webbpsf Documentation

Release 1.0.0

Association of Universities for Research in Astronomy

Sep 23, 2022

CONTENTS

WebbPSF is a Python package that computes simulated point spread functions (PSFs) for NASA's James Webb Space Telescope (JWST) and Nancy Grace Roman Space Telescope (formerly WFIRST). 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 Roman's instruments.

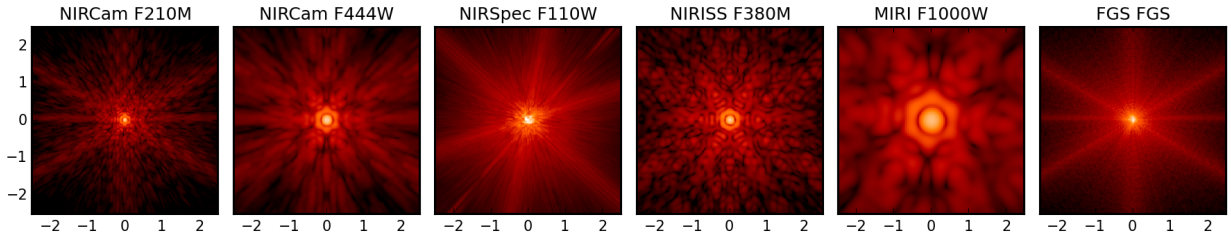


Fig. 1: Sample PSFs for JWST's instrument suite, all on the same angular scale and display stretch.

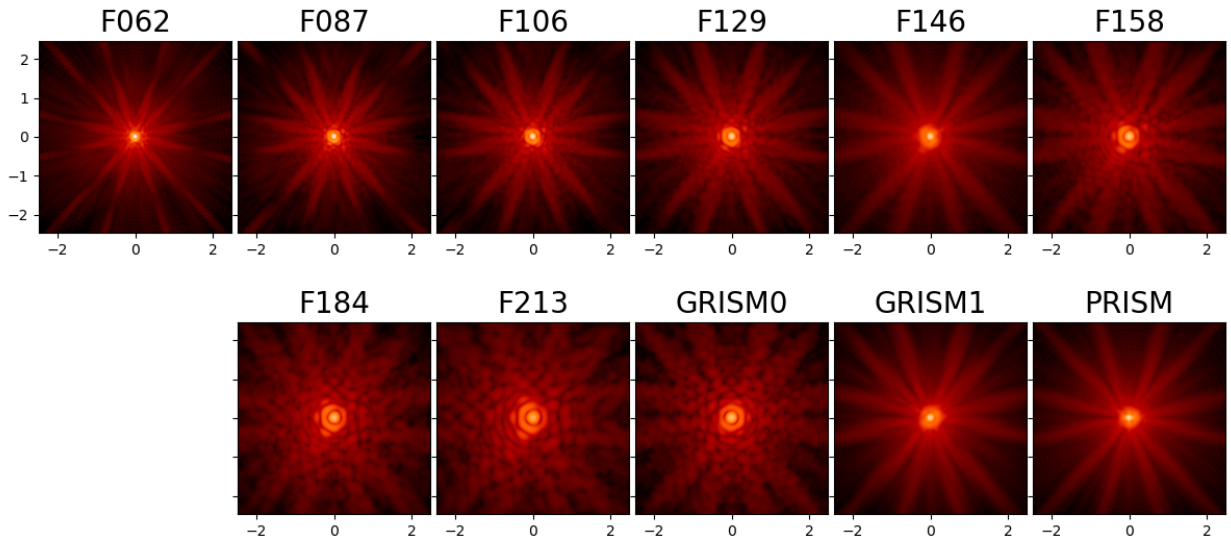


Fig. 2: Sample PSFs for the filters in the Roman WFI.

Contributors: WebbPSF has been developed by Marshall Perrin, Shannon Osborne, Robel Geda, Joseph Long, Justin Otor, Jarron Leisenring, Neil Zimmerman, Keira Brooks, and Anand Sivaramakrishnan, with contributions from Marcio Meléndez Hernandez, Alden Jurling, Lauren Chambers, Ewan Douglas, Charles Lajoie, Megan Sosey, and Kathryn St.Laurent. We also are grateful to the contributors of the astropy-helpers and stsci package templates.

Part I

Getting Started with WebbPSF

See *Using WebbPSF*.

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.

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

What's new in the latest release?

Part II

Contents

CHAPTER

ONE

INTRODUCTION

What this software does:

- Uses OPD maps precomputed by detailed optical simulations of JWST and the Nancy Grace Roman Space Telescope (formerly 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 Roman, 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 Roman. 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 Roman modeling, see the Roman 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.)

Conceptually, this simulation code has two layers of abstraction:

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

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 Roman, 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

See *Requirements & Installation*.

1.4 Quick Start

Once you have installed the software and data files, we recommend you begin with the Jupyter Notebook quickstart tutorial. Downloading and running that notebook is a great way to get started using WebbPSF.

CHAPTER

TWO

REQUIREMENTS & INSTALLATION

Note: This is entirely optional, but you may wish to sign up to the mailing list `webbpsf-users@maillist.stsci.edu`. This is a very 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

2.1 Recommended: 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
```

Updates to the latest version can be done as for any conda package:

