_list	
camera	Currently selected camera name
camera_list	
detector	Detector selected for simulated PSF
detector_position	The pixel position in (X, Y) on the detector
filter	Currently selected filter name
filter_list	
fpm	Currently selected FPM name
fpm_list	
lyotstop	Currently selected Lyot stop name
lyotstop_list	
mode	Currently selected mode name
mode_list	Available Observation Modes

Methods Summary

print_mode_table()	Print the table of observing mode options and their associated optical configuration

Attributes Documentation

`apodizer`
Currently selected apodizer name

`apodizer_list = ['CHARSPC', 'DISKSPC']`

`camera`
Currently selected camera name

`camera_list = ['IMAGER', 'IFS']`

`detector`
Detector selected for simulated PSF

Used in calculation of field-dependent aberrations. Must be selected from detectors in the `detector_list` attribute.

`detector_position`
The pixel position in (X, Y) on the detector

`filter`
Currently selected filter name

`filter_list = ['F660', 'F721', 'F770', 'F890']`

`fpm`
Currently selected FPM name

`fpm_list = ['CHARSPC_F660_BOWTIE', 'CHARSPC_F770_BOWTIE', 'CHARSPC_F890_BOWTIE', 'DISKSPC_F721_ANNULUS'`

`lyotstop`
Currently selected Lyot stop name

`lyotstop_list = ['LS30D88']`

`mode`
Currently selected mode name

`mode_list`
Available Observation Modes

Methods Documentation

`print_mode_table``()`
Print the table of observing mode options and their associated optical configuration

Conf

`class` `webbpsf.`**`Conf`**
Bases: `astropy.config.configuration.ConfigNamespace`

Configuration parameters for `webbpsf`.

Attributes Summary

`WEBBPSF_PATH`	Directory path to data files required for WebbPSF calculations, such as OPDs and filter transmissions.
`autoconfigure_logging`	Should WebbPSF configure logging for itself and POPPY? This adds handlers that report calculation progress and information
`default_fov_arcsec`	Default field of view size, in arcseconds per side of the square
`default_output_mode`	Should output include the oversampled PSF, a copy rebinned onto the integer detector spacing, or both? Options: 'oversampled','detector','both'
`default_oversampling`	Default oversampling factor: number of times more finely sampled than an integer pixel for the grid spacing in the PSF calculation.
`logging_filename`	Desired filename to save log messages to.
`logging_format_file`	Format for lines logged to a file.
`logging_format_screen`	Format for lines logged to the screen.
`logging_level`	Desired logging level for WebbPSF optical calculations.

Attributes Documentation

`WEBBPSF_PATH`
Directory path to data files required for WebbPSF calculations, such as OPDs and filter transmissions. This will be overridden by the environment variable $WEBBPSF_PATH, if present.

`autoconfigure_logging`
Should WebbPSF configure logging for itself and POPPY? This adds handlers that report calculation progress and information

`default_fov_arcsec`
Default field of view size, in arcseconds per side of the square

`default_output_mode`
Should output include the oversampled PSF, a copy rebinned onto the integer detector spacing, or both? Options: 'oversampled','detector','both'

`default_oversampling`
Default oversampling factor: number of times more finely sampled than an integer pixel for the grid spacing in the PSF calculation.

`logging_filename`
Desired filename to save log messages to.

`logging_format_file`
Format for lines logged to a file.

`logging_format_screen`
Format for lines logged to the screen.

`logging_level`
Desired logging level for WebbPSF optical calculations.

FGS

class webbpsf.**FGS**

Bases: `webbpsf.webbpsf_core.JWInstrument`

A class modeling the optics of the FGS.

Not a lot to see here, folks: There are no selectable options, just a great big detector-wide bandpass and two detectors.

JWInstrument

class webbpsf.**JWInstrument**(**args*, ***kwargs*)

Bases: `webbpsf.webbpsf_core.SpaceTelescopeInstrument`

Superclass for all JWST instruments

telescope : name of telescope pupilopd : filename or FITS file object

include_si_wfe

[boolean (default: True)] Should SI internal WFE be included in models? Requires the presence of `si_zernikes_isim_cv3.fits` in the `WEBBPSF_PATH`.

Attributes Summary

`detector`	Detector selected for simulated PSF
`pupilopd`	Filename *or* fits.HDUList for JWST pupil OPD.
`telescope`	

Methods Summary

`calc_psf`([outfile, source, nlambda, . . .])	Compute a PSF.
`interpolate_was_opd`(array, newdim)	Interpolates an input 2D array to any given size.
`load_was_opd`(inputWasOpd[, size, save, filename])	Load and interpolate an OPD from the WAS.
`psf_grid`([num_psfs, all_detectors, save, . . .])	Create a PSF library in the form of a grid of PSFs across the detector based on the specified instrument, filter, and detector.
`set_position_from_aperture_name`(aperture_name)	Set the simulated center point of the array based on a named SIAF aperture.

Attributes Documentation

detector

Detector selected for simulated PSF

Used in calculation of field-dependent aberrations. Must be selected from detectors in the `detector_list` attribute.

pupilopd = None

Filename *or* fits.HDUList for JWST pupil OPD.

This can be either a full absolute filename, or a relative name in which case it is assumed to be within the instrument's `data/OPDs/` directory, or an actual fits.HDUList object corresponding to such a file. 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.

`telescope = 'JWST'`

Methods Documentation

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'*, *add_distortion=True*, *crop_psf=True*)

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.

add_distortion
[bool] If True, will add 2 new extensions to the PSF HDUlist object. The 2nd extension will be a distorted version of the over-sampled PSF and the 3rd extension will be a distorted version of the detector-sampled PSF.

crop_psf
[bool] If True, when the PSF is rotated to match the detector's rotation in the focal plane, the PSF will be cropped so the shape of the distorted PSF will match it's undistorted counterpart. This will only be used for NIRCam, NIRISS, and FGS PSFs.

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.

interpolate_was_opd(*array*, *newdim*)
Interpolates an input 2D array to any given size.

array: float
input array to interpolate

newdim: int
new size of the 2D square array (newdim x newdim)

newopd: new array interpolated to (newdim x newdim)

load_was_opd(*inputWasOpd*, *size=1024*, *save=False*, *filename='new_was_opd.fits'*)
Load and interpolate an OPD from the WAS.

Ingests a WAS OPD and interpolates it to the proper size for WebbPSF.

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

size: int, optional
Desired size of the output OPD. Default is 1024.

save: bool, optional
Save the interpolated OPD if True. Default is False.

filename
[string, optional] Filename of the output OPD, if 'save' is True. Default is 'new_was_opd.fits'.

HDUlist
[string] fits.HDUList object of the interpolated OPD

psf_grid(*num_psfs=16*, *all_detectors=True*, *save=False*, *outfile=None*, *overwrite=True*, *verbose=True*, *use_detsampled_psf=False*, *single_psf_centered=True*, ***kwargs*)
Create a PSF library in the form of a grid of PSFs across the detector based on the specified instrument, filter, and detector. The output GriddedPSFModel object will contain a 3D array with axes [i, y, x] where i is the PSF position on the detector grid and (y,x) is the 2D PSF.

num_psfs
[int] The total number of fiducial PSFs to be created and saved in the files. This number must be a square number. Default is 16. E.g. num_psfs = 16 will create a 4x4 grid of fiducial PSFs.

all_detectors
: [bool] If True, run all detectors for the instrument. If False, run for the detector set in the instance. Default is True

save
: [bool] True/False boolean if you want to save your file. Default is False.

outfile
: [str] If "save" keyword is set to True, your current file will be saved under "{outfile}_det_filt.fits". Default of None will save it in the current directory as: instr_det_filt_fovp#_samp#_npsf#.fits

overwrite
: [bool] True/False boolean to overwrite the output file if it already exists. Default is True.

verbose
: [bool] True/False boolean to print status updates. Default is True.

use_detsampled_psf
: [bool] If True, the grid of PSFs returned will be detector sampled (made by binning down the oversampled PSF). If False, the PSFs will be oversampled by the factor defined by the oversample/detector_oversample/fft_oversample keywords. Default is False. This is rarely needed - if uncertain, leave this alone.

single_psf_centered
: [bool] If num_psfs is set to 1, this defines where that psf is located. If True it will be the center of the detector, if False it will be the location defined in the WebbPSF attribute detector_position (reminder - detector_position is (x,y)). Default is True This is also rarely needed.

****kwargs**
: Any extra arguments to pass the WebbPSF calc_psf() method call.

gridmodel
: [photutils GriddedPSFModel object] Returns a GriddedPSFModel object or a list of objects if more than one configuration is specified (1 per instrument, detector, and filter) User also has the option to save the grid as a fits.HDUlist object.

nir = webbpsf.NIRCam() nir.filter = "F090W" grid = nir.psf_grid(all_detectors=True, num_psfs=4)

nir = webbpsf.NIRCam() nir.filter = "F090W" nir.detector = "NRCA2" grid = nir.psf_grid(all_detectors=False, oversample=5, fov_pixels=101)

`set_position_from_aperture_name`(*aperture_name*)
: Set the simulated center point of the array based on a named SIAF aperture. This will adjust the detector and detector position attributes.

MIRI

class `webbpsf.`**`MIRI`**

Bases: `webbpsf.webbpsf_core.JWInstrument`

A class modeling the optics of MIRI, the Mid-InfraRed Instrument.

Relevant attributes include `filter`, `image_mask`, and `pupil_mask`.

The pupil will auto-select appropriate values for the coronagraphic filters if the auto_pupil attribute is set True (which is the default).

Attributes Summary

`filter`	Currently selected filter name (e.g.

Attributes Documentation

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

NIRCam

class `webbpsf.`**`NIRCam`**

Bases: `webbpsf.webbpsf_core.JWInstrument`

A class modeling the optics of NIRCam.

Relevant attributes include `filter`, `image_mask`, and `pupil_mask`.

The NIRCam class is smart enough to automatically select the appropriate pixel scale for the short or long wavelength channel based on the selected detector (NRCA1 vs NRCA5, etc), and also on whether you request a short or long wavelength filter. The auto-selection based on filter name can be disabled, if necessary, by setting `auto_channel = False`. Setting the detector name always toggles the channel regardless of `auto_channel`.

Note, if you use the `monochromatic` option for calculating PSFs, that does not invoke the automatic channel selection. Make sure to set the correct channel *prior* to calculating any monochromatic PSFs.

Special Options: The 'bar_offset' option allows specification of an offset position along one of the coronagraph bar occulters, in arcseconds. `` ` nc.image_mask = 'MASKLWB' nc.options['bar_offset'] = 3 # 3 arcseconds towards the right (narrow end on module A) ` ``

The 'nd_squares' option allows toggling on and off the ND squares for TA in the simulation. Note that these of course aren't removable in the real instrument; this option exists solely for some simulation purposes.

Attributes Summary

`LONG_WAVELENGTH_MAX`	
`LONG_WAVELENGTH_MIN`	
`SHORT_WAVELENGTH_MAX`	
`SHORT_WAVELENGTH_MIN`	
`channel`	
`detector`	Detector selected for simulated PSF
`filter`	Currently selected filter name (e.g.
`module`	

Attributes Documentation

`LONG_WAVELENGTH_MAX = 5.299999999999999e-06`

`LONG_WAVELENGTH_MIN = 2.35e-06`

`SHORT_WAVELENGTH_MAX = 2.35e-06`

`SHORT_WAVELENGTH_MIN = 6e-07`

`channel`

`detector`

Detector selected for simulated PSF

Used in calculation of field-dependent aberrations. Must be selected from detectors in the `detector_list` attribute.

`filter`

Currently selected filter name (e.g. F200W)

`module`

NIRISS

`class` `webbpsf.`**`NIRISS`**(*auto_pupil=True*)

Bases: `webbpsf.webbpsf_core.JWInstrument`

A class modeling the optics of the Near-IR Imager and Slit Spectrograph
(formerly TFI)

Relevant attributes include `image_mask`, and `pupil_mask`.

Imaging:

WebbPSF models the direct imaging and nonredundant aperture masking modes of NIRISS in the usual manner.

Note that long wavelength filters (>2.5 microns) have a pupil which includes the pupil alignment reference. If auto_pupil is set, the pupil will be toggled between CLEAR and CLEARP automatically depending on filter.

Spectroscopy:

Added in version 0.3 is partial support for the single-object slitless spectroscopy ("SOSS") mode using the GR700XD cross-dispersed grating. Currently this includes the clipping of the pupil due to the undersized grating and its mounting hardware, and the cylindrical lens that partially defocuses the light in one direction.

Warning: Prototype implementation - Not yet fully tested or verified.

Note that WebbPSF does not model the spectral dispersion in any of NIRISS' slitless spectroscopy modes. For wide-field slitless spectroscopy, this can best be simulated by using webbpsf output PSFs as input to the aXe spectroscopy code. Contact Van Dixon at STScI for further information. For SOSS mode, contact Loic Albert at Universite de Montreal.

The other two slitless spectroscopy grisms use the regular pupil and do not require any special support in WebbPSF.

Attributes Summary

`LONG_WAVELENGTH_MAX`
`LONG_WAVELENGTH_MIN`
`SHORT_WAVELENGTH_MAX`

Continued on next page

Table 10 – continued from previous page

SHORT_WAVELENGTH_MIN	
filter	Currently selected filter name (e.g.

Attributes Documentation

LONG_WAVELENGTH_MAX = 5.299999999999999e-06

LONG_WAVELENGTH_MIN = 2.35e-06

SHORT_WAVELENGTH_MAX = 2.35e-06

SHORT_WAVELENGTH_MIN = 6e-07

filter

Currently selected filter name (e.g. F200W)

NIRSpec

class webbpsf.**NIRSpec**

Bases: `webbpsf.webbpsf_core.JWInstrument`

A class modeling the optics of NIRSpec, in **imaging** mode.

This is not a substitute for a spectrograph model, but rather a way of simulating a PSF as it would appear with NIRSpec in imaging mode (e.g. for target acquisition). NIRSpec support is relatively simplistic compared to the other instruments at this point.

Relevant attributes include `filter`. In addition to the actual filters, you may select ‘IFU’ to indicate use of the NIRSpec IFU, in which case use the `monochromatic` attribute to set the simulated wavelength.

If a grating is selected in the pupil, then a rectangular pupil mask 8.41x7.91 m as projected onto the primary is added to the optical system. This is an estimate of the pupil stop imposed by the outer edge of the grating clear aperture, estimated based on optical modeling by Erin Elliot and Marshall Perrin.

Note: IFU to be implemented later

UnsupportedPythonError

exception webbpsf.**UnsupportedPythonError**

WFI

class webbpsf.**WFI**(*set_pupil_mask_on=None*)

Bases: `webbpsf.wfirst.WFIRSTInstrument`

WFI represents to the to-be-named wide field imager for the WFIRST mission

WARNING: This model has not yet been validated against other PSF

simulations, and uses several approximations (e.g. for mirror polishing errors, which are taken from HST).

Initiate WFI

set_pupil_mask_on
[bool or None] Set to True or False to force using or not using the cold pupil mask, or to None for the automatic behavior.

Attributes Summary

`MASKED_PUPIL_WAVELENGTH_MAX`	
`MASKED_PUPIL_WAVELENGTH_MIN`	
`UNMASKED_PUPIL_WAVELENGTH_MAX`	
`UNMASKED_PUPIL_WAVELENGTH_MIN`	
`pupil_mask`	Currently selected Lyot pupil mask, or None for direct imaging

Attributes Documentation

`MASKED_PUPIL_WAVELENGTH_MAX = 2e-06`

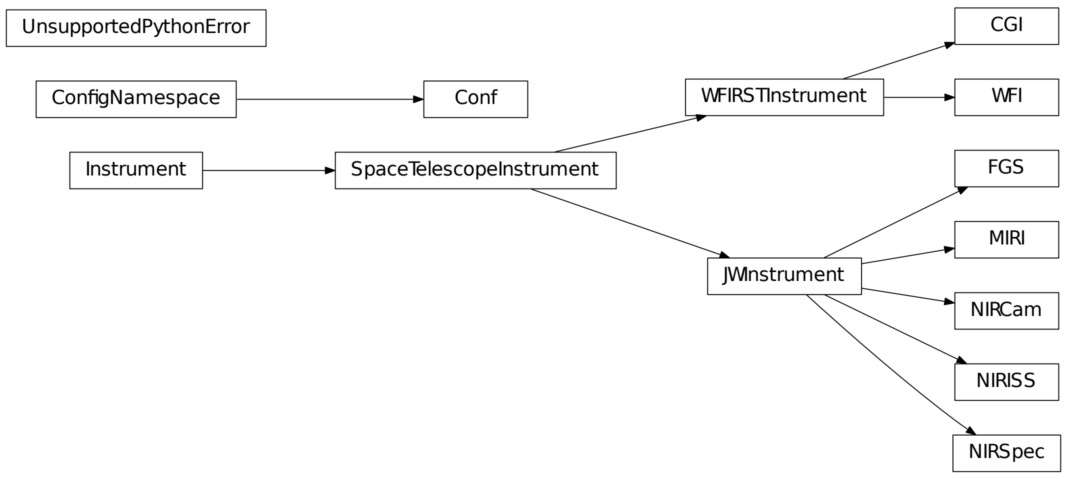
`MASKED_PUPIL_WAVELENGTH_MIN = 1.38e-06`

`UNMASKED_PUPIL_WAVELENGTH_MAX = 1.454e-06`

`UNMASKED_PUPIL_WAVELENGTH_MIN = 7.6e-07`

`pupil_mask`
Currently selected Lyot pupil mask, or None for direct imaging

10.1.4 Class Inheritance Diagram



CHAPTER 11

Diagnostics & Troubleshooting

If something does not work right, the first place to look is the *Known Issues* section of the release notes. The next place to check is the GitHub issues page, where another user may have reported the problem.

To report a new issue, you will need a free GitHub account. Alternatively, you may report the issue via email to the project maintainers. Include code that exhibits the issue to facilitate debugging.

WebbPSF includes a helper function that will return a report with information that may be useful for troubleshooting. An example of its usage is given below:

```
In [1]: import webbpsf
WebbPSF log messages of level INFO and above will be shown.
WebbPSF log outputs will be directed to the screen.

In [2]: print webbpsf.system_diagnostic()

OS: Darwin-13.4.0-x86_64-i386-64bit
Python version: 2.7.8 (default, Oct  2 2014, 13:50:25)  [GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.
↪0.51)]
numpy version: 1.9.1
poppy version: 0.3.3.dev335
webbpsf version: 0.3rc4

tkinter version: 0.3.1
wxpython version: not found

astropy version: 0.4.2
pysynphot version: 0.9.6
pyFFTW version: 0.9.2

Floating point type information for numpy.float:
Machine parameters for float64
---------------------------------------------------------------------
precision= 15   resolution= 1.0000000000000001e-15
machep=   -52   eps=        2.2204460492503131e-16
negep =   -53   epsneg=     1.1102230246251565e-16
```

(continues on next page)

(continued from previous page)

```
minexp= -1022   tiny=       2.2250738585072014e-308
maxexp=  1024   max=        1.7976931348623157e+308
nexp   =    11   min=        -max
---------------------------------------------------------------------

Floating point type information for numpy.complex:
Machine parameters for float64
---------------------------------------------------------------------
precision= 15   resolution= 1.0000000000000001e-15
machep=   -52   eps=        2.2204460492503131e-16
negep =   -53   epsneg=     1.1102230246251565e-16
minexp= -1022   tiny=       2.2250738585072014e-308
maxexp=  1024   max=        1.7976931348623157e+308
nexp   =    11   min=        -max
---------------------------------------------------------------------
```

CHAPTER 12

Sampling Requirements for Numerical Accuracy

The purpose of this appendix is to help you decide how many wavelengths and how much oversampling is required for your particular science application.