```
(astroconda)$ conda update webbpsf
```

2.2 Installing with conda (but not 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** *described* **below.** To obtain source spectra for calculations, you should also follow *installation instructions for synphot*.

2.4 Installing or updating synphot

Stsynphot is an optional dependency, but is highly recommended. Stsynphot is best installed via AstroConda. Further installation instructions can be found in the synphot docs or a discussion 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-1.0.0.tar.gz [approx. 280 MB]
2. Untar `webbpsf-data-1.0.0.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.

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.

Note: 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

See the environment.yml specification file for the required package dependencies.

Required Python version: WebbPSF 1.0 and above require Python 3.7 or higher.

The major dependencies are the standard NumPy, SciPy, matplotlib stack, and Astropy

Recommended Python packages:

- synphot enables the simulation of PSFs with proper spectral response to realistic source spectra. Without this, PSF fidelity is reduced. See above for *installation instructions for synphot*. Stsynphot 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

THREE

RELEASE NOTES

3.1 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.

3.2 Version History and Change Log

3.2.1 Version 1.0.0

2021 December 10

For JWST, this release includes updates to WebbPSF just prior to the launch. For Roman, it includes updates to use the Cycle 9 optical model results.

James Webb Space Telescope OTE model improvements:

- Updates in sign conventions for representing WFE, for strict consistency with the JWST WSS and other tools. Much of this was implemented by upstream changes in `poppy`; see this page in the poppy docs for details. (#397, #419 by @mperrin, #418 by @Skyhawk172)
- Significant update to JWST OTE optical models, to reflect more recent 2020 optical modeling of the as-built observatory (the "PSR2020" integrated modeling cycle). These have noticably lower WFE than the prior models (which were intentionally conservative, but ended up being more conservative than intended); typically the WFE is lower by some tens of nanometers in the new "prelaunch_predicted" OPDs. See details in *Optical Telescope Element (OTE)*. We will all learn together in 2022 how well these models predict the observatory's performance in flight. (#512, @mperrin).

- Add models of OTE field dependence from the nominal OTE design and as-built optics (#389 by @grbrady, #505 by @mperrin) and from any misalignment of the secondary mirror, such as would be measured and corrected in MIMF (#392 by @Skyhawk172). These additions were enabled by more consistent use of JWST Linear Optical Model framework behind the scenes (#378 by @mperrin). This model of field dependence plus the updated OTE OPD files should yield a more comprehensive and precise model of PSF variations across the observatory.
- Add an option to use a lookup table of field dependent OPDs from Ball's ITM tool (for JWST team internal use in pre-launch wavefront team practices and rehearsals). (#425 by @Skyhawk172, #474 by @mperrin)
- Update the JWST OTE Linear Model to allow more flexible pupil sampling, in particular using higher sampling to reduce Fourier aliasing in certain FGS calculations (#440 by @kjbrooks)
- New capability for visualizing the JWST optical budget terms as represented in WebbPSF. See jwst_optical_budgets.

James Webb Space Telescope instrument model improvements:

- MIRI: Minor updates to pixel scale and rotation (#456 by @mperrin), an improved model of the MIRI imager detector cross artifact (#417 by @mperrin) and correctly label MIRI's P750L prism for the LRS mode as a prism, not a grating (#477 by @mperrin and @skendrew)
- MIRI: Add capability for shifting MIRI coronagraph masks, consistent with NIRCam sim capabilities (#428 by @JarronL)
- NIRCam: Higher fidelity model of NIRCam weak lenses, including field dependence, non-linear interactions between lenses, and as-built measured performances. (#496 by @mperrin, using results of calibration work by Randal Telfer)
- All SIs: Substantial performance improvements speeding up the calculation of optical distortion (#429, @jarronL)

Nancy Grace Roman Space Telescope and instrument model improvements:

- Use of Cycle 9 optical and integrated modeling results, including updated Zernike coefficients, pupil images, and filter throughputs.
- Updated `RomanInstrument` pointing stability to 12 milliarcseconds per axis, following new predictions [#466 by @ojustino with @robelgeda]
- `WFI` wavelength range now covers 0.48 - 2.3 μm [#466 by @ojustino with @robelgeda]
- Added `WFI`'s new F213 filter [#466 by @ojustino with @robelgeda]
- Renamed `WFI`'s `'P120'` filter to `'PRISM'` [#466 by @ojustino with @robelgeda]
- Split `WFI`'s `'G150'` filter into `'GRISM0'` and `'GRISM1'` components to represent the transmission for the grism's undispersed zeroth order and dispersed first order, respectively [#466 by @ojustino with @robelgeda]
- Renamed WFI pupil masks to `'SKINNY'` (formerly `'RIM_MASK'` in version 0.9.2), `'WIDE'` (formerly `'FULL_MASK'`), `'GRISM'`, and `'PRISM'` (also formerly captured in `'RIM_MASK'`) [#466 by @ojustino with @robelgeda]
- Created new *`lock_pupil()`* and *`lock_pupil_mask()`* methods for advanced users who prefer to disable automated selections and instead stick with a specific pupil file or mask, respectively. The corresponding `WFI.unlock_pupil()` and `WFI.unlock_pupil_mask()` methods return the class to its normal behavior [#466 by @ojustino with @robelgeda]
- Locked `WFI.pupil` and `WFI.pupil_mask` attributes from direct assignment given the new lock/unlock schema [#466 by @ojustino with @robelgeda]
- Renamed `WFI.override_aberrations()` to *`lock_aberrations()`* and `WFI.reset_override_aberrations()` to *`unlock_aberrations()`* to reinforce the new lock/unlock schema [#466 by @ojustino with @robelgeda]

- Condensed and refactored existing tests [#466 by @ojustino with @robelgeda]
- New algorithm for field point nearest approximation/extrapolation [#466 by @ojustino with @robelgeda]
- Renamed `CGI` class to `RomanCoronagraph` [#516, #517, @ojustino with @mperrin]

Software and Package Infrastructure Updates:

- Software engineering improvements to meet STScI INS-JWST Software Standards (#404 by @shanosborne)
- Migrate optional dependency for synthetic photometry from pysynphot to synphot (#424 by @shanosborne)
- Deprecated the `jwxml` package, and moved the SUR (Segment Update Request) parsing code from that package into WebbPSF (#390 by @shanosborne)
- Various minor bug fixes (#410, #422, #427, #497 by @mperrin, #423 by @kjbrooks, #493 by @JarronL)
- Updates to recommended (not minimum) dependency versions. Drop support for Python 3.6. (various PRs by @shanosborne)
- Remove deprecated older code including the GUIs (#439 by @mperrin)
- Streamline test suite to keep CI runtimes managable (#459 by @mperrin)

3.2.2 Version 0.9.2

2021 July 23

This release only improves a subset of WFIRST functionality; additional improvements to both WFIRST (including renaming to Roman) and JWST models will be at the upcoming 1.0.0 major release.

WFIRST Improvements

- New Grism and Prism filters: [#416, #471, @robelgeda]
  - `GRISM_FILTER = 'G150'`
  - `PRISM_FILTER = 'P120'`
- Changing filters to `G150` or `P120` changes the mode of the WFI and the aberrations files (unless there is a user aberrations override) [#416, #471, @robelgeda]
- New `WFI.mode`: Class property that returns the current mode of the WFI instance by passing the current filter to `WFI. _get_filter_mode`. WFI modes are: [#416, #471, @robelgeda]
  - Imaging
  - Grism
  - Prisim
- New `WFI.override_aberrations(aberrations_path)`: Overrides and locks the current aberrations with aberrations at `aberrations_path`. Lock means changing the filter/mode has no effect on the aberrations. [#416, #471, @robelgeda]
- New `WFI.reset_override_aberrations()`: Releases `WFI.override_aberrations` lock and start using the default aberrations. [#416, #471, @robelgeda]
- New Tests for mode and filter switching. [#416, #471, @robelgeda]
- New Field point nearest point approximation (extrapolation). [#416, #471, @robelgeda]

Software and Package Infrastructure Updates:

- This release uses Github Actions CI and removes TravisCI. [#455, @shanosborne, #471, @robelgeda]

3.2.3 Version 0.9.1

2020 June 22

This minor release resolves several bugs and occasional installation issues and updates behind-the-scenes package infrastructure for consistency with current astropy and numpy releases. There are small improvements to a few aspects of JWST models as detailed below (in particular for wavelength dispersion in NIRCam LW coronagraphy and in tools for modeling time-dependent WFE) but the vast majority of JWST PSF calculations are not changed in any way.

There are no changes in reference data, so the WebbPSF reference data files for 0.9.0 should continue to be used with this release.

Python version support: Python 3.6+ required

This version drops support for Python 3.5. The minimum supported version of Python is now 3.6.

JWST Improvements

- *Apply wavelength dependent offsets for NIRCam coronagraphic PSFs* due to the dispersion from the optical wedge in the coronagraphic pupil masks. This primarily affects the LW channel with approximately 0.015 mm/um dispersion. The SW channel is almost a factor of 10 smaller and mostly negligible, but has been included for completeness. [#347, @JarronL]
- *Improved models for OTE wavefront variations over time* by adding utility functions for decomposing WFE models into piston, tip, tilt motions in the JWST control coordinate system, adding a model for frill-induced WFE drift, adding a model for IEC-heater-induced WFE drift, and adding an option to adjust amplitude of OTE backplane thermal drift model for B.O.L. vs E.O.L. expected amplitudes. [#340, @mperrin]
- *Add new* `aperturename` *attribute* for JWST instruments which returns the SIAF aperture name used for transforming between the detector position and instrument field of view on the sky. [#360, @mperrin]. Releatedly, improves setting of detector geometry for NIRCam to automatically set the SIAF aperture name based on detector, filter, and coronagraph image mask and pupil mask settings. This can be turned off by setting `auto_apname=False`. [#351, @JarronL]
- Add model for image jitter with JWST in coarse point mode under two different assumptions about LOS stability. This is relevant only for commissioning simulations. [#345, #346, @mperrin]
- Documentation updates, in particular adding *figures of JWST instrument internal wavefront error models*. [#369, @mperrin]

General bug fixes and small changes:

- Allow FGS detector to be set to `GUIDER1` and `GUIDER2`, while still supporting old method of setting the detector (using `FGS1` and `FGS2`) [#361, @mperrin]
- Add `allow_huge=True` option to `astropy.convolution.convolve_fft` call when applying MIRI distortion so it can handle large arrays when calculating PSFs in very large FOV by using a higher resolution pupil and OPD. [#354, @obi-wan76]
- Fixed bug that caused an error when plotting OPDs using the `display_opd` function [#362, @shanosborne]
- Update default NIRSpec detector coordinates to be the S1600A1 square aperture coordinates in imaging mode, rather than an implausible location outside of the MSA field of view. [#348, @mperrin]
- Updated Simulated OTE Mirror Move Demo notebook. [#343, @kjbrooks]

- Improved the reproducibility of the thermal slew model with small updates to the `update_opd` and `move_jsc_acf` functions. [#339, @mperrin]

Software and Package Infrastructure Updates:

- *The minimum Python version is now 3.6.* [#353, @mperrin]
- Removed dependency on `astropy-helpers` sub-package [#337, @shanosborne]
- Fixed problem that resulted in the `otelm/` and `tests/surs/` sub-directories not installing correctly. [#356, @shanosborne]
- Removed python 3.5 testing and added python 3.8 testing in Travis continuous integration. [#352, @mperrin]
- Documentation added and/or updated for a variety of features, including referencing the newly renamed Nancy Grace Roman Space Telescope (formerly WFIRST). [#364, #360, #330, @shanosborne@mperrin]

3.2.4 Version 0.9.0

2019 November 25

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-0.9.0.tar.gz.

JWST Improvements

- *Added a new capability to model the impact of thermal variations*, from telescope slews relative to the sun, onto mirror alignments and therefore onto PSFs. This new `thermal_slew` method can be used to create a delta OPD for some elapsed time after the slew at either the maximum slew angle, some specified angle, or with a scaling factor applied to maximum case. Once combined with an input OPD (requirements or predicted), the new shape of the mirrors can be used to simulate predicted PSFs some time after a slew. See this Jupyter notebook for examples. [#269, @kjbrooks]
- *Improved wavefront error extrapolation method for field points near FOV corners* that are outside the bounds of Zernike reference table data, in order to provide more seamless extrapolation. [#283, @JarronL]
- *Improvements in NIRCam optical model*: Updated polynomial model for NIRCam defocus versus wavelength. Adds Zernike coefficients for the wavefront error at NIRCam coronagraphy field points. [#283, @JarronL]
- NIRISS NRM mask was flipped along the X axis to match the as-built instrument and measured PSFs [#275, @KevinVolkSTScI, @anand0xff, @mperrin]
- Updated FGS throughput values to use data from the instrument sub-level testing that was done by Comdev/Honeywell, detector quantum efficiency as measured by Teledyne, and the OTE throughput from Lightsey 2012. The throughput file was also updated to include the WAVEUNIT keyword, which removes a warning. [@shanosborne]]

WFIRST Improvements

- *The WFI optical model has been updated to use optical data from the Cycle 8 design revision.* These include updated Zernike coefficients for field-dependent wavefront error, and masked and unmasked pupil images for each SCA, and updated filter throughputs (consistent with values used in Pandeia 1.4.2). The correct pupil file will automatically be selected for each calculation based on the chosen detector position and filter. The pupil files are consistent with those provided in the WFI cycle 8 reference information, but have been resampled onto a common pixel scale. See WFIRST instrument model details for more. [#309 @robelgeda]
- Note, WFI's filters have been renamed so they all begin with "F"; see the table here .
- *The WFI wavelength range has now been extended to cover the 0.48 - 2.0 μm range.* [#309 @robelgeda]

- *Expanded ``psf_grid`` method's functionality so it can also be used to make grids of WFIRST PSFs.* Note that focal plane distortion is not yet implemented for WFIRST PSFs and so `add_distortion` keyword should not be used for this case. [#294, @shanosborne]
- *The WFIRST F062 filter bandpass red edge was corrected* from 8000A to 7600A, and associated unit tests were updated to include F062 [#288, @robelgeda]
- The WFI simulations now include the pointing jitter model, using the predicted WFI pointing stability of 14 milliarcseconds per axis. [#322, @mperrin]

General bug fixes and small changes:

- *Many improvements in the PSF Grid functionality for generating photutils.GriddedPSFModels*:
  - New options in `psf_grid` to specify both/either the output filename and output directory location. See this Jupyter notebook for examples. [#294, @shanosborne]
  - sFfilenames when saving out a `psf_grid` FITS object which has it's `filename` parameter set will now end with `_det.fits` instead of the previous `_det_filt.fits` [#294, @shanosborne]
  - Update added to `utils.to_griddedpsfmodel` where a 2-dimensional array input with a header containing only 1 `DET_YX` keyword can be turned into `GriddedPSFModel` object without error as it implies the case of a PSF grid with num_psfs = 1. [#294, @shanosborne]
  - Remove deletion of `det_yx` and `oversamp` keywords from `psf_grid` output to allow for easier implementation in certain cases. Normal case users will have extra keywords but will not change functionality [#291, @shanosborne]
  - Updated normalization of PSFs from `psf_grid` to be in surface brightness units, independent of oversampling in order to match the expectation of `photutils.GriddedPSFModel`. This is diferent than webbpsf's default in which PSFs usually sum to 1 so the counts/pixel varies based on sampling. [#311, @mperrin]
  - Fix bug in how `pupilopd` keyword is saved and include extra keywords `opd_file`, `opdslice`, `coronmsk`, and `pupil` in the `psf_grid` output, both the GriddedPSFModel meta data and FITS object's header [#284, #293, #299, @shanosborne]
- The `set_position_from_aperture_name` method now correctly sets the detector position parameter in the science frame [#281, @shanosborne, @JarronL, @mperrin]
- Fix OPD HDUList output from the `as_fits` method inside the OPD class to include the previously existing header information [#270 @laurenmarietta]
- Added support for secondary mirror moves to the move_sur() method through the move_sm_local method [#295, @AldenJurling]
- Remove `units` keyword from `get_opd` method, now the wave input needs to be a Wavefront object [#304, @shanosborne]

Software and Package Infrastructure Updates:

- Added `environment.yml` file [#321, @shanosborne, @mperrin]
- Remove leftover deprecated syntax `_getOpticalSystem` for `_get_optical_system` and `display_PSF` for `display_psf` [#280, #294, @mperrin, @shanosborne]
- Various smaller code cleanup and doc improvements, including code cleanup for better Python PEP8 style guide compliance [@mperrin, @shanosborne, @robelgeda]
- Documentation added and/or updated for a variety of features [#277, #280, #318, @mperrin@shanosborne]

3.2.5 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.

3.2.6 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.

3.2.7 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.

3.2.8 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`

3.2.9 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.

3.2.10 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).

3.2.11 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.

3.2.12 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)

3.2.13 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.)

3.2.14 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

3.2.15 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.

3.2.16 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.

3.2.17 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 pickoff 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.

3.2.18 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.

3.2.19 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)

3.2.20 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.

3.2.21 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

FOUR

USING WEBBPSF

WebbPSF provides five classes corresponding to the JWST instruments and two for the Roman instruments, with consistent interfaces. It also provides a variety of supporting tools for measuring PSF properties and manipulating telescope state models. 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.

4.1 Usage and Examples

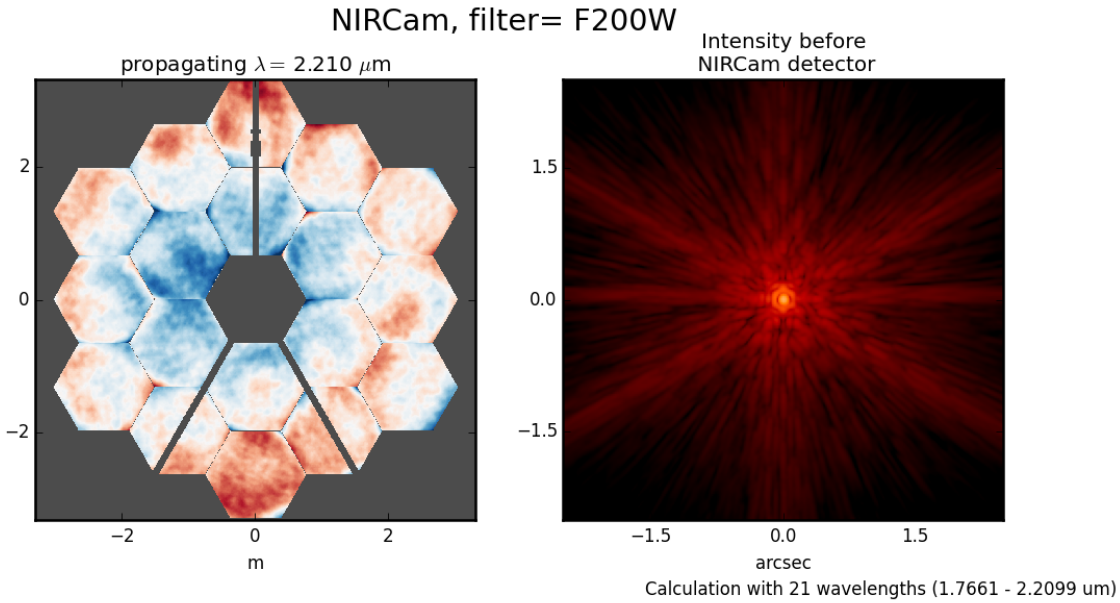
Simple PSFs are easily obtained.

Instantiate a model of *NIRCam*, set attributes to configure a particular observing mode, then call *calc_psf()*:

```
>>> 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.
```

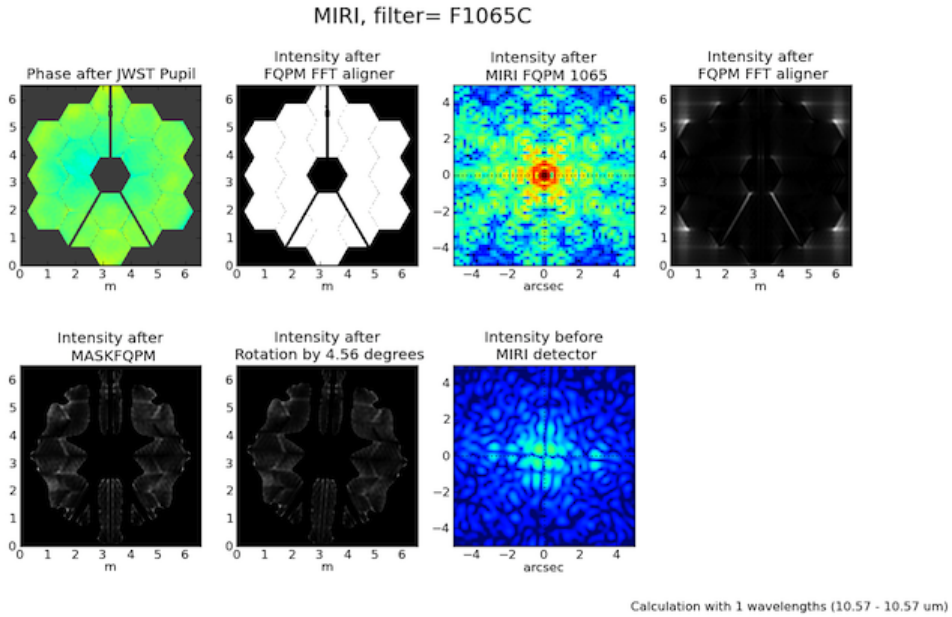
For interactive use, you can have the PSF displayed as it is computed:

```
>>> 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')
```



4.1.1 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 `synphot` is installed, a G2V star spectrum from Castelli & Kurucz 2004. Without `synphot`, 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 `synphot.SourceSpectrum` object. This is the best option, providing maximum ease and accuracy, but requires the user to have `synphot` installed. In this case, the `SourceSpectrum` object is combined with a `synphot.SpectralElement` 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 `synphot.SourceSpectrum` 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)
```

4.1.2 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)
```

4.1.3 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 per axis, 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.
>>>
>>> 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.

4.1.4 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.

   ```
   >>> hdulist = instrument.calc_psf(oversample=2)    # hdulist will contain a primary␣
   ↪HDU with the
   >>>                                                # oversampled data
   ```

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.

4.1.5 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.

4.1.6 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.

4.2 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.

4.2.1 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)
```

4.2.2 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.

4.2.3 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.

4.2.4 Calculating Data Cubes

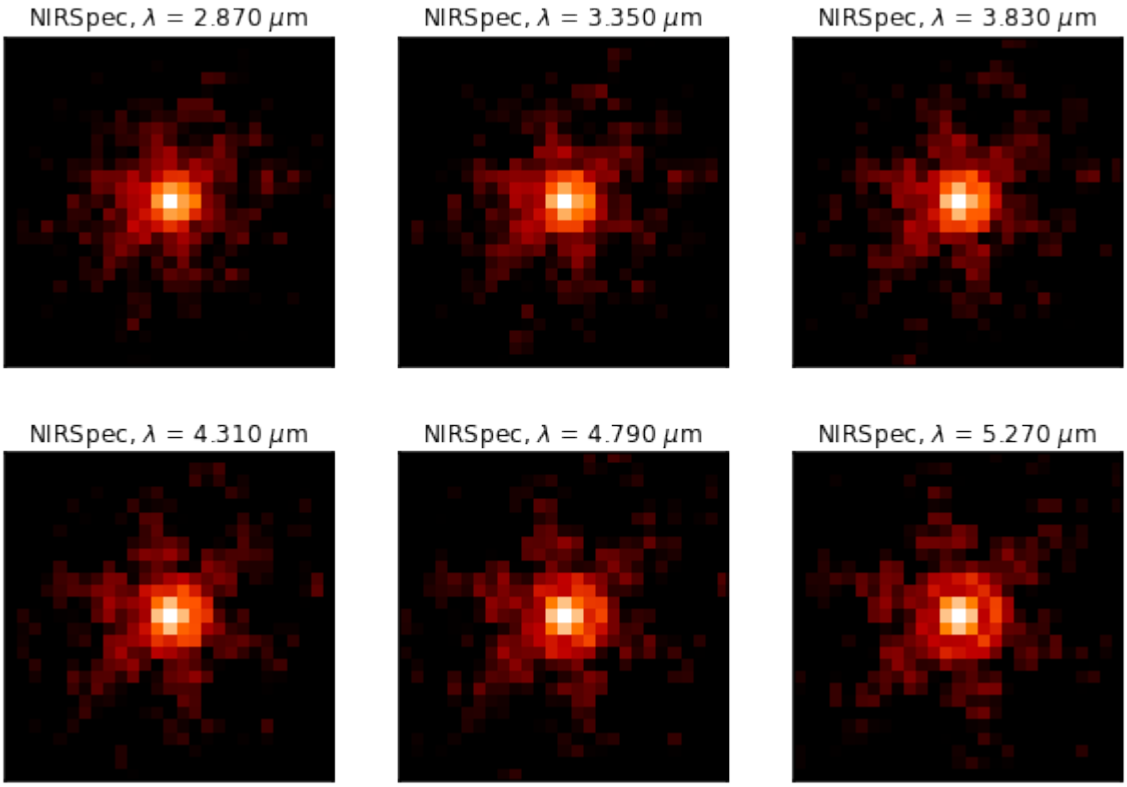
Sometimes it is convenient to calculate many PSFs at different wavelengths with the same instrument config. You can do this just by iterating over calls to `calc_psf`, but there's also a function to automate this: `calc_datacube`. For example, here's something loosely like the NIRSpec IFU in F290LP:

```
# 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
```

(continues on next page)

(continued from previous page)

```
        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)
```



4.2.5 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(
```

(continues on next page)

(continued from previous page)

```
>>>             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
>>>
>>> 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)
```

4.2.6 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

FIVE

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 order (X,Y).
- The `aperturename` attribute provides the SIAF aperture name used for transforming between detector position and instrument field of view on the sky. By default this will be a full-frame aperture for the currently-selected detector, but you may select any subarray aperture or other aperture named in the SIAF for that instrument. The aperturename will always update automatically when you select a new detector name. For NIRCam and MIRI, the aperturename can also (optionally) automatically update for coronagraphic subarrays if/when a coronagraphic optic is selected for the image or pupil mask. .

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.

5.1 JWST Optical Budgets

The total system performance for JWST is tracked in optical budgets for OTE and SI WFE. WebbPSF includes representations of many of these component terms. These can be visualized as plots of OPDs. See jwst_optical_budgets.

5.2 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 two "predicted" OPDs and a "requirements" OPD map. For example:

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

As of WebbPSF 1.0, WebbPSF selects the 'JWST_OTE_OPD_RevAA_prelaunch_predicted.fits.gz' OPD as the default OPD map for all instruments. This is a *significant change* from prior versions:

```
 >>> nc.pupilopd
'JWST_OTE_OPD_RevAA_prelaunch_predicted.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 "prelaunch_predicted" OPD maps provided with WebbPSF are based on a recent integrated modeling cycle, the so-called PSR2020 ("Predicted Stability Requirements 2020") modeling effort, and provide a reasonable approximation

of current performance expectations. However, performance at such levels is not guaranteed. See jwst_optical_budgets for more details on the contents of this OPD model.

The older "predicted" and "requirements" OPD maps are more conservative, dating to 2016. The Requirements map is 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. Further, they also include very conservative levels of instrument WFE, which is both higher than the as-built instruments *and* is double-booked relative to the SI WFE models elsewhere in webbpsf. These files are kept for consistency with past versions of WebbPSF, but we now know hopefully we may do better in flight.

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

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

For all provided WFE cases, the OPD files contain *10 Monte Carlo realizations of the telescope*, representing slight variations or uncertainties in the alignment process. 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 "prelaunch_predicted" OPD file is for the telescope only, and has ~60-65 nm rms WFE (consistent with budget predictions). This is for the global WFE of the telescope on-axis. Additional terms for off-axis telescope WFE and SI WFE are modeled separately and added on top of this. Again, see jwst_optical_budgets. For the older OPD files, the average levels of WFE from the telescope itself used in those "predicted" and "requirements" 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

As noted above, these older files accidentally do also include conservative models for wavefront error contributions from optics internal to the science instrument. This is why the models for NIRSpec and MIRI have such higher WFE. We recommend the use of the newer "prelaunch_predicted" OPDs instead. Additional details on the SI-specific wavefront error models are given under each instrument model section below.

How well will any of these models represent the true in-flight performance that will be achieved by the observatory? We'll all learn together in 2022. Stay tuned for WebbPSF 1.1 and beyond.

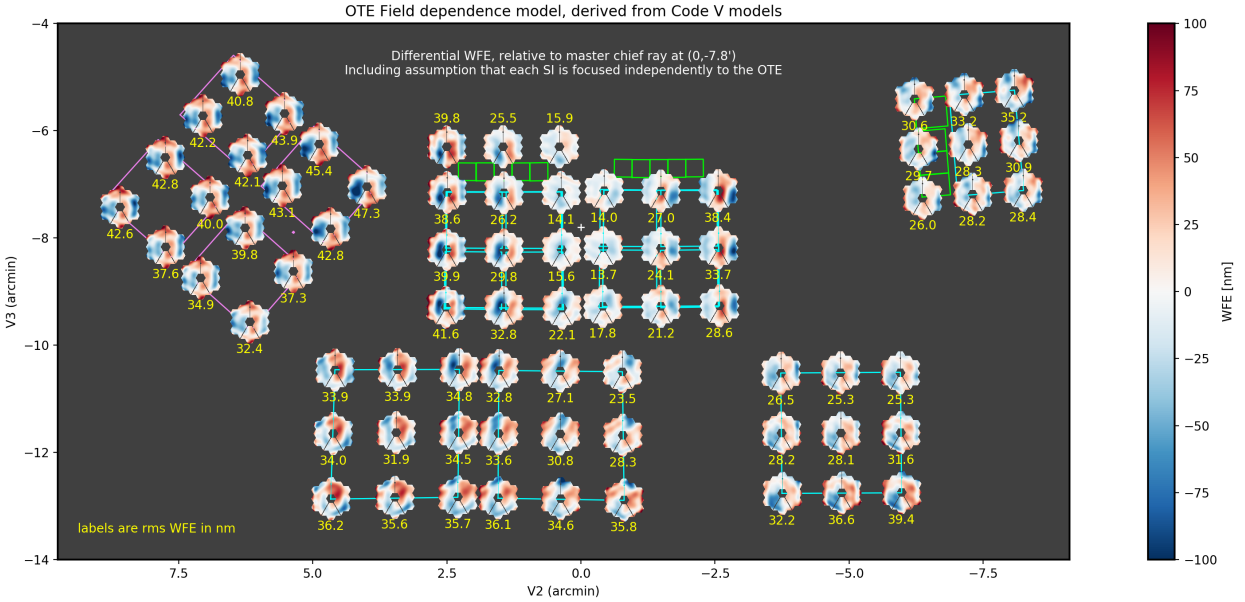
5.2.1 Field Dependent Aberrations

While the OTE is designed to have low aberrations across all of the science instruments, it has small intrinsic aberrations which furthermore vary across the field. This is true even if all mirrors are aligned perfectly, due to design residuals and the as-built mirror surface quality. For the as-built WFE, a particularly significant contributor is the tertiary mirror. Because this is not at a pupil plane, different portions are illuminated for different field points. Surface print-through of manufacturing artifacts into the tertiary mirror surface then results in increased field dependent WFE.

In an effort to capture the contribution of these field-dependent aberrations a polynomial model of the field dependent aberrations was derived, based on the as-built OTE optical model Revision H, which includes measured surface errors of the optical elements. This optical model was used in the CodeV lens design and anaylsis software package to generate

OPD maps capturing the variation of the OTE's aberrations across the fields of each of the science instruments. Each of these OPD maps were fit to a set of Zernike polynomials so that the wavefront was represented by a small number of coefficients, varying at each field point. These variations are captured by fitting these varying Zernike coefficients to a second set of polynomials. Since the fields are generally rectangular, a set of two-dimensional Legendre polynomials were used for this field-fit. Legendres are well-suited for this task because they are orthonormal over a rectangle and JWST's science instrument fields are also rectangular. The resulting model can be used to interpolate the OTE WFE at any field point.