12.1 Key Concepts

Obtaining high accuracy and precision in PSF calculations requires treating both the multiwavelength nature of the selected bandpass and also the details of subpixel sampling and integration onto the detector pixels.

Note: The current version of this code makes no attempt to incorporate detector effects such as pixel MTF and interpixel capacitance. If you care about such effects, you should add them with another code.

Multiwavelength effects scale the PSF linearly with respect to wavelength. Thus the absolute scale of this effect increases linearly with distance from the PSF center. The larger a field of view you care about, the more wavelengths you will need to include.

Pixel sampling matters most near the core of the PSF, where the flux is changing very rapidly on small spatial scales. The closer to the core of the PSF you care about fine structure, the more finely sampled your PSF will need to be.

12.2 Some Useful Guidance

We consider two types of measurement one might wish to make on the PSF:

1. measuring the encircled energy curve to a given precision
2. measuring individual pixel flux levels to a given precision

The latter is substantially more challenging a measurement. The below tables present the number of (oversamplings, wavelengths) needed to achieve SNR=100 in a single pixel at a given radius (where SNR in this context is calculated as `(image-truth)/truth` on a per-detector-pixel basis and then averaged in an annulus as a function of radius). This calculation is motivated by modeling coronagraphic PSF subtraction, where we might hope to achieve 1-2 orders of magnitude reduction in the PSF wings through PSF subtraction. Accurately simulating that process demands a comparable level of fidelity in our PSF models. We also present tables giving the requirements for SNR=20 in a given pixel for less demanding modeling tasks.

Note that we do not consider here the case of trying to model the PSF core at SNR=100/pixel. If you are interested in doing so, I believe very fine subsampling would be needed. This might be most efficiently computed using a highly oversampled PSF for just the core, glued in to a larger image computed at lower angular resolution for the rest of the field of view. Investigating this is left as an exercise for another day.

Because NIRSpec, NIRISS, and FGS sample the PSF relatively coarsely, they will require a higher degree of oversampling in simulations than NIRCam to reach a given SNR level. MIRI is fairly well-sampled.

12.3 Per-Instrument Sampling Requirements

To evaluate what levels of sampling are needed in practice, for each NIRCam and MIRI filter we first computed a very highly oversampled image (nlambda=200, oversampling=16; field of view 5 arcsec for NIRCam and 12 arcsec for MIRI), which we used as a "truth" image. (For practical purposes, we consider this level of sampling likely to be sufficiently fine that it's a good stand-in for an infinitely sampled PSF, but this is an assumption we have not quantitatively validated. However, since there are >200 subsamples in both pixel and wavelength space, the residuals ought to be <1/200 and thus these are sufficient for our purposes of testing SNR=100.)

These tables list the (oversampling, wavelengths) required to achive the specified SNR, in comparison with a 'truth' image based on simulations using `oversampling = 16` (i.e. 256 subpixels per detector pixel) and `nlambda=200`.

Required sampling for NIRCam:

```
NIRCam, SNR=100
             r=0.5"      1.0"       2.0"       3.0"
     F070W    higher!    (4, 13)    (4, 21)    (4, 30)
     F090W    higher!    (4, 13)    (4, 21)    (4, 30)
     F115W    higher!     (4, 9)    (4, 21)    (4, 30)
     F140M    higher!     (4, 9)     (4, 9)    (4, 13)
    F150W2    higher!    (4, 30)    (2, 75)    (2, 75)
     F150W    higher!     (4, 9)    (4, 21)    (4, 21)
     F162M    higher!     (4, 9)     (4, 9)    (4, 13)
     F164N    higher!     (8, 3)     (8, 3)     (8, 3)
     F182M    higher!     (4, 9)     (4, 9)    (4, 13)
     F187N    higher!     (8, 1)     (4, 5)     (8, 3)
     F200W     (8, 5)     (4, 9)    (2, 21)    (2, 30)
     F210M     (8, 3)     (4, 5)     (4, 9)     (4, 9)
     F212N     (8, 1)     (4, 3)     (4, 3)    (4, 13)
     F225N     (8, 1)     (4, 3)     (4, 3)     (4, 5)
     F250M    higher!     (8, 5)    (4, 13)     (4, 9)
     F277W     (8, 5)     (4, 9)    (4, 13)    (4, 21)
     F300M     (8, 3)     (8, 5)     (4, 9)     (4, 9)
    F322W2     (8, 9)    (4, 21)    (4, 21)    (4, 30)
     F323N     (8, 1)     (8, 1)     (8, 3)     (8, 3)
     F335M     (8, 3)     (8, 5)     (4, 9)     (4, 9)
     F356W     (8, 5)     (4, 9)     (4, 9)    (4, 13)
     F360M     (8, 3)     (8, 5)     (4, 5)     (4, 9)
     F405N     (8, 1)     (8, 1)     (4, 9)     (8, 3)
     F410M     (8, 3)     (8, 5)     (4, 5)     (4, 9)
     F418N     (8, 1)     (8, 1)     (4, 5)     (8, 3)
     F430M     (8, 1)     (8, 3)     (4, 9)     (4, 9)
     F444W     (8, 5)     (4, 9)    (4, 13)    (2, 21)
     F460M     (8, 3)     (8, 5)     (4, 9)     (4, 9)
     F466N     (8, 1)     (8, 1)     (4, 3)     (4, 9)
```

(continues on next page)

(continued from previous page)

```
     F470N     (8, 1)     (8, 1)     (4, 3)     (4, 3)
     F480M     (8, 3)    (4, 21)     (4, 5)     (4, 9)

NIRCam, SNR=20
              r=0.5"       1.0"       2.0"       3.0"
     F070W     (8, 3)     (2, 9)    (2, 21)    (2, 21)
     F090W     (8, 3)     (2, 9)    (2, 13)    (2, 21)
     F115W     (8, 3)     (2, 9)    (2, 13)    (2, 21)
     F140M     (8, 3)     (2, 5)     (2, 5)     (2, 9)
    F150W2     (8, 9)    (2, 21)    (1, 50)    (1, 75)
     F150W     (8, 3)     (2, 9)    (2, 13)    (2, 21)
     F162M     (8, 3)     (2, 5)     (2, 5)     (2, 9)
     F164N     (8, 1)     (4, 1)     (2, 3)     (4, 3)
     F182M     (8, 3)     (2, 3)     (2, 5)     (2, 9)
     F187N     (8, 1)     (4, 1)     (2, 3)     (2, 5)
     F200W     (4, 3)     (2, 5)    (1, 13)    (1, 21)
     F210M     (4, 3)     (2, 3)     (2, 5)     (2, 5)
     F212N     (4, 1)     (2, 1)     (2, 3)     (2, 3)
     F225N     (4, 1)     (2, 1)     (2, 3)     (2, 3)
     F250M     (8, 1)     (4, 3)     (2, 5)     (2, 5)
     F277W     (4, 3)     (2, 5)     (2, 9)    (2, 13)
     F300M     (4, 3)     (4, 3)     (2, 5)     (2, 5)
    F322W2     (4, 5)     (2, 9)    (2, 21)    (2, 21)
     F323N     (4, 1)     (4, 1)     (4, 1)     (4, 1)
     F335M     (4, 3)     (4, 3)     (2, 5)     (2, 5)
     F356W     (4, 3)     (2, 5)     (2, 9)     (2, 9)
     F360M     (4, 3)     (4, 3)     (2, 3)     (2, 5)
     F405N     (4, 1)     (4, 1)     (2, 3)     (2, 3)
     F410M     (4, 1)     (4, 3)     (2, 3)     (2, 5)
     F418N     (4, 1)     (4, 1)     (2, 1)     (4, 1)
     F430M     (4, 1)     (4, 3)     (2, 3)     (2, 5)
     F444W     (4, 3)     (2, 5)     (1, 9)    (1, 13)
     F460M     (4, 1)     (2, 5)     (2, 3)     (2, 5)
     F466N     (4, 1)     (4, 1)     (2, 1)     (2, 3)
     F470N     (4, 1)     (2, 3)     (2, 1)     (2, 1)
     F480M     (4, 1)     (2, 5)     (2, 3)     (2, 3)
```

We have not yet performed simulations for the case of NIRISS. The number of wavelengths used for each filter is set equal to that used for NIRCam. This should certainly be adequate for the long-wavelength filters (given the NIRISS detector and NIRCam LW are identical) but users may wish to investigate using finer sampling for the shorter wavelength filters that are very undersampled on NIRISS.

And for MIRI:

```
MIRI, SNR = 100
                  r=1.0"      2.0"       3.5"       5.0"
         F560W    (4, 5)     (4, 9)    (4, 13)    (4, 13)
         F770W    (4, 5)     (2, 9)    (2, 13)    (2, 21)
        F1000W    (4, 3)     (4, 5)     (2, 9)     (2, 9)
        F1065C    (4, 3)     (4, 5)     (4, 5)     (2, 5)
        F1130W    (4, 3)     (4, 5)     (2, 5)     (2, 5)
        F1140C    (4, 3)     (4, 3)     (4, 5)     (2, 5)
        F1280W    (4, 3)     (2, 5)     (2, 9)     (2, 9)
        F1500W    (4, 3)     (2, 5)     (2, 9)     (2, 9)
        F1550C    (4, 3)     (2, 3)     (2, 3)     (2, 5)
        F1800W    (2, 3)     (2, 3)     (2, 9)     (2, 9)
        F2100W    (2, 3)     (2, 5)     (2, 9)     (1, 9)
```

(continues on next page)

(continued from previous page)

```
        F2300C     (2, 3)     (2, 5)     (1, 9)     (1, 9)
        F2550W     (2, 3)     (1, 5)     (1, 9)     (1, 9)
           FND    (2, 30)    (2, 40)    (2, 50)    (2, 75)


MIRI, SNR=20
                   r=1.0"       2.0"       3.5"       5.0"
         F560W     (2, 3)     (2, 5)     (2, 9)     (2, 9)
         F770W     (2, 3)     (1, 9)     (1, 9)     (1, 9)
        F1000W     (2, 3)     (1, 5)     (1, 5)     (1, 5)
        F1065C     (2, 1)     (2, 3)     (2, 3)     (1, 3)
        F1130W     (2, 1)     (2, 3)     (1, 3)     (1, 3)
        F1140C     (2, 1)     (2, 3)     (1, 3)     (1, 3)
        F1280W     (2, 3)     (1, 3)     (1, 5)     (1, 5)
        F1500W     (2, 3)     (1, 3)     (1, 5)     (1, 5)
        F1550C     (2, 1)     (1, 3)     (1, 3)     (1, 3)
        F1800W     (1, 3)     (1, 3)     (1, 5)     (1, 5)
        F2100W     (1, 3)     (1, 3)     (1, 5)     (1, 5)
        F2300C     (1, 3)     (1, 3)     (1, 5)     (1, 5)
        F2550W     (1, 3)     (1, 3)     (1, 3)     (1, 5)
           FND    (1, 13)    (1, 21)    (1, 40)    (1, 50)
```

The defaults for MIRI are set to 9 wavelengths for all filters, except for F560W and F770W which use 13 and FND which uses 40.