Any field variations in focus will be compensated for as part of focusing each SI. This SI focus optimization is taken into account in the OTE WFE model by the simple expedient of removing the average defocus across each SI's full field of view.



Click to enlarge figures

For the above figure, and all others on this page, click the figure to view it larger and full screen.

5.3 NIRCam

5.3.1 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.

5.3.2 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.

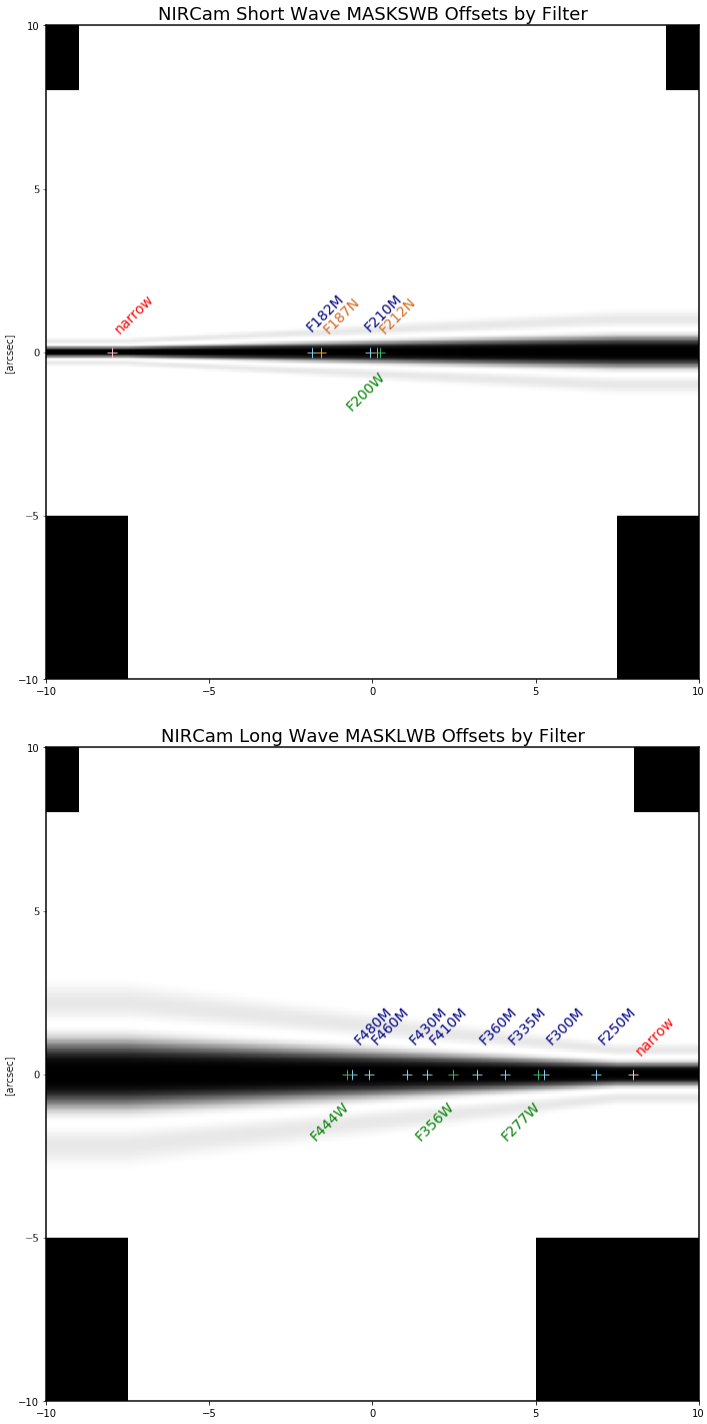
The NIRCam class can automatically switch its `aperturename` attribute when a coronagraphic mask is selected, to select the aperturename for the appropriate coronagraphic subarray. The detector reference pixel location will also update to the center of the coronagraphic subarray. This behavior on image mask or pupil mask selection can be disabled by setting `nircam.auto_aperturename = False`.

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 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.

5.3.3 Weak Lenses for Wavefront Sensing

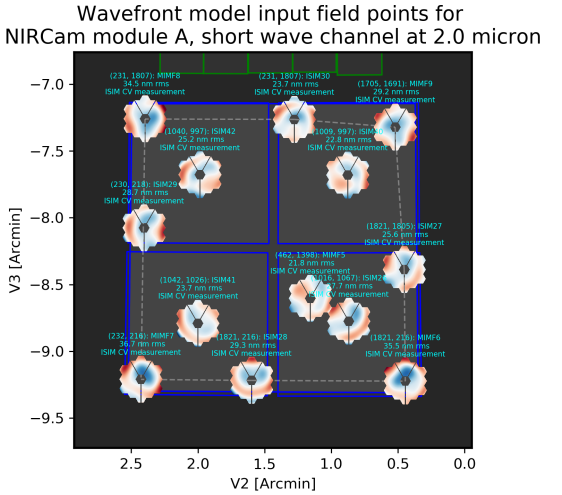
WebbPSF includes models for the three weak lenses used for wavefront sensing, including the pairs of lenses that can be used together simultaneously.

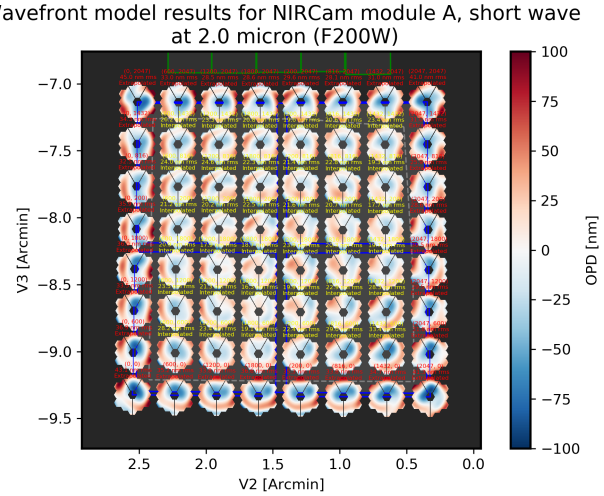
The convention is such that the "negative" 8 waves lens is concave, the "positive" two lenses are convex. Thus positive weak lenses move best focus in front of the detector, or equivalently the electric field imaged on the detector becomes behind or beyond best focus. Negative weak lenses move best focus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

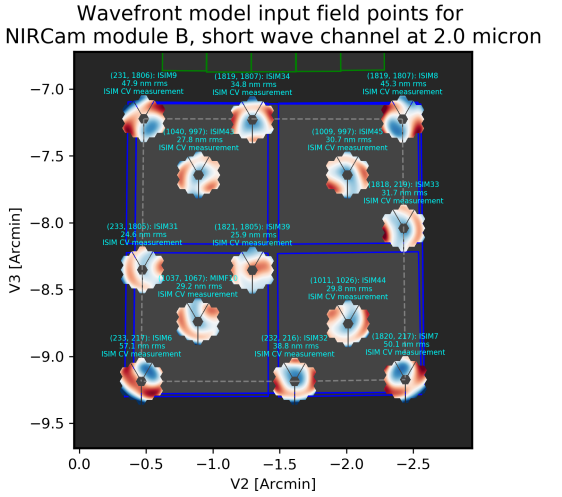
Note that the weak lenses are in the short wave channel only; webbpsf won't stop you from simulating a LW image with a weak lens, but that's not a real configuration that can be acheived with NIRCam.

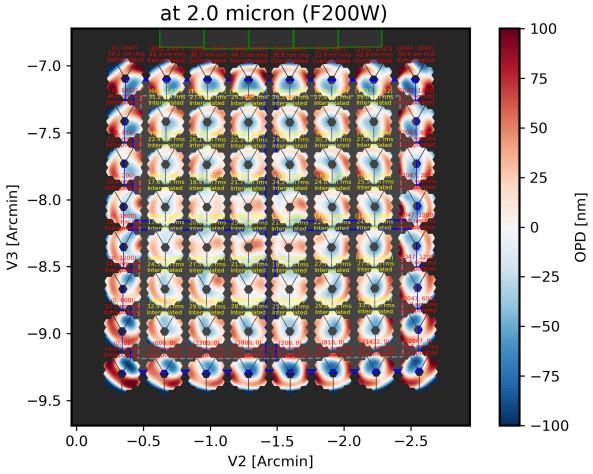
5.3.4 SI WFE

SI internal WFE measurements are from ISIM CV3 testing (See JWST-RPT-032131 by David Aronstein et al.) 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. For field-points outside of the measurement bounds, WebbPSF performs an extrapolation routine.









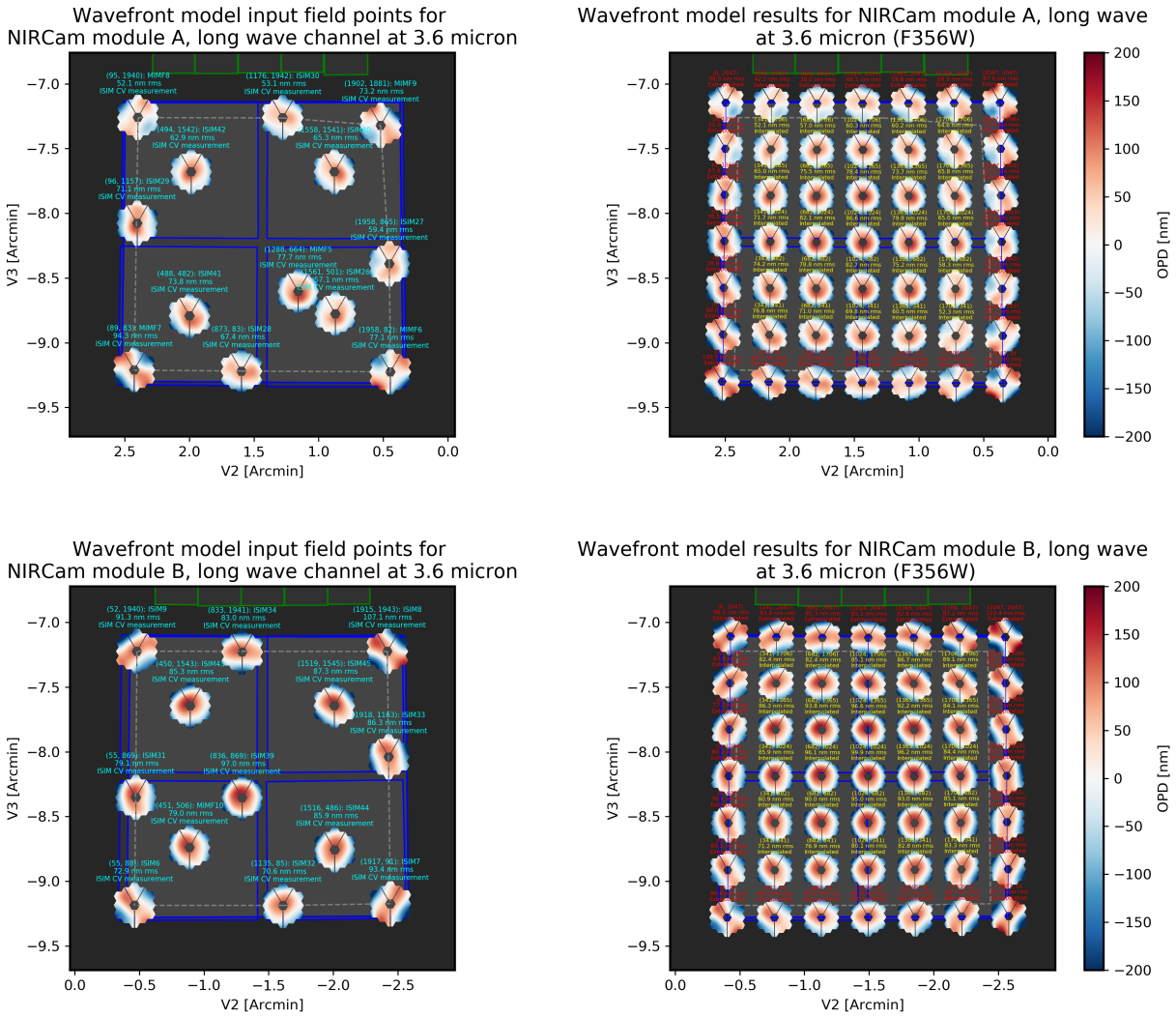


Fig. 1: Instrument WFE models for NIRCam. Click for full size.

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. While SI WFE for imaging mode were measured directly from the instrument during ISIM CV3, the coronagraphic WFE maps were built based on the NIRCam Zemax optical model. This model was first validated in imaging mode, and then the appropriate optical elements were inserted to produce the coronagraphic configuration. In this case, both modules were assumed have the exact same (albeit, mirrored) field-dependent WFE maps. Note, this substantial WFE occurs physically *after* the coronagraphic focal plane spots in NIRCam, and is modeled as such in WebbPSF.

5.3.5 Wavelength-Dependent Focus Variations

NIRCam's wavelength-dependent defocus was measured during ISIM CV2 at a given field point (See JWST-RPT-029985 by Randal Telfer). Overall, the measurements are consistent with predictions from the nominal optical model. The departure of the data from the model curve has been determined to be from residual power in individual filters. In particular, the F323N filter has a significant extra defocus; WebbPSF includes this measured defocus if the selected filter is F323N.

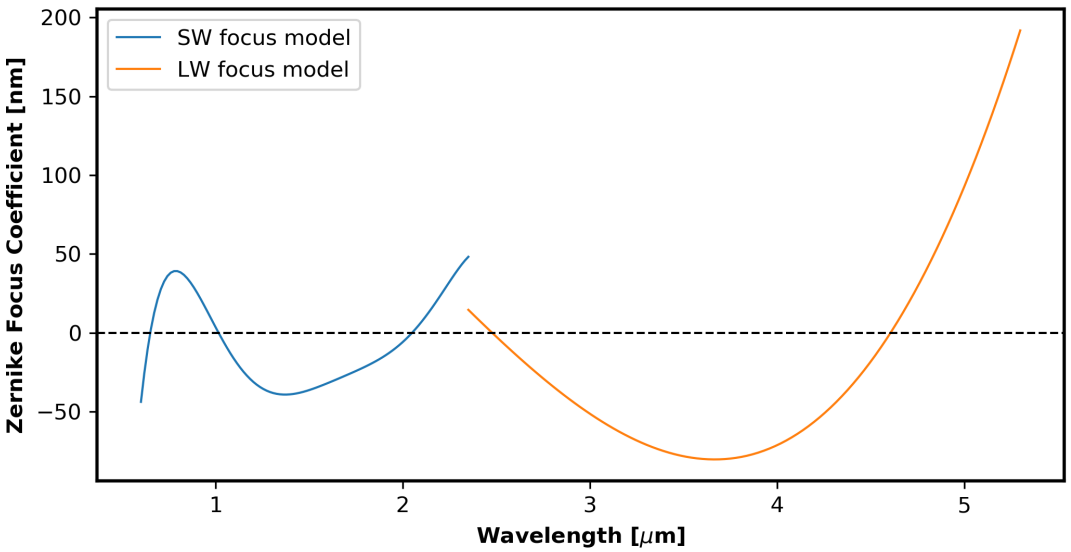


Fig. 2: Instrument focus models for NIRCam. Click for full size.

All SI WFE maps were derived from measurements with the F212N and F323N filters. WebbPSF utilizes polynomial fits to the nominal focus model to derive focus offset values relative to these narrowband filters for a given wavelength. The derived delta focus is then translated to a Zernike focus image, which is subsequently applied to the instrument OPD map.

5.4 NIRSpec

5.4.1 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.

5.4.2 SI WFE

SI internal WFE measurements are from ISIM CV3 testing (See JWST-RPT-032131 by David Aronstein et al.).

The ISIM CV3 data on their own do not indicate how the sources of WFE are distributed within the NIRSpec optical train. For simulation purposes here, the SI WFE measurements are allocated as 1/3 in the foreoptics, prior to the MSA image plane, and 2/3 in the spectrograph optics, after the MSA image plane. This follows a recommendation from Maurice Te Plate of the NIRSpec team, based on metrology and testing of the NIRSpec flight model optics.

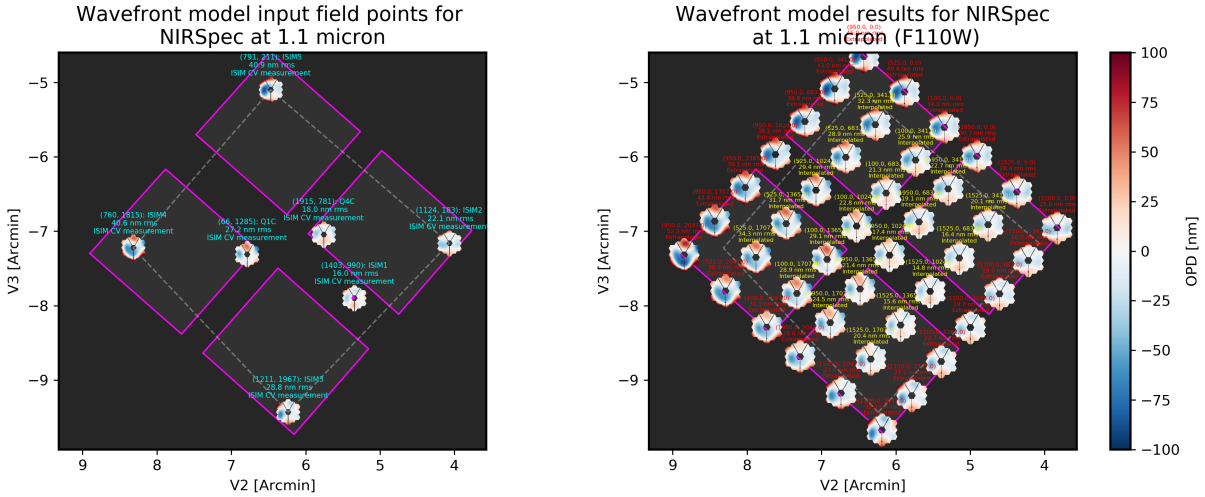


Fig. 3: Instrument WFE models for NIRSpec. Click for full size.

5.5 NIRISS

5.5.1 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`).

AMI mask geometry is as provided to the WebbPSF team by Anand Sivaramakrishnan. To match the orientation of the mask as installed in the flight hardware, the simulated mask model was flipped in X coordinates as of the spring 2019 version of WebbPSF; thanks to Kevin Volk and Deepashri Thatte for determining this was necessary to match the test data.

5.5.2 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.

5.5.3 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.

5.5.4 SI WFE

SI internal WFE measurements are from ISIM CV3 testing (See JWST-RPT-032131 by David Aronstein et al.).

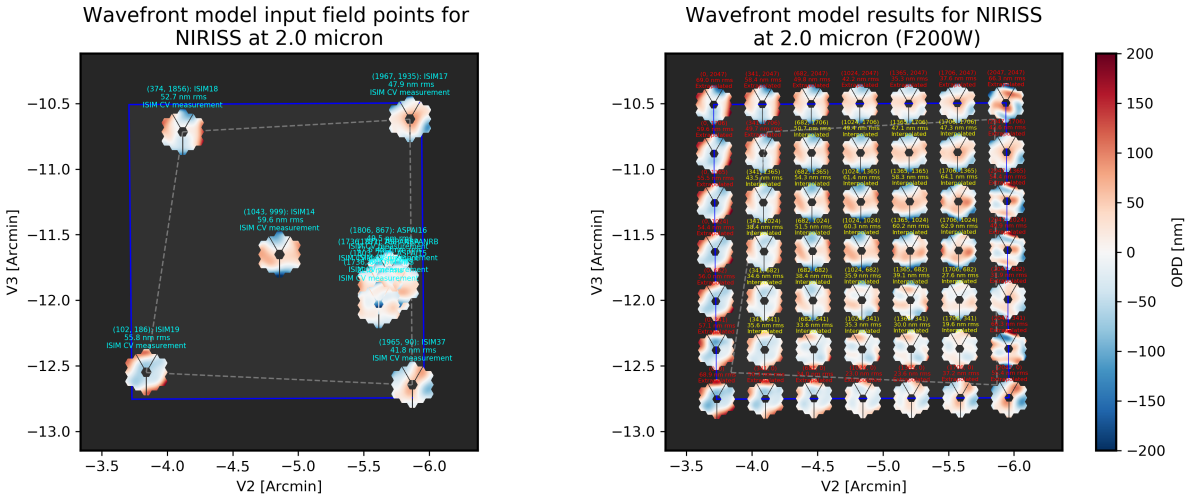


Fig. 4: Instrument WFE models for NIRISS. Click for full size.

5.6 MIRI

5.6.1 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.

5.6.2 MIRI detector cross artifact

The MIRI imager's Si:As IBC detector exhibits a so-called "cross artifact", particularly at short wavelengths (5-8 microns), due to internal diffraction of photons within the detector subtrate itself. See Gaspar et al. 2021 for details. WebbPSF implements a simplified model for this effect, following the approach described by Glasse et al. in MIRI technical report MIRI-TN-00076-ATC_Imager_PSF_Issue_4.pdf. The model coefficients have been adjusted to better match the cross artifact amplitudes from WebbPSF to the MIRI Calibration Data Product reference PSFs.

Note: Where to find Results from the Cross Artifact Model

The cross artifact is added alongside the geometric distortion step, after the optical propagation. The results are stored in FITS extensions 2 and 3 (ext names OVERDIST and DET_DIST for oversampled and detector sampled, respectively *not* in the default 0th extension which is the raw oversampled PSF. E.g.:

```
miri = webbpsf.MIRI()
psf = miri.calc_psf()
webbpsf.display_psf(psf, ext=3)
result = psf['DET_DIST'].data   # This is the PSF with the cross artifact model included
```

5.6.3 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`.

The MIRI class can automatically switch its `aperturename` attribute when a coronagraphic mask is selected, to select the aperturename for the appropriate coronagraphic subarray. The detector reference pixel location will also update to the center of the coronagraphic subarray. This behavior on image mask selection can be disabled by setting `miri.auto_aperturename = False`.

5.6.4 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'`. That said, the LRS simulations have not been extensively tested yet; feedback is appreciated about any issues encountered.

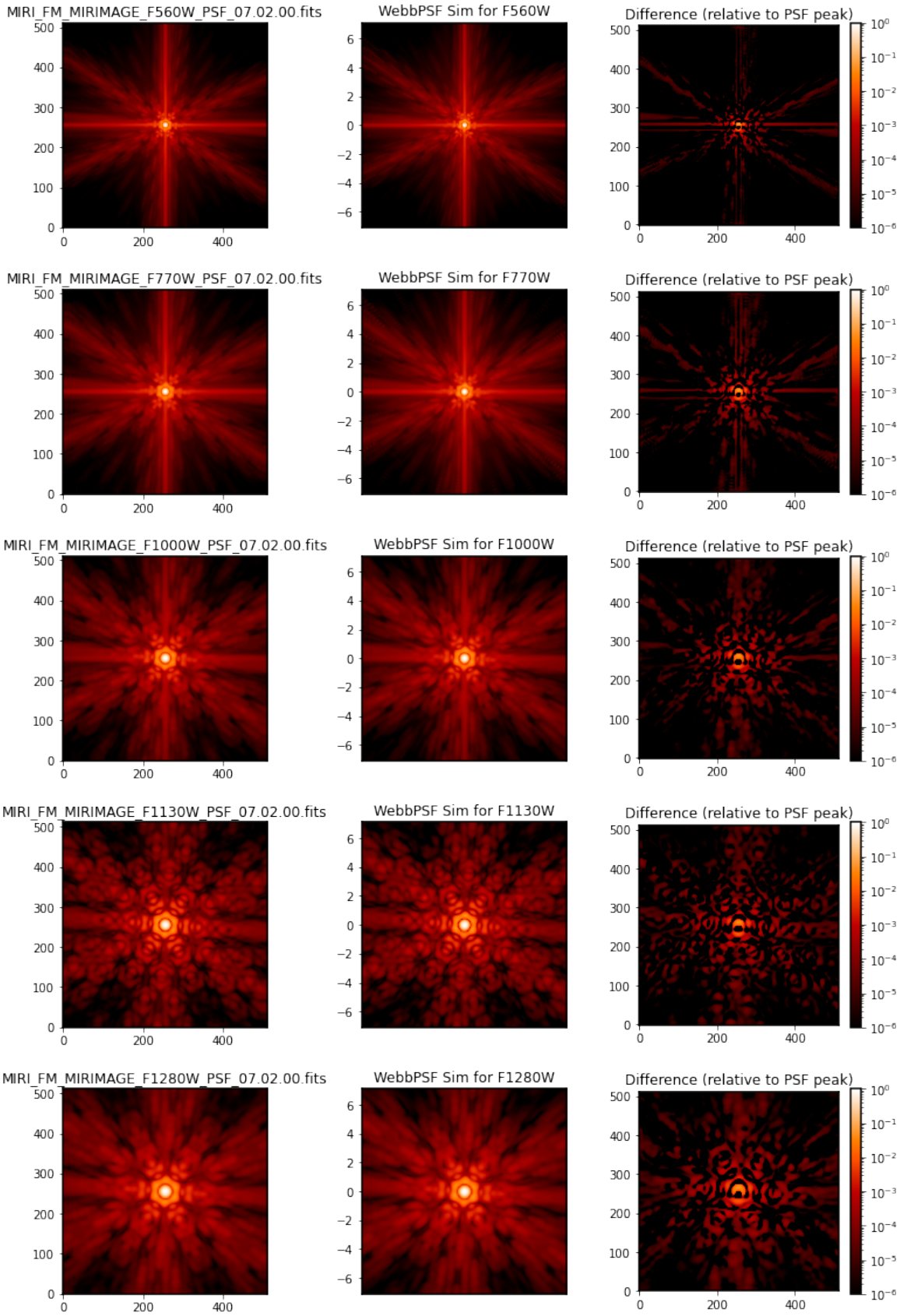


Fig. 5: Comparison of models for the MIRI detector cross artifact. Click for full size. Shown are the MIRI Calibration Data Product PSFs (Left), the WebbPSF results (Center) and their difference. The cross artifact is negligible at wavelengths beyond ~12 microns.