More later.

CHAPTER 13

Optimizing FFT Performance for PSF Computations with FFTW

Optimizing numerical performance of FFTs is a very 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 creates 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.

13.1 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

13.2 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.

13.3 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
```

13.4 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
```

13.5 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
```

13.6 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

Appendices and Reference

CHAPTER 14

Release Notes

14.1 Known Issues

See https://github.com/spacetelescope/webbpsf/issues for currently open issues and enhancement suggestions.

- Calculations at large radii (> 500 lambda/D ~ 30 arcsec for 2 microns) will show numerical artifacts from Fourier aliasing and the implicit repetition of the pupil entrance aperture in the discrete Fourier transform. If you need accurate PSF information at such large radii, please contact Marshall Perrin or Marcio Melendez for higher resolution pupil data.

The following factors are NOT included in these simulations:

- Coronagraphic masks are assumed to be perfect (i.e. the masks exactly match their design parameters.)
- Most detector effects, such as intrapixel sensitivity variations or interpixel capacitance. There are currently no plans to include these WebbPSF itself. Generate a subsampled PSF and use a separate detector model code instead. The one exception is a scattering artifact in the MIRI imager detector substrate.

14.2 Road Map for Future Releases

- Continued validation and updates as needed based on further analyses of instrument and telescope hardware test data.
- Support for the NIRSpec and MIRI IFUs may be added in a future release; level of detail is still TBD.
- Improved models for telescope WFE evolution over time.
- Possibly: separate handling of pre- and post- coronagraphic WFE in instruments, or pre- and post- NIRSpec MSA plane WFE; pending receipt of test data and models from the instrument teams.

14.3 Version History and Change Log

14.3.1 Version 0.8.0

2018 Dec 15

This release focused on software engineering improvements, rather than changes in any of the optical models or reference data. (In particular, there are NO changes in the reference data files; the contents of the WebbPSF version 0.8 data zip file are identical to the reference data as distributed for version 0.7. This version of WebbPSF will work with either of those interchangably.).

Python version support: Python 3 required

This version drops support for Python 2.7. The minimum supported version of Python is now 3.5.

New functionality:

- *Added new capability to create grids of fiducial, distorted PSFs* spanning a chosen instrument/detector. This new `psf_grid` method is meant to be used as the first step of using the `photutils` package to do PSF-fitting photometry on simulated JWST PSFs. This method will output a list of or single `photutils` `GriddedPSFModel` object(s) which can then be read into `photutils` to apply interpolation to the grid and simulate a spatially dependent PSF anywhere on the instrument. See this Jupyter notebook for examples. This method requires `photutils` version 0.6 or higher. [`#241, _`, @shanosborne with inputs from @mperrin, @larrybradley, @hcferguson, and @eteq]

Bug fixes and small changes:

- *Improved the application of distortion to PSFs* to allow distorted PSFs to be created when the output mode is set to only "oversampled" or only "detector-sampled." When either of these modes is set in the options dictionary, the output will be an HDUList object with two extensions, where the 1st extension is the same PSF as in the 0th extension but with distortion applied. [`#229, _`, @shanosborne]
- Also fixed distorted PSFs which were shifted off-center compared to their undistorted counterparts. These distorted PSFs had always been created in the correct detector location, but the values in the array returned by `calc_psf` were shifted off from the center. This bug was particularly apparent when the PSFs were set with a location near the edge of the detector. [`#219, _`, @shanosborne]
- Fix FITS output from JWST OTE linear model, plus typo fixes and PEP8 improvements [#232, @laurenmarietta]
- Display code added for the PSF grid functionality mentioned above [#247, @mperrin]

Software and Package Infrastructure Updates:

- Removed Python 2.7 compatibility code, use of six and 2to3 packages, and Python 2 test cases on Travis (#236, #239, @mperrin, @kjbrooks]
- Packaging re-organized for consistency with current STScI package template (#240, @robelgeda)
- Documentation template updated for consistency with current STScI docs template (#250, @robelgeda)
- Documentation added or updated for a variety of features [#248, @mperrin]
- Various smaller code cleanup and doc improvements, including code cleanup for better Python PEP8 style guide compliance [#227, #255, @shanosborne]
- Updated to newer syntax for specifying pupil shifts of optical elements [#257, @mperrin]
- Unit tests added for defocused instruments, including the NIRCam weak lenses [#256, @mperrin]

- Updated astropy-helpers submodule to 3.0.2 [#249, @mperrin]
- Software development repo on Github shifted to within the spacetelescope organization.

14.3.2 Version 0.7.0

2018 May 30

Note, when upgrading to this version you will need to update to the latest data files as well. This is handled automatically if you use conda, otherwise you will need to download and install the data from: webbpsf-data-0.7.0.tar.gz.

Python version support: Future releases will require Python 3.

Please note, this is the *final* release of WebbPSF 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 of WebbPSF 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.

General:

- Improved numerical performance in calculations using new accelerated math functions in poppy. It is highly recommended that users install the `numexpr` package, which enables significant speed boosts in typical propagations. `numexpr` is easily installable via Anaconda. Some use cases, particularly for coronagraphy or slit spectroscopy, can also benefit from GPU acceleration. See the latest poppy release notes for more.

JWST optical model improvements:

- *Models of field-dependent wavefront error are now included for all the SIs.* The OPD information is derived from the ISIM CV3 test campaign at Goddard, as described extensively in David Aronstein et al. "Science Instrument Wavefront Error and Focus: Results Summary from the ISIM Cryogenic Vacuum Tests:", JWST-RPT-032131. (See also the SPIE paper version.) The measured SI wavefront errors are small, some tens of nanometers, and are in general less than the telescope WFE at given location. This information on SI WFE is provided to help inform modeling for what potential variations in PSFs across the field of view might look like, in broad trends. However it should _not_ be taken as precise guarantee of the exact amplitudes or functional form of those variations. The WFE was measured at a small handful of particular field points during CV3, and the resulting Zernike coefficients are interpolated to produce _estimated_ wavefront maps at all other field points across the focal planes. Density and precision of the available measurements vary substantially between instruments. [@mperrin, with contributions from @josephoenix in prior releases, and from @robelgeda and @JarronL for the interpolation between field points. [#121, #187]
- *Added new capabilities for modeling distortions of the image planes*, which cause slight deflections in the angles of diffractive features. The result of geometric distortion is that detector pixels are not ideal square sections of the sky; they're slightly skewed parallelograms. (See the ACS handbook for examples of what this looks like for Hubble PSFs) For the JWST instruments, this effect is largest for FGS, and fairly small but noticeable for the other SIs. See this Jupyter notebook for examples of the effect on JWST PSFs. Note that the distorted PSFs are added as *additional extensions* in the output FITS file, so you will need to read from extension 2 or 3 if you want the PSF with the distortion included; extensions 0 and 1 remain consistent with prior versions. The distortion information is taken from the Science Instrument Aperture file (SIAF) reference data maintained at STScI. As a result the `pysiaf` package is a new dependency required for using `webbpsf`. The distortion calculations can

add 1-3 seconds to each PSF calculation, and double the size of the output FITS files; if modeling distortion is not needed for your use case, you can deactivate this by setting `add_distortion=False` in calls to `calc_psf`. [#209, @shanosborne]

- *Added small nonzero pupil shears* for most instruments, based on measurements from the ISIM CV3 and OTIS cryo tests, adjusted for gravity release to produce predicted on-orbit pupil shears. See JWST-RPT-028027 and JWST-RPT-037134. For most imaging mode PSFs, this has _no_ practical effect because the SI internal pupils are oversized to provide tolerance, and the measured shears are well below that amount. It has a small but nonzero effect for long-wave NIRISS filters with the CLEARP pupil obscuration. The greatest effect is for MIRI coronagraphy since MIRI's Lyot stops were not undersized to allow for pupil shear, but even so the impact is small for the < 1% expected shift. Note that for NIRCam, the expected pupil shear is set to precisely zero, given the expectation that NIRCam's steerable pickoff mirror will be used in flight to achieve precise pupil alignment. [#212,, @shanosborne, with inputs from Melendez, Telfer, and Hartig]
- *For MIRI only*, added new capability for modeling blurring due to *scattering of light within the MIRI imager detector substrate itself*. This acts as a cross-shaped convolution kernel, strongest at the shortest wavelengths. See MIRI document MIRI-TN-00076-ATC for details on the relevant physics and detector calibration. This is implemented as part of the distortion framework, though it is different physics. See this Jupyter notebook for example output. For F560W through F1000W this is a much more obvious effect than the subtle distortions. [#209,, @shanosborne]
- *Added new capabilities for modeling mirror moves of the JWST primary segments and secondary mirror*, using a linear optical model to adjust OPDs. Added a new notebook demonstrating these capabilities. Note this code allows simulation of arbitrary mirror motions within a simplified linear range, and relies on user judgement what those mirror motions should be; it is not a detailed rigorous optomechanical model of the observatory. [Code by @mperrin, with some fixes by Geda in <#185]
- All the instrument+filter relative spectral response functions have been updated to values derived from the official validated JWST ETC reference data, using the Pandeia ETC release version 1.2.2. [@mperrin]

WFIRST optical model improvements:

- *The WFI optical model has been updated to use optical data from the Cycle 7 design revision for WFI*. This includes a change in the instrument field of view layout relative to the axes, as shown here. [#184, @robelgeda]
- Added R062 filter.
- Updated `pupil_mask` attribute for toggling between the masked and non-masked pupils now works the same way as that attribute does for the JWST instrument classes. Note, most users will not need to deal with this manually as the WFI class will by default automatically select the correct pupil based on the selected filter. [#203, @robelgeda]