5.6.5 SI WFE

SI internal WFE measurements are from ISIM CV3 testing (See JWST-RPT-032131 by David Aronstein et al.).

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

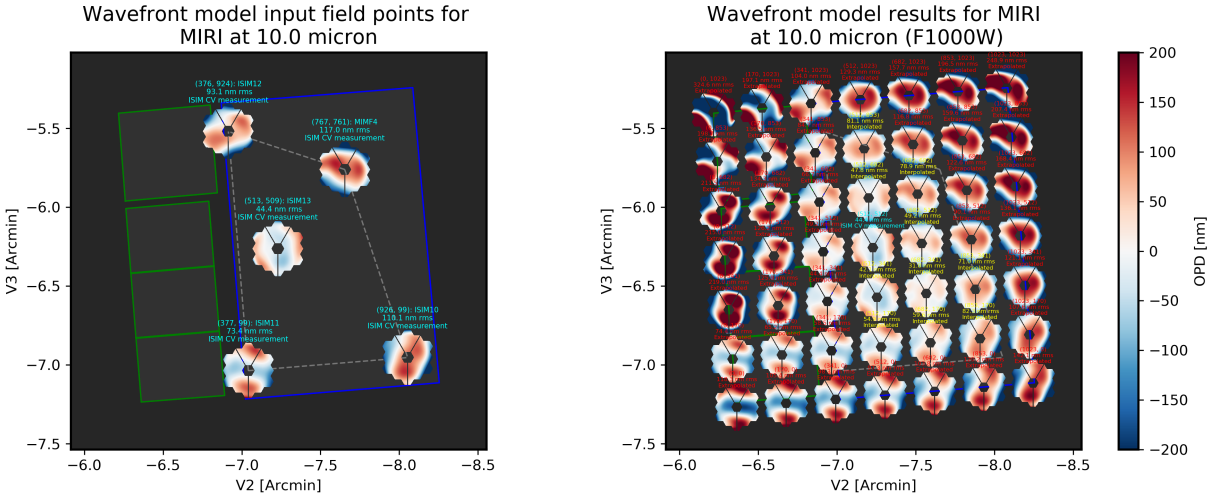


Fig. 6: Instrument WFE models for MIRI. Click for full size.

5.6.6 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.

5.7 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'.

5.7.1 SI WFE

SI internal WFE measurements are from ISIM CV3 testing (See JWST-RPT-032131 by David Aronstein et al.).

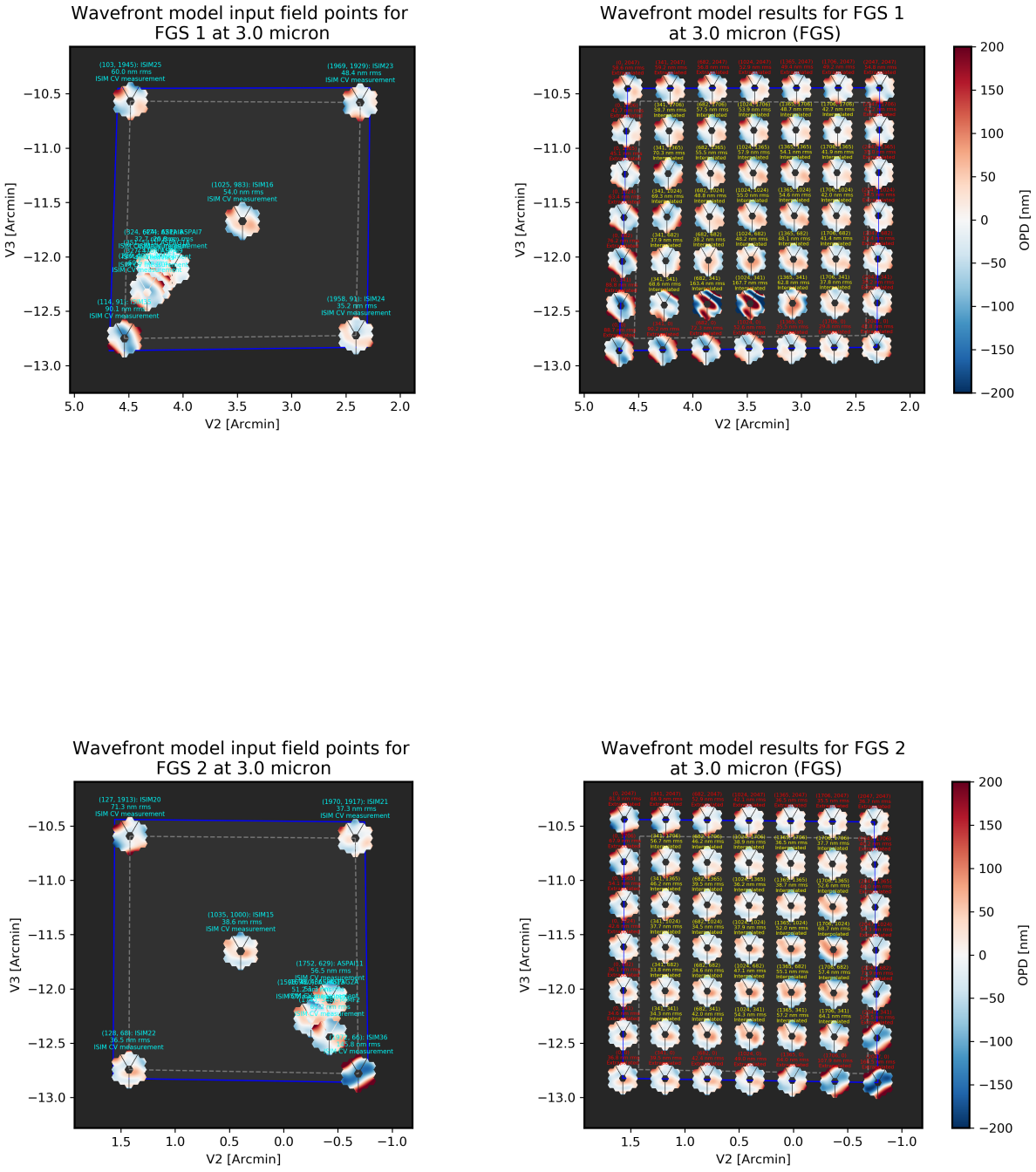


Fig. 7: Instrument WFE models for FGS. Click for full size.

CHAPTER

SIX

ROMAN INSTRUMENT MODEL DETAILS

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

6.1 Wide Field Instrument (WFI)

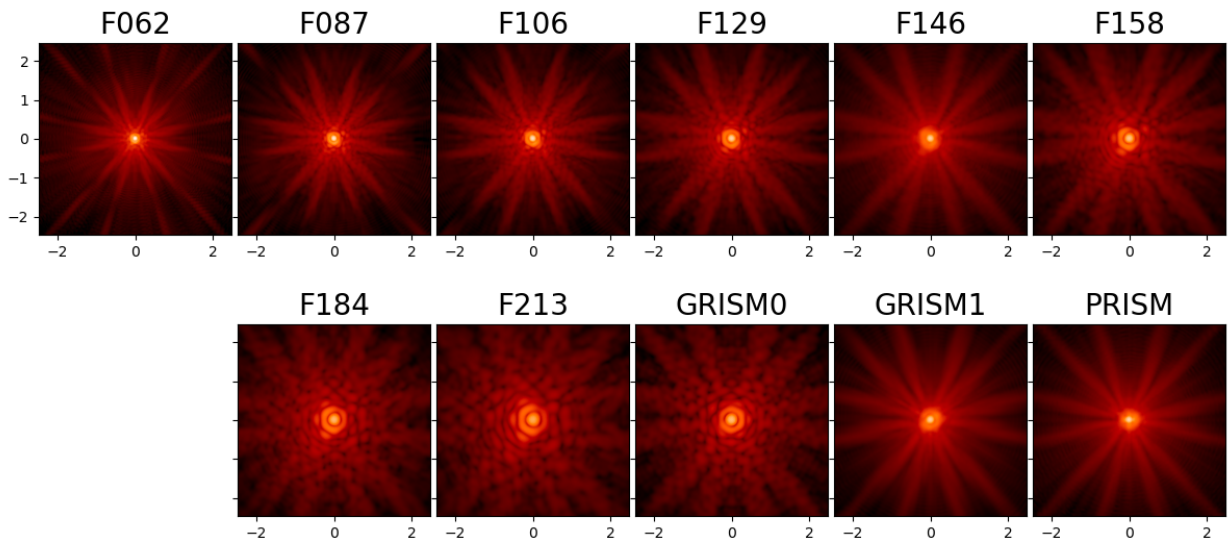


Fig. 1: Sample PSFs for the filters in the Roman WFI. Angular scale in arcseconds, log-scaled intensity.

The WFI model is based on the Cycle 9 instrument reference information from the Roman team at Goddard Space Flight Center (GSFC). The reported jitter for the Roman observatory is 0.012 arcsec per axis, per GSFC.

To work with the WFI model, import and instantiate it just like any of the JWST instruments:

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

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

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 Roman team's Cycle 9 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.

Note: *Tutorial notebook for Roman*

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

Caution: Note that unlike most JWST modes, Roman 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.)

6.1.1 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, photosensitive regions of the Wide Field Instrument's detectors are slightly smaller than their 4096 by 4096 pixel dimensions because the outermost four rows and columns are reference pixels that are not sensitive to light. To change the position at which to calculate a PSF, assign an (X, Y) tuple:

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

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 18 unique wavelengths providing coverage from 0.76 μm to 2.3 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. WebbPSF will approximate the aberrations for an out of range detector position by using the nearest field point.