Bug fixes and minor changes:

- All JWST instruments: Added new feature for importing OPD files produced with the JWST Wavefront Analysis System software [#208, @skyhawk172]
- All JWST instruments: Fix to generalize OPD loading code to handle either compressed or uncompressed OPDs [#173, @JarronL]
- All JWST instruments: Fix to properly load the default number of wavelengths per calculation from the filters.tsv file, rather than defaulting to 10 wavelengths regardless. [@shanosborne])
- All JWST instrument: Fix to more correctly handle non-integer-pixel positions of the PSF when writing DET_X and DET_Y header keywords (#205, @shanosborne]
- NIRCam and MIRI coronagraphy: Automatically set the detector coordinates and SI WFE maps based on the location of a selected coronagraph occulter. [#181, @mperrin]
- NIRCam coronagraphy: Fix a sign error in offsets for the NIRCam coronagraph SWB occulters [#172, @mperrin].

- NIRCam coronagraphy: Fix a half-percent throughput error in the round occulter masks [#206, @mperrin]
- NIRCam coronagraphy: Fix an issue with transmission of the coronagraph bars precisely along the y axis, due to a typo [#190, @JarronL]
- NIRCam coronagraphy: New option for shifting the coronagraph masks relative to the source, rather than vice versa. This is mostly of use for edge cases such as PSF library generation for the ETC, and is probably not of widespread utility. [#191, @mperrin]
- NIRISS: Fix the `pupil_rotation` option so it works for NIRISS too, in particular for NRM/AMI. [#118, @mperrin]
- NIRSpec: Very incomplete initial rudimentary support for the NIRSpec IFU, specifically just implementing the field stop for the IFU aperture. [@mperrin]
- Updated to newer version of the astropy_helpers package infrastructure [@sosey]
- Various smaller code cleanup and doc improvements, including code cleanup for better Python PEP8 style guide compliance [@mperrin, @shanosborne, @robelgeda, @douglase]
- The `utils.system_diagnostic` function now checks and reports on a few more things that might be useful in diagnosing performance issues.

14.3.3 Version 0.6.0

2017 August 11

JWST optical models:

- Substantial update to the optical models for the telescope, to incorporate measurements of the as-built optics plus the latest expectations for alignments in flight. The reference data layout has changed: each instrument now includes only two OPD files, a `predicted` and a `requirements` OPD. Ex: `OPD_RevW_ote_for_NIRCam_predicted.fits.gz`. The OPD files are now derived from measured flight mirror surfaces (for high spatial frequencies), plus statistical models for their alignment in flight following wavefront sensing and control (for mid and lower spatial frequencies), as described in *JWST Instrument Model Details*. Each OPD file still contains 10 different realizations of the statistical part.
- The NIRISS `auto_pupil` feature now recognizes that the `CLEAR` filter is used with the `GR700XD` pupil mask [#151]
- Correctly convert wavelengths to microns when computing NIRISS ZnS index of refraction [#149]
- Aperture definitions now come from a copy of the SIAF bundled in `jwxml` rather than in the WebbPSF reference data.
- An alpha version of a linear optical model for adjusting OPDs is now provided for power-users, but currently unsupported and not documented.

WFIRST optical models:

- Addition of a model for the WFIRST CGI (Coronagraph Instrument) shaped pupil coronagraph by @neilzim [#154]

General:

- Jitter is now enabled by default (approximated by convolution with 0.007 arcsec FWHM Gaussian)
- Source offsets can now be specified as `source_offset_x` and `source_offset_y` in `instrument.options` (in addition to the existing `instrument.options['source_offset_r']` and `instrument.options['source_offset_theta']`)

- The Astropy Helpers have been updated to v2.0.1 to fix various install-time issues.

14.3.4 Version 0.5.1

Released 2016 November 2. Bug fix release to solve some issues that manifested for AstroConda users.

- Fixed a few missed version number->0.5.0 edits in install docs
- Updated install instructions for Ureka->Astroconda change
- Clarified release instructions for data packages
- Fixed ConfigParser import in setup.py
- Documented PSF normalization options better. (#112)
- Updated Travis-CI config, consistent with poppy#187
- Made a display tweak for the primary V2V3 annotation
- Removed redundant `calcPSF` in favor of just using the superclass `calc_psf` (#132)
- Updated `measure_strehl` to turn off SI WFE for perfect PSF calcs
- Enforced Python 3.0+ compliance on code with `__future__` imports
- Used `six.string_types` for Python 3.x compliance
- Add version specs to dependencies in `setup.py`
- Made `jwxml` a dependency in `setup.py`

14.3.5 Version 0.5.0

Released 2016 June 10. Various updates to instrument properties, improved documentation, and overhaul of internals in preparation for measured WFE data on JWST SIs.

JWST updates:

- New documentation on *JWST Instrument Model Details*
- Updated all JWST SI pixel scales to latest measured values from ISIM CV3 and STScI Science Instruments Aperture File.
- Add coordinate inversion to get the correct (inverted) orientation of the OTE exit pupil relative to the ISIM focal plane. This will show up as an extra intermediate optical plane in all PSF calculations from this point, with the OTE pupil obscuration flipped upside down in orientation relative to the entrance pupil.
  - As a consequence of this, many optical planes displayed will now look "upside down" relative to prior versions of WebbPSF. This affects all coronagraphic Lyot masks for instance, the NIRISS CLEARP and NRM pupils, etc. This is as intended, and reflects the actual orientation of those optics in the internal pupil planes relative to a detector image that has been oriented to have +V3 up and +V2 left (e.g. 'SCI' frame orientation on the sky, with north up and east left if the position angle is zero).
- Added software infrastructure for using measured instrument WFE from ISIM cryo-tests - however the data files are not yet ready and approved. This functionality will be fully activated in a near-future release (later this summer).
- Added attributes for detector selection and pixel positions to all SIs, backed with latest science instrument aperture file mapping between detector pixels and angular positions on the JWST focal plane.
- Improved automatic toggling based on selected filter of instrument properties such as NIRCam short/long channel and pixel scales, and NIRISS and MIRI pupil masks.

- *Thanks to Kyle van Gorkom, Anand Sivaramakrishnan, John Stansberry, Colin Cox, Randal Telfer, and George Hartig for assisting with information and data to support these updates.*

WFIRST updates:

- Updated to GSFC Cycle 6 modeling results for WFI.
- Some behind-the-scenes refactoring to implementation details for field dependent WFE to support code sharing between the JWST and WFIRST classes.
- *Thanks to Alden Jurling for assisting with information and clarifications on the Cycle 6 models.*

General:

- New Python PEP8 style guide compliant names have been added for most function calls, e.g. `calc_psf` instead of `calcPSF`, `display_psf` instead of `display_PSF` and so forth. For now these are synonymous and both forms will work. The new styling is preferred and at some future point (but not soon!) the older syntax may be removed.

14.3.6 Version 0.4.1

Released 2016 April 04. Mostly minor bug fixes, plus some updates to better match orientations of output files.

- Fix an bug that ignored the rotation of the MIRI coronagraph occulters, introduced by changes in poppy 0.4.0; (#91; @kvangorkom, @josephoenix, @mperrin) and also flip the sign of that rotation from 4.5 degrees counter-clockwise to 4.5 clockwise, to match the actual hardware (#90; @kvangorkom, @josephoenix, @mperrin)
- Also flip orientations of some NIRCam coronagraphic masks and improve modeling of NIRCam coronagraph ND squares and occulter bar mounting hardware (#85; @mperrin); and remove two obsolete filter data files that don't correspond to any actual filters in NIRCam.
- Relocate `measure_strehl` function code into `webbpsf` (#88; Kathryn St.Laurent, @josephoenix, @mperrin)
- Other minor bug fixes and improved error catching (#87; @mperrin) (#95; @mperrin) (#98; @josephoenix) (#99; @mperrin)
- Better document how to make monochromatic PSFs (#92; @mperrin) and fix broken link in docs (#96; @josephoenix).

14.3.7 Version 0.4.0

Released 2015 November 20

- **WFIRST WFI support added**:
  - including all WFI filters and filter-dependent pupil masks.
  - including field dependence based on GSFC Cycle 5 modeling (#75, @josephoenix)
  - including initial/prototype GUI interface based on Jupyter/IPython notebook widgets (#79, @josephoenix)
- Updated filter transmission files for MIRI (based on Glasse et al. 2015 PASP) and NIRISS (based on flight filter measurement data provided by Loic Albert). (#66, #78; @mperrin)
- Added utility to check for appropriate version of the data files and request an update if necessary (#76, @josephoenix)
- Some documentation updates, including new documentation for the WFIRST functionality (@josephoenix, @mperrin)
- Bug fixes for minor issues involving OPD file units (#74, @josephoenix), cleaner logging output, and some Python 3 compatibility issues.

Note: When updating to version 0.4 you will need to also update your WebbPSF data files to the latest version as well.

14.3.8 Version 0.3.3

Released July 1, 2015

- **Python 3 compatibility added.** All tests pass on Python 3.4. (#2)
- Fixed an issue that would prevent users from adding defocus to PSF calculations
- WebbPSF no longer attempts to display a welcome message on new installs; that idea proved to be less helpful than originally expected.
- Added a `CLEAR` filter option for NIRISS, since the corresponding clear position is actually in the filter wheel rather than the pupil mask wheel. Rather than an actual filter, the profile for `CLEAR` is 1.0 between 0.6 microns and 5.0 microns per the stated limits of the detector, and 0.0 everywhere else. (#64)
- Multi-wavelength calculations across a filter were not choosing a sensible number of wavelengths from the tables included in `webbpsf-data`. (#68)

14.3.9 Version 0.3.2

Released February 23, 2015

This is a bug-fix release to address an issue that rendered the GUI unusable. (See #55.) API usage was unaffected.

(Ask not what happened to 0.3.1.)

14.3.10 Version 0.3.0

Released 2015 February

This is a major release of WebbPSF, with several additions to the optical models (particularly for slit and slitless spectroscopy), and extensive software improvements and under-the-hood infrastructure code updates. Many default settings can now be customized by a text configuration file in your home directory.

Updates to the optical models:

- Initial support for spectroscopy: *NIRSpec fixed slit and some MSA spectroscopy*, *MIRI LRS spectroscopy* (for both slit and slitless modes), and *NIRISS single-object slitless spectroscopy*. To model one of these modes, select the desired image plane stop (if any) plus the pupil plane stop for the grating. WebbPSF does not yet include any model for the spectral dispersion of the prisms, so you will want to perform monochromatic calculations for the desired wavelengths, and coadd the results together yourself into a spectrum appropriately. For example:

```
>> nirspec.image_mask = 'S200A1'
>> nirspec.pupil_mask = 'NIRSpec grating'
>> monopsf = nirspec.calcPSF(monochromatic=3e-6, fov_arcsec=3)

>> miri.image_mask = 'LRS slit'
>> miri.pupil_mask = 'LRS grating'
>> miripsf = miri.calcPSF(monochromatic=10e-6)

>> niriss.pupil_mask = 'GR700XD'
>> monopsf = niriss.calcPSF(monochromatic=1.5e-6, oversample=4)
```

In fact the NIRSpec class now automatically defaults to having the NIRSpec grating pupil stop as the selected pupil mask, since that's always in the beam. For MIRI you must explicitly select the 'LRS grating' pupil mask, and may select the 'LRS slit' image stop. For NIRISS you must select the 'GR700XD' grating as the pupil mask, though of course there is no slit for this one.

Please note This is new/experimental code and these models have not been validated in detail against instrument hardware performance yet. Use with appropriate caution, and we encourage users and members of the instrument teams to provide input on how this functionality can be further improved. Note also that MIRI MRS and NIRSpec IFU are still unsupported.

Thanks to Loic Albert (U de Montreal) and Anand Sivaramakrishnan for data and many useful discussions on NIRISS SOSS. Thanks to Klaus Pontoppidan for proposing the NIRSpec and MIRI support and useful discussions. Thanks to Erin Elliott for researching the NIRSpec grating wheel pupil stop geometry, and Charles Lajoie for information on the MIRI LRS pupil stop.

- Added NIRISS CLEARP pupil mask; this includes the obscuration from the pupil alignment reference. Given the pupil wheel layout, this unavoidably must be in the beam for any NIRISS long-wave PSFs, and WebbPSF will automatically configure it in the necessary cases. Thanks to Anand Sivaramakrishnan.
- Minor bug fix to weak lens code for NIRCam, which previously had an incorrect scaling factor. Weak lens defocus values updated to the as-built rather than ideal values (which differ by 3%, but the as built values are very well calibrated).
- Added defocus option to all instruments, which can be used to simulate either internal focus mechanism moves or telescope defocus during MIMF. For example, set

```
>> nircam.options['defocus_waves']=3
>> nircam.options['defocus_wavelength']=2.0e-6
```

  to simulate 3 waves of defocus at 2 microns, equivalently 6 microns phase delay peak-to-valley in the wavefront.
- Added new option to offset intermediate pupils (e.g. coronagraphic Lyot stops, spectrograph prisms/grisms, etc) in rotation as well as in centering:

```
>> niriss.options['pupil_rotation'] = 2  # degrees counterclockwise
```

- 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.calcPSF('F212N', fov_arcsec=[3,6])
>> nc.calcPSF('F187N', fov_pixels=(300,100) )
```

  Those two elements give the desired field size as (Y,X) following the usual Python axis order convention. This is motivated in particular by the rectangular subarrays used in some spectroscopic modes.

Other Software Updates & Enhancements:

- Required Python modules updated, now with dependency on astropy:
  - `astropy.io.fits` replaces `pyfits` for FITS I/O.
  - `astropy.io.ascii` replaces `asciitable` for ASCII table I/O.
  - `atpy` is no longer required.
  - New `astropy.config` configuration system is used for persistent settings. This includes saving accumulated FFTW 'wisdom' so that future FFT-based calculations will begin more rapidly.
  - `lxml` now required for XML parsing of certain config files

  - psutil strongly recommended for cross-platform detection of available free RAM to enable better parallelization.

- Improved packaging infrastructure. Thanks to Christine Slocum, Erik Bray, Mark Sienkiewicz, Michael Droettboom, and the developers of the Astropy affiliated package template. Thanks in particular to Christine Slocum for integration into the STScI SSB software distribution.
- Improvements to parallelization code. Better documentation for parallelization. PyFFTW3 replaced with pyFFTW for optimized FFTs (yes, those are two entirely different packages).
- Alternate GUI using the wxpython widget toolkit in place of the older/less functional Tkinter tool kit. Thanks to Klaus Pontoppidan for useful advice in wxpython. This should offer better cross-platform support and improved long term extensibility. The existing Tkinter GUI remains in place as well.
  - The calculation options dialog box now has an option to toggle between monochromatic and broadband calculations. In monochromatic mode, the "# of wavelengths" field is replaced by a "wavelength in microns" field.
  - There is also an option to toggle the field of view size between units of arcseconds and pixels.
  - Log messages giving details of calculations are now displayed in a window as part of the GUI as well.
  - The wx gui supports rectangular fields of view. Simply enter 2 elements separated by a comma in the 'Field of view' text box. As a convenience, these are interpreted as (X,Y) sizes. (Note that this is opposite of the convention used in the programming interface noted above; this is potentially confusing but seems a reasonable compromise for users of the webbpsf GUI who do not care to think about Python conventions in axis ordering. Comments on this topic are welcome.)
- Improved configuration settings system. Many settings such as default oversampling, default field of view size, and output file format can now be set in a configuration file for persistence between sessions. So if you always want e.g. 8x oversampling, you can now make that the default. An example configuration file with default values will be created automatically the first time you run webbpsf now, including informative comments describing possible settings. This file will be in your astropy config directory, typically something like "~/.astropy/config".
  - New 'Preferences' dialog allows changing these persistent defaults through the GUI.
- New function webbpsf.setup_logging() adds some more user-friendliness to the underlying python logging system. This includes persistent log settings between sessions. See updated documentation in the webbpsf page.
- The first time it is invoked on a computer, WebbPSF will display a welcome message providing some information of use to new users. This includes checking whether the requisite data files have been installed properly, and alerting users to the location of the configuration file, among other things.
- Refactoring of instrument class and rebalancing where the lines between WebbPSF and POPPY had been blurry.
- Some bugfixes in the example code. Thanks to Diane Karakla, Anand Sivaramakrishnan, Schuyler Wolff.
- Various updates & enhancements to this documentation. More extensive documentation for POPPY now available as well. Doc theme derived from astropy.
- Improved unit test suite and test coverage. Integration with Travis CI for continuous testing: https://travis-ci.org/mperrin/webbpsf
- Updated to astropy package helpers framework 0.4.4

14.3.11 Version 0.2.8

Released May 18, 2012

- Repaired functionality for saving intermediate opticals planes

- Coronagraph pupil shear shifts now use scipy.ndimage.shift instead of numpy.roll to avoid wrapping pixels around the edge of the array.
- Significant internal code reorganizations and cleanup:
  - switched package building to use `setuptools` instead of `distutils`/`stsci_distutils_hack`
  - poppy now installed as a separate package to more easily allow direct use.
  - 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.
  - Better packaging in general, with more attention to public/private API consistency
  - Built-in test suite available via `python setup.py test`
- Minor fix to MIRI ND filter transmission curve (Note: MIRI ND data is available on internal STScI data ditribution only)
- Binset now specified when integrating across bandpasses in pysynphoteliminating a previous warning message for that calculation.
- Stellar spectra are now by default drawn from the PHOENIX models catalog rather than the Castelli & Kurucz 2004 models. This is because the PHOENIX models have better spectral sampling at mid-infrared wavelengths.
- Default centroid box sizes are now consistent for measure_centroid() and the markcenter option to display_PSF(). (Thanks to Charles Lajoie for noting the discrepancy)
- TFI class (deprecated in version 0.2.6) now removed.

14.3.12 Version 0.2.7

Released December 6, 2011

- Bug fix for installation problems in previous release 0.2.6 (thanks to Anand Sivaramakrishnan and Kevin Flaherty for bringing the problem to my attention).
- Updated FITS keywords for consistency with JWST Data Management System (DMS) based on DMS Software Design Review 1.
  - "PUPIL" keyword now is used for pupil mechanisms instead of OTE pupil intensity filename; the filename is available in "PUPILINT" now, for consistency with the OPD filename in "PUPILOPD" now.
  - "CORONMSK" instead of CORON
  - Some minor instrument-specific FITS keywords added via new _instrument_fits_header() functions for each instrument object.
  - For instance, NIRCam PSFs now have "MODULE" and "CHANNEL" keywords (eg. "MODULE = A", "CHANNEL = Short"). Note that there is no optical difference between modules A and B in this version of webbpsf.
- Added support for weak lenses in NIRCam. Note that the +4 lens is in the filter wheel and is coated with a narrowband interference filter similar to but wider than F212N. WebbPSF currently does not model this, and will let you simulate weak lens observations with any filter you want. As always, it's up to the user to determine whether a given webbpsf configuration corresponds to an actual physically realizable instrument mode.

14.3.13 Version 0.2.6

Released November 7, 2011

- Updated & renamed TFI -> NIRISS.