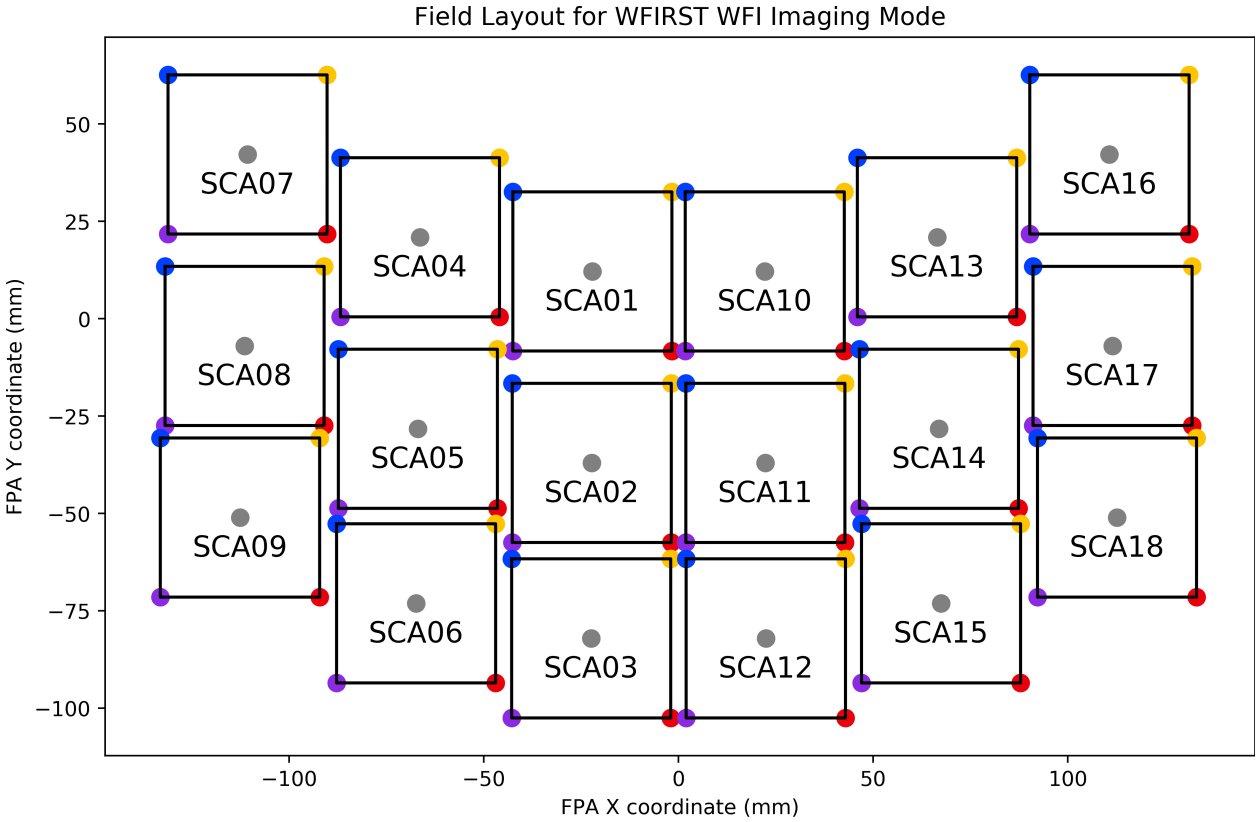


Fig. 2: The Wide Field Instrument's field of view, as projected on the sky.

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
```

6.1.2 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 = roman.WFI()
>>> wfi.filter = 'F129'
>>> 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 - F129')
>>> webbpsf.display_psf(psf_sca17, ax=ax_sca17, imagecrop=2.0,
                        title='WFI SCA17, top right - F129')
>>> webbpsf.display_psf_difference(psf_sca09, psf_sca17, ax=ax_diff,
                                   vmax=5e-3, title='SCA09 - SCA17', imagecrop=2.0)
```

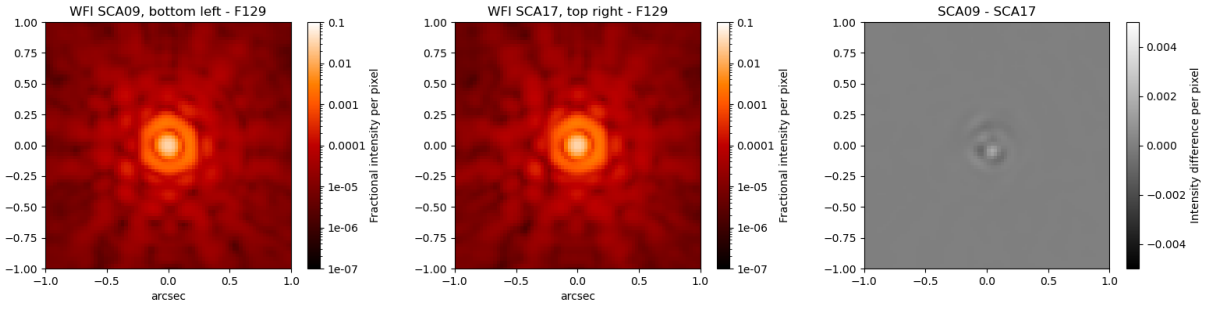


Fig. 3: This figure shows oversampled PSFs in the F129 filter at two different field points, and the intensity difference image between the two.

6.1.3 Pupil variation and pupil masks in the WFI model

As before, the Cycle 9 reference data release from the Goddard Space Flight Center features field-dependent pupil images for the WFI. However, this cycle's pupil images are categorized in a manner that diverges from that of previous cycles.

A plurality of the filters – F062, F087, F106, F129, and F158 – now use the "Skinny" mask, which is exclusive to the imaging mode. The remaining imaging filters, F184 and the new F213, share F184's "Wide" mask. Both the undispersed zeroth order and dispersed first order of the grism mode share the eponymous "Grism" mask. Finally, though the prism mode operates sans obstruction, its maskless arrangement is termed the "Prism" mask for the sake of consistency.

Please note that these pupil mask category names are not fully backward compatible with those from previous versions of WebbPSF. For example, the `pupil_mask_list` of `['AUTO', 'FULL_MASK', 'RIM_MASK', 'COLD_PUPIL', 'UNMASKED']` in versions 0.9.X is now obsolete.

Fig. 4: Pupil masks at different field points.

The pupil and pupil mask are dynamically selected as needed whenever the detector or filter is changed. To override this behavior for either attribute, see `WFI.lock_pupil()` and `WFI.lock_pupil_mask()`. The following pupils are available:

Pupil Mask	pupil_mask setting
Skinny Mask	'SKINNY' (formerly 'RIM_MASK', 'UNMASKED')
Wide Mask	'WIDE' (formerly 'FULL_MASK', 'COLD_PUPIL')
Grism Mask	'GRISM'
Prism Mask	'PRISM' (formerly 'RIM_MASK', 'UNMASKED')

6.2 Coronagraph Instrument

We have begun developing a Coronagraph Instrument simulation module. The goal is to provide an open source modeling package for the Coronagraph Instrument 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 Coronagraph 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: The Coronagraph model has not been actively updated or developed since circa 2017. It does not well represent the current PDR-level state of the instrument. There are plans to refresh this model. Interested users should contact Ewan Douglas.

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 RomanCoronagraph class is available in this Jupyter Notebook. Here we briefly summarize the key points, but see that for more detail.

The RomanCoronagraph 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

```
>>> cor = roman.RomanCoronagraph()
>>> cor.mode = "CHARSPC_F770"
```

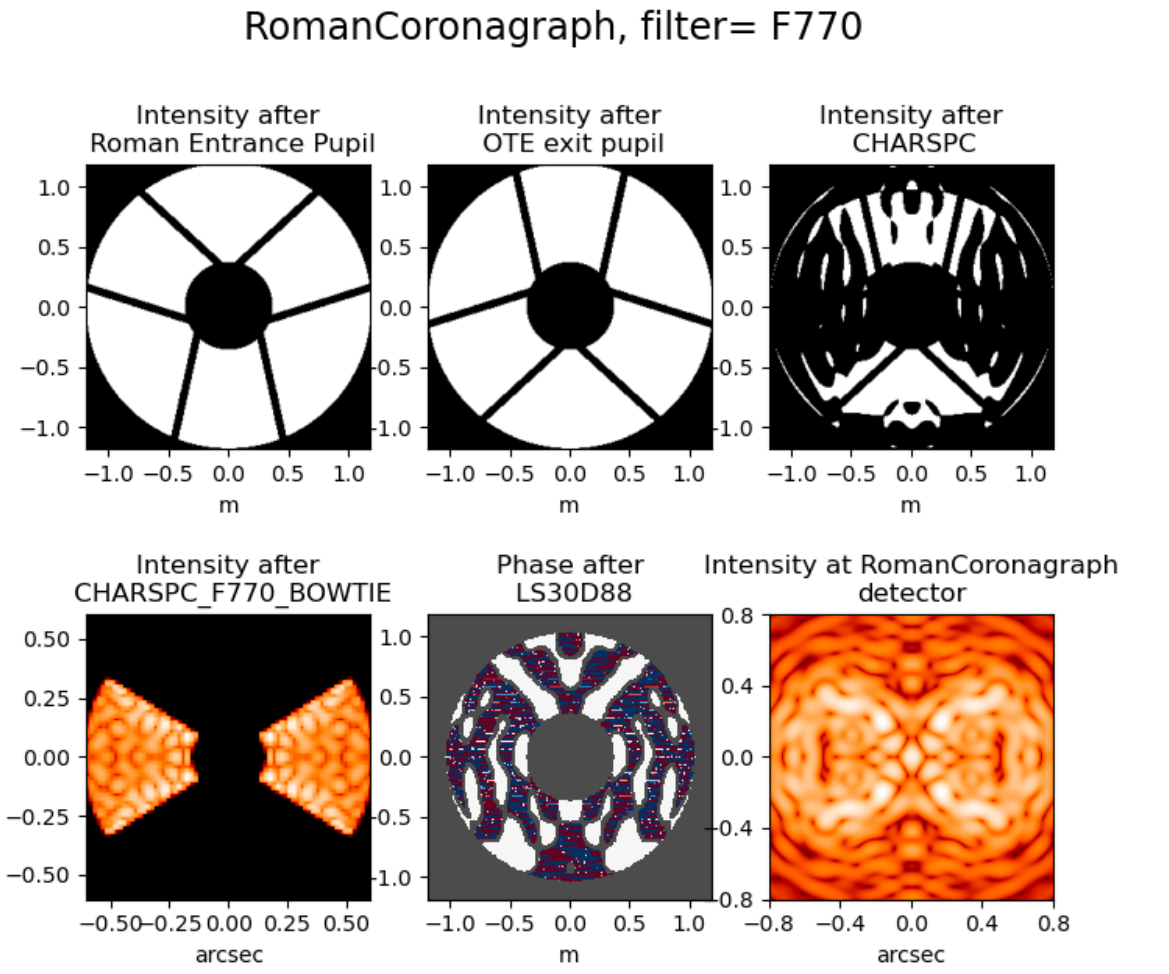
is equivalent to:

```
>>> cor.camera = 'IFS'
>>> cor.filter = 'F770'
>>> cor.apodizer = 'CHARSPC'
>>> cor.fpm = 'CHARSPC_F770_BOWTIE'
>>> cor.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 = cor.calc_psf(nlambda=1, fov_arcsec=1.6, display=True)
```



Calculation with 1 wavelengths (0.77 - 0.77 um)

CHAPTER

SEVEN

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.