- Removed etalon code.
- Added in filters transmissions copied from NIRCam
- Removed coronagraphic Lyot pupils. Note: the coronagraphic occulting spots are machined into the pick-off mirror so will still fly, and thus are retained in the NIRISS model.
- Slitless spectroscopy not yet supported; check back in a future version.
- Fix to FITS header comments for NIRISS NRM mask file for correct provenance information.
- TFI class still exists for back compatibility but will no longer be maintained, and may be removed in a future version of webbpsf.

• Strehl measurement code caches computed perfect PSFs for improved speed when measuring many files.

• Added GUI options for flat spectra in F_nu and F_lambda. (Thanks to Christopher Willmer at Steward Observatory for this suggestion)

• “display_psf” function renamed to “display_PSF” for consistency with all-uppercase use of PSF in all function names.

• numpy and pylab imports changed to ‘np’ and ‘plt’ for consistency with astropy guidelines (http://astropy.wikispaces.com/Astropy+Coding+Guidelines)

• poppy.py library updates (thanks to Anand Sivaramakrishnan for useful discussions leading to several of these improvements):
  - `Rotation` angles can be specified in either degrees or radians. Added units parameters to Rotations.__init__
  - `OpticalElement` objects created from FITS files use the filename as a default optic name instead of “unnamed optic”.
  - `FITSOpticalElement` class created, to separate FITS file reading functionality from the base OpticalElement class. This class also adds a ‘pixelscale’ keyword to directly specify the pixel scale for such a file, if not present in the FITS header.
  - Removed redundant ‘pupil_scale’ attribute: ‘pixelscale’ is now used for both image and pupil plane pixel scales.
  - unit test code updates & improvements.

• Miscellaneous minor documentation improvements.

14.3.14 Version 0.2.5

Initial public release, June 1 2011. Questions, comments, criticism all welcome!

• Improved spectrum display
• Improved display of intermediate results during calculations.

14.3.15 Versions 0.2.1 - 0.2.3

• Smoother installation process (thanks to Anand Sivaramakrishan for initial testing)
• Semi-analytic coronagraphic algorithm added for TFI and NIRCam circular occulters (Soummer et al. 2007)
• Advanced settings dialog box added to GUI
• NIRCam pixel scale auto-switching will no longer override custom user pixelscales.

- slight fix to pupil file pixel scales to reflect JWST flat-to-flat diameter=6.559 m rather than just "6.5 m"
- Corrected NIRCam 430R occulter profile to exactly match flight design; other occulters still need to be tuned. Corrected all for use of amplitude rather than intensity profiles (thanks to John Krist for comparison models).
- added TFI NRM mode (thanks to Anand Sivaramakrishnan)

14.3.16 Version 0.2

Initial STScI internal release, spring 2011. Questions, comments, criticism all welcome!

- Much improved pysynphot support.
- Reworked calling conventions for calcPSF() routine source parameters.
- poppy.calcPSFmultiprocessor merged in to regular poppy.calcPSF
- Minor bug fixes to selection of which wavelengths to compute for more even sampling
- Default OPDs are now the ones including SI WFE as well as OTE+ISIM.
- Improved fidelity for NIRCam coronagraphic occulter models including ND squares and substrate border.

14.3.17 Version 0.1

Development, fall 2010.

- Support for imaging mode in all SIs and FGS
- Support for coronagraphy with MIRI, NIRCam, and TFI. Further enhancements in fidelity to come later. Coronagraphic calculations are done using the direct FFT method, not Soummer's semi-analytic method (though that may be implemented in the future?).
- Up-to-date science frame axes convention, including detector rotations for MIRI and NIRSpec.
- Tunable wavelengths and appropriate bandwidths for TFI.
- Partial support for modeling IFU PSFs through use of the 'monochromatic' parameter.
- Revision V OPD files for OTE and SIs. Produced by Ball Aerospace for Mission CDR, provided by Mark Clampin.

CHAPTER 15

Appendix: Available Optical Path Difference (OPD) files

For each of the five instruments (four SIs plus FGS) there are three provided OPD files. These represent wavefronts as follows:

1. The OTE and ISIM intrinsic WFE
2. The above, plus a slight defocus to blur the image slightly to approximate image motion.
3. The above #2, plus additional WFE due to SI internal optics.

The latter is the largest WFE, and is the default file used in simulations unless another is explicitly chosen. For NIRCam only there is a second, duplicate set of these files with slightly improved WFE based on an optimistic case scenario for instrument and telescope alignment.

The provided OPDs are based on the observatory design requirements, and were developed for the Mission Critical Design Review. The represent the nominal case of performance for JWST, and have not yet been updated with as-built details of mirror surface figures, etc. We intend to make updated OPD files available once suitable reference data have been provided to STScI. For now, see Lightsey et al. 2014 for recent predictions of JWST's likely performance

Note that the trick of adding some (nonphysical) defocus to blur out the PSF is computationally easy and rapid, but does not give a high fidelity representation of the true impact of image jitter. This is particularly true for coronagraphic observations. Future versions of WebbPSF will likely provide higher fidelity jitter models.

The units of the supplied OPD files are wavefront error in microns.

Table 1: Rev V OPDs

File	Instrument	RMS WFE	Includes OTE + ISIM OPD?	Image motion (as defocus)?	SI OPD?
OPD_RevV_fgs_150.fits	FGS	150.0	Yes	No	No
OPD_RevV_fgs_163.fits	FGS	163.0	Yes	Yes	No
OPD_RevV_fgs_186.fits	FGS	186.0	Yes	Yes	Yes
OPD_RevV_miri_204.fits	MIRI	204.0	Yes	No	No
OPD_RevV_miri_220.fits	MIRI	220.0	Yes	Yes	No
OPD_RevV_miri_421.fits	MIRI	421.0	Yes	Yes	Yes
OPD_RevV_nircam_115.fits	NIRCam	115.0	Yes, optimistic case	No	No
OPD_RevV_nircam_123.fits	NIRCam	123.0	Yes	No	No
OPD_RevV_nircam_132.fits	NIRCam	132.0	Yes, optimistic case	Yes	No
OPD_RevV_nircam_136.fits	NIRCam	136.0	Yes	Yes	No
OPD_RevV_nircam_150.fits	NIRCam	150.0	Yes, optimistic case	Yes	Yes
OPD_RevV_nircam_155.fits	NIRCam	155.0	Yes	Yes	Yes
OPD_RevV_nirspec_125.fits	NIRSpec	125.0	Yes	No	No
OPD_RevV_nirspec_145.fits	NIRSpec	145.0	Yes	Yes	No
OPD_RevV_nirspec_238.fits	NIRSpec	238.0	Yes	Yes	Yes
OPD_RevV_niriss_144.fits	NIRISS	144.0	Yes	No	No
OPD_RevV_niriss_162.fits	NIRISS	162.0	Yes	Yes	No
OPD_RevV_niriss_180.fits	NIRISS	180.0	Yes	Yes	Yes

CHAPTER 16

Appendix: Instrument Property References

We give here references for the instrumental properties assumed in PSF computations, with particular attention to coronagraphic optics. It also notes several places where the current models or available files are limited in some manner that might be improved in a future release.

Instrument pixel scales are all based on *average best estimate* scales available in April 2016, specifically from values in the Science Instruments Aperture File (SIAF) data, as provided by the various instrument teams to the Telescope group via the SIAF Working Group. For instruments with multiple detectors, the values provided are averaged over the relevant detectors. WebbPSF calculates PSFs on an isotropic pixel grid (i.e. square pixels), but at high precision the SI pixel scales can differ between the X and Y axes by between 0.5% (for NIRCam) up to 2.5% (for FGS). WebbPSF also does not model any of the measured distortions within the instruments.

WebbPSF does not include any absolute throughput information for any SIs, only the relative weighting for different wavelengths in a broadband calculation. See *the note on PSF normalization* for further discussion.

Note: The WebbPSF software and all of its associated data files are entirely ITAR-free.

16.1 OTE

The supplied OPDs are the Mission CDR OPD simulation set, produced in March 2010 by Ball Aerospace staff (Paul Lightsey et al.) via the IPAM optical model using Zernike WFE coefficients consistent with Revision V of the JWST optical error budget.

Note: The provided files included no header metadata, and in particular no pixel scale, so one was assumed based on the apparent pupil diameter in the files. The estimated uncertainty in this scale is 1 part in 1000, so users concerned with measurements of PSF FWHMs etc at that level should be cautious.

The current model pixel scale, roughly 6 mm/pixel, is too coarse to resolve well the edge roll-off around the border of each segment. We make no attempt to include such effects here at this time. An independent study using much more finely sampled pupils has shown that the effect of segment edge roll-off is to scatter ~2% of the light from the PSF core out to large radii, primarily in the form of increased intensity along the diffraction spikes (Soummer et al. 2009, Technical Report JWST-STScI-001755)

16.2 NIRCam

NIRCam focal plane scale: 0.0311 +- 0.0002 (short wave), 0.0630 +- 0.0002 (long wave). SOC PRD SIAF PRDDEVSOC-D-012, 2016 April

The coronagraph optics models are based on the NIRCam instrument team's series of SPIE papers describing the coronagraph designs and flight hardware. (Krist et al. 2007, 2009, 2010 Proc. SPIE), as clarified through cross checks with information provided by the NIRCam instrument team (Krist, private communication 2011). Currently, the models include only the 5 arcsec square ND acquisition boxes and not the second set of 2 arcsec squares.

Weak lenses: The lenses are nominally +- 8 and +4 waves at 2.14 microns. The as built defocus values are as follows based on component testing: 7.76198, -7.74260, 3.90240.

16.3 NIRSpec

NIRspec field of view rotation: 138.4 degrees (average over both detectors). SOC PRD SIAF PRDDEVSOC-D-012, 2016 April

NIRSpec pixel scale 0.1043 +- 0.001 arcsec/pixel. SOC PRD SIAF PRDDEVSOC-D-012, 2016 April

NIRSpec internal pupil at the grating wheel: based on size of grating stop in Zemax file as analyzed and back projected onto the primary mirror by Erin Elliott, private communication 2013.

16.4 NIRISS

NIRISS focal plane scale, 0.0656 +- 0.0005 arcsec/pix: SOC PRD SIAF PRDDEVSOC-D-012, 2016 April

Occulting spots: Assumed to be perfect circles with diameters 0.58, 0.75, 1.5, and 2.0 arcsec. Doyon et al. 2010 SPIE 7731. While these are not likely to see much (any?) use with NIRISS, they are indeed still present in the pickoff mirror hardware, so we retain the ability to simulate them.

NIRISS internal pupils: The regular imaging mode internal pupil stop is a 4% oversized tricontagon (with sharp corners). See Doyon et al. Proc SPIE 2012 Figure 2. The CLEARP pupil has an oversized central obscuration plus 3 support vanes. Details based on NIRISS design drawing 196847Rev0.pdf "Modified Calibration Optic Holder" provided by Loic Albert. NRM occulter mask: digital file provided by Anand Sivaramakrishnan. GR700XD mask design details provided by Loic Albert.

16.5 MIRI

MIRIM focal plane scale, 0.1110 +- 0.001 arcsec/pix: SOC PRD SIAF PRDDEVSOC-D-012, 2016 April

MIRIM field of view rotation, 5.0152 degrees: SOC PRD SIAF PRDDEVSOC-D-012, 2016 April

Coronagraph pupils rotated to match, 4.56 degrees: MIRI-DD-00001-AEU 5.7.8.2.1

Coronagraphic FOVs, 30.0 arcsec for Lyot, 24.0x23.8 arcsec for FQPMs: MIRI-DD-00001-AEU 2.2.1

Lyot coronagraph occulting spot diameter, 4.25 arcsec:

Lyot coronagraph support bar width, 0.46 mm = 0.722 arcsec: Anthony Boccaletti private communication December 2010 to Perrin and Hines

Lyot mask files: Anthony Boccaletti private communication to Remi Soummer

LRS slit size (4.7 x 0.51 arcsec): MIRI-TR-00001-CEA. And LRS Overview presentation by Silvia Scheithaur to MIRI team meeting May 2013.

LRS P750L grating aperture mask (3.8% oversized tricontagon): MIRI OBA Design Description, MIRI-DD-00001-AEU

MIRI imager internal pupil stop in regular imaging mode: 4% oversized tricontagon, mounted on each of the imaging filters (with smoothed corners).

16.6 Instrument + Filter Throughputs

Where possible, instrumental relative spectral responses were derived from the Pysynphot CDBS files used for the development version of the JWST Exposure Time Calculators (ETCs), normalized to peak transmission = 1.0 (because absolute throughput is not relevant for PSF calculations). Not all filters are yet supported in Pysynphot, however.

For the following filters we take information from alternate sources other than the CDBS:

```
Instrument    Filter         Source
-----------  -------------   ---------------------------------------------------------------------------
↪-------------------------------
NIRCam       F150W2          Top-hat function based on filter properties list at http://ircamera.as.
↪arizona.edu/nircam/features.html
NIRCam       F322W2          Top-hat function based on filter properties list at http://ircamera.as.
↪arizona.edu/nircam/features.html
NIRSpec      F115W           Assumed to be identical to the NIRCam one
NIRSpec      F140X           NIRSpec "BBA" transmission curve traced from NIRSpec GWA FWA Assembly␣
↪Report, NIRS-ZEO-RO-0051, section 6.3.2
MIRI         F*W filters     Data published in Glasse et al. 2015 PASP Vol 127 No. 953, p. 688 Fig 2
MIRI         F*C filters     Data published in Bouchet et al. 2015 PASP Vol 127 No. 953, p. 612 Fig 3
NIRISS       all filters     Measurement data provided by Loic Albert of the NIRISS team
FGS          none            Assumed top-hat function based on detector cut-on and cut-off wavelengths.
```

The MIRI wide filters (F*W) are total system photon conversion efficiencies including filter, telescope, instrument, and detector throughputs, normalized to unity. The MIRI coronagraphic filters are just the filters themselves, but the detector and optics throughputs are relatively flat with wavelength compared to the narrow coronagraphic filters. These are sufficiently accurate for typical coronagraphic modeling but be aware of that caveat if attempting precise photometric calculations.

For the NIRCam and NIRSpec filters called out in the table above, the provided throughputs do not include the detector QE or OTE/SI optics throughputs versus wavelength.

All other filters do include these effects, to the extent that they are accurately captured in the Calibration Database in support of the ETCs.

CHAPTER 17

Releasing a new version of WebbPSF

17.1 Prerequisites

- Is the build passing on Travis?
- Are you up to date with `master` on the upstream branch (spacetelescope/webbpsf)?
- Do you have twine installed?
- Do you have access to WebbPSF on PyPI with the owner or maintainer role?
- Do you have your `~/.pypirc` filled out? (more info)

17.2 Releasing new data packages

1. Run `dev_utils/make-data-sdist.sh` (details below) to make a gzipped tarred archive of the WebbPSF data
2. If the new data package is **required** (meaning you can't run WebbPSF without it, or you can run but may get incorrect results), you must bump `DATA_VERSION_MIN` in `__init__.py` to `(0, X, Y)`
3. Extract the resulting data archive and check that you can run the WebbPSF tests with `WEBBPSF_PATH` pointing to it
4. Copy the data archive into public web space
5. `cd` to `/grp/jwst/ote` and remove the `webbpsf-data` symlink
6. Copy the archive into `/grp/jwst/ote/` and extract it to `/grp/jwst/ote/webbpsf-data`
7. Rename the folder to `webbpsf-data-0.x.y`
8. Create a symbolic link at `/grp/jwst/ote/webbpsf-data` to point to the new folder
9. Update the URL in `installation.rst` under *Installing the Required Data Files*

Invoke `dev_utils/make-data-sdist.sh` one of the following ways to make a gzipped tarred archive of the WebbPSF data suitable for distribution.

If you are on the Institute network:

```
$ cd webbpsf/dev_utils/
$ ./make-data-sdist.sh 0.X.Y
$ cp ./webbpsf-data-0.X.Y.tar.gz /path/to/public/web/directory/
```

If you're working from a local data root:

```
$ cd webbpsf/dev_utils/
$ DATAROOT="/Users/you/webbpsf-data-sources/" ./make-data-sdist.sh 0.X.Y
$ cp ./webbpsf-data-0.X.Y.tar.gz /where/ever/you/want/
```

17.3 Releasing new versions on PyPI

1. Edit `relnotes.rst` to add a release date and reference anchor (e.g. `..  _rel0.X.Y:`) to the section for this release
2. Update the link to "What's new" in `index.rst`
3. Add any important notes to the appropriate section in the release notes
4. Edit `setup.py` in this repository to remove `.dev` from the version number in the `VERSION` variable
5. Build a source distribution with `python setup.py build sdist`
6. Copy the resulting file (`webbpsf-0.X.Y.tar.gz`) to a new folder, extract it, and `cd` there
7. Run `python setup.py test` (preferably in a new `virtualenv` containing only the WebbPSF dependencies) and verify that the test suite passes with the code you're about to release
8. If that runs as expected, `cd` back to your `webbpsf` repository and run `twine upload dist/webbpsf-0.X.Y.tar.gz` for your new version
9. Verify that the latest version is visible and others are hidden on the PyPI package editing page

17.3.1 Finishing the release

1. Commit your edits to `relnotes.rst` and `setup.py`
2. Tag that commit as the release with `git tag v0.X.Y` and push the tags to origin and upstream with `git push --tags origin` and `git push --tags upstream`
3. Edit `setup.py` to increment the version number in the `VERSION` variable and re-add the `.dev` suffix
4. Edit `relnotes.rst` to add a new heading for the upcoming version
5. Commit your edits with a message like "Back to development: version 0.X.Y+1"
6. Email an announcement to `webbpsf-users@stsci.edu`

17.4 Releasing a new version through AstroConda

To test that an Astroconda package builds, you will need `conda-build`:

```
$ conda install conda-build
```

1. Fork (if needed) and clone https://github.com/astroconda/astroconda-contrib
2. If there is a new version of POPPY available to package, edit poppy/meta.yaml to reflect the new `version` and `git_tag`.
3. If the minimum needed version of the webbpsf-data package has changed in `webbpsf/__init__.py`, edit webbpsf-data/meta.yaml to reflect the new `version` and `url`.
4. Edit webbpsf/meta.yaml to reflect the new versions of POPPY and webbpsf-data, if necessary.
5. Edit in the `git_tag` name from `git tag` in the PyPI release instructions (`v0.X.Y`).
6. Verify that you can build the package from the astroconda-contrib directory: `conda build -c http://ssb.stsci.edu/astroconda webbpsf`
7. Commit your changes to a new branch and push to GitHub.
8. Create a pull request against `astroconda/astroconda-contrib`.
9. Wait for SSB to build the conda packages.
10. (optional) Create a new conda environment to test the package installation following *these instructions*.

How to cite WebbPSF

In addition to this documentation, WebbPSF is described in the following references. Users of WebbPSF are encouraged to cite one of these.

- Perrin et al. 2014, "Updated point spread function simulations for JWST with WebbPSF", Proc. SPIE. 9143,
- Perrin et al. 2012, "Simulating point spread functions for the James Webb Space Telescope with WebbPSF", Proc SPIE 8842, and
- Perrin 2011, `Improved PSF Simulations for JWST: Methods, Algorithms, and Validation`, JWST Technical report JWST-STScI-002469.

In particular, the 2012 SPIE paper gives a broad overview, the 2014 SPIE paper presents comparisons to instrument cryotest data, and the Technical Report document describes in more detail the relevant optical physics, explains design decisions and motivation for WebbPSF's architecture, and presents extensive validation tests demonstrating consistency between WebbPSF and other PSF simulation packages used throughout the JWST project.

- genindex
- search

Mailing List

If you would like to receive email announcements of future versions, please contact Marshall Perrin, or visit `maillist.stsci.edu` to subscribe yourself to the "webbpsf-users@maillist.stsci.edu" list.

Python Module Index

Index