7.1 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)
```

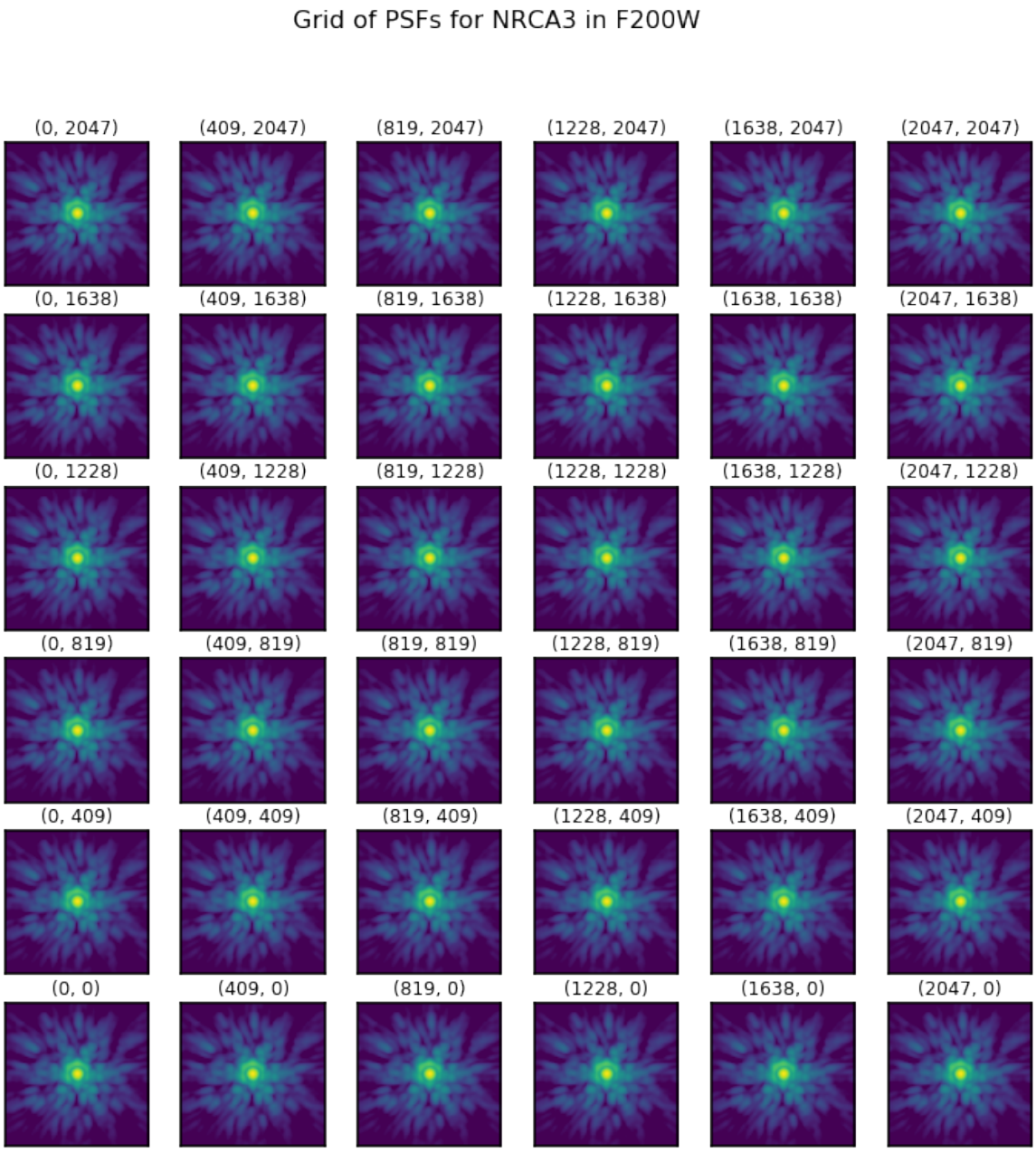


Fig. 1: An example of grid calculated across the NRCA3 detector in NIRCam. These PSFs are all very similar.

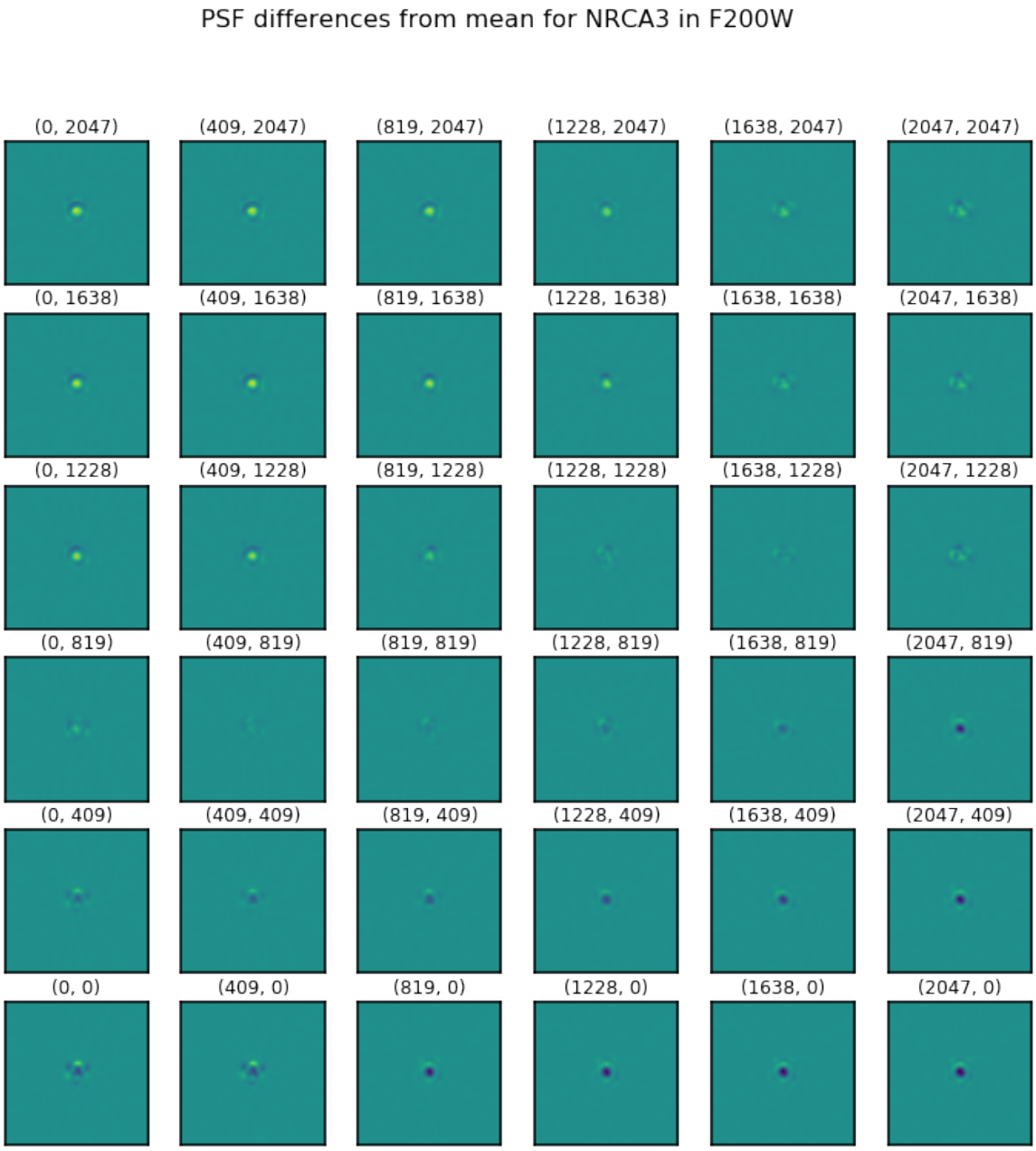


Fig. 2: By subtracting off the average PSF, the subtle differences from point to point become clear. The PSF is sharpest in the upper left corner of this detector.

CHAPTER

EIGHT

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:

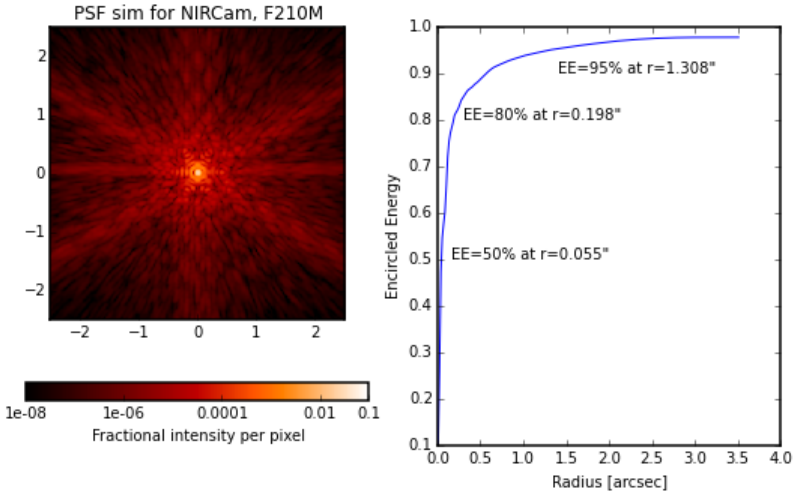
- *Typical Usage Cases*
- *Spectroscopic PSFs, Slit and Slitless*
- *Coronagraphy and Complications*

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.

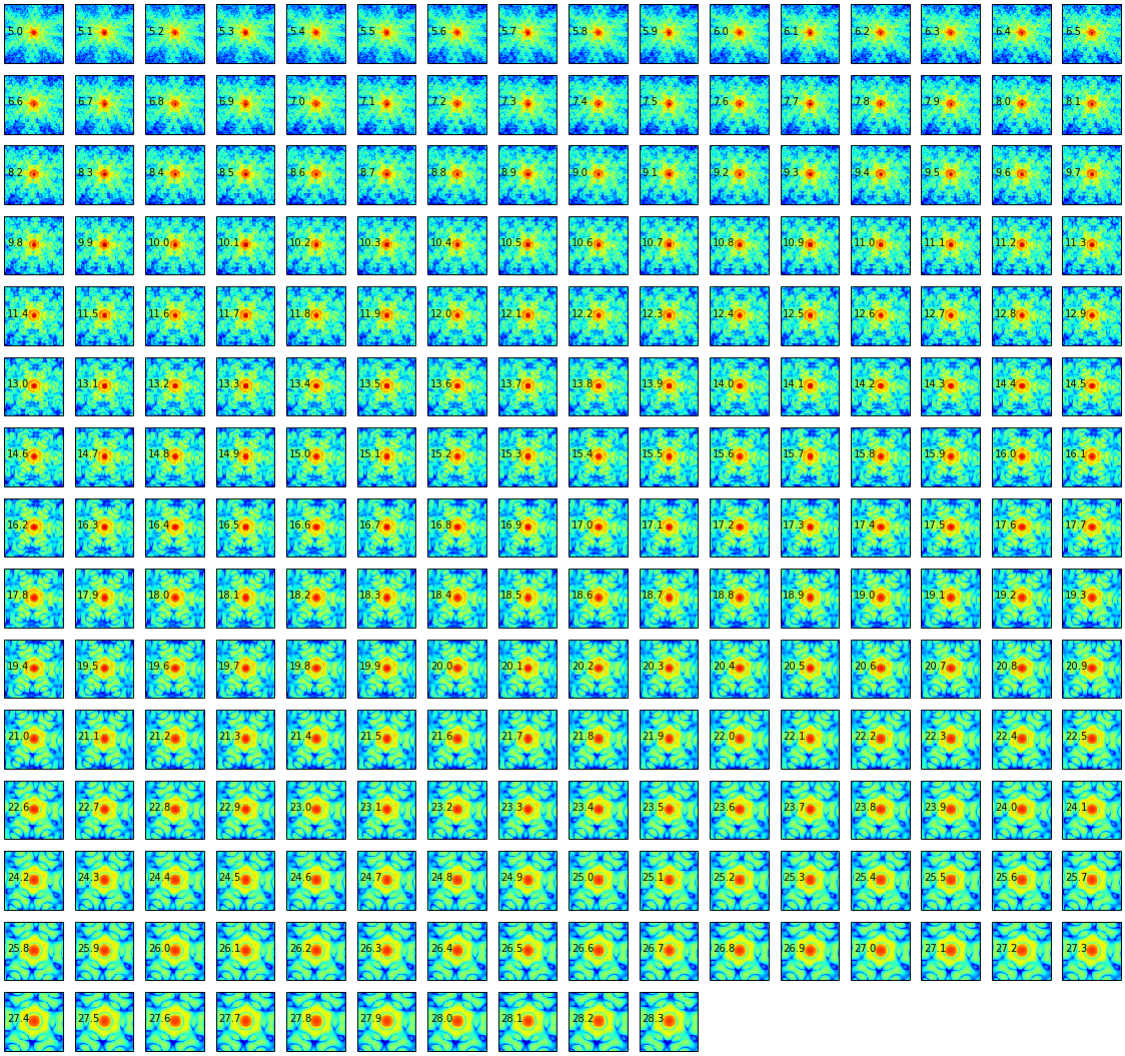
```
m = webbpsf.MIRI()
m.pupilopd = 'OPD_RevW_ote_for_MIRI_requirements.fits.gz'       # select an OPD
                                          # looks inside $WEBBPSF_DATA/MIRI/OPD by␣
↪default
                                           # or you can specify a full path name.
m.options['parity'] = 'odd'               # 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
#waves = np.linspace(5.0, 28.3, 20)*1e-6     # iterate over wavelengths in meters
```

(continues on next page)

(continued from previous page)

```
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:

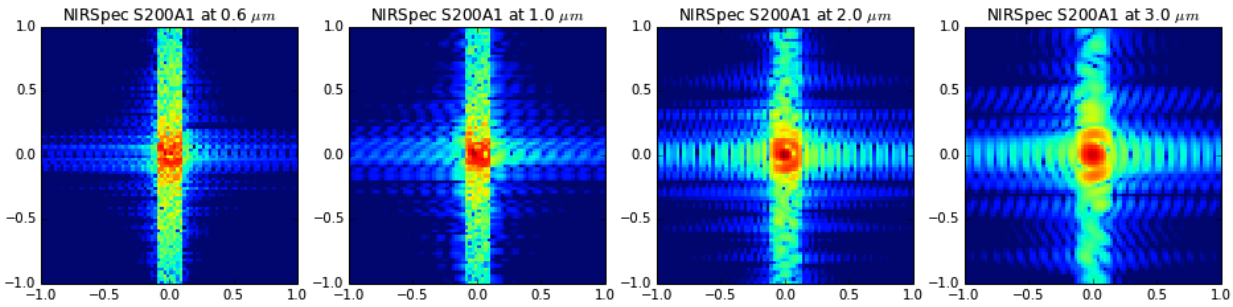


8.2 Spectroscopic PSFs, Slit and Slitless

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.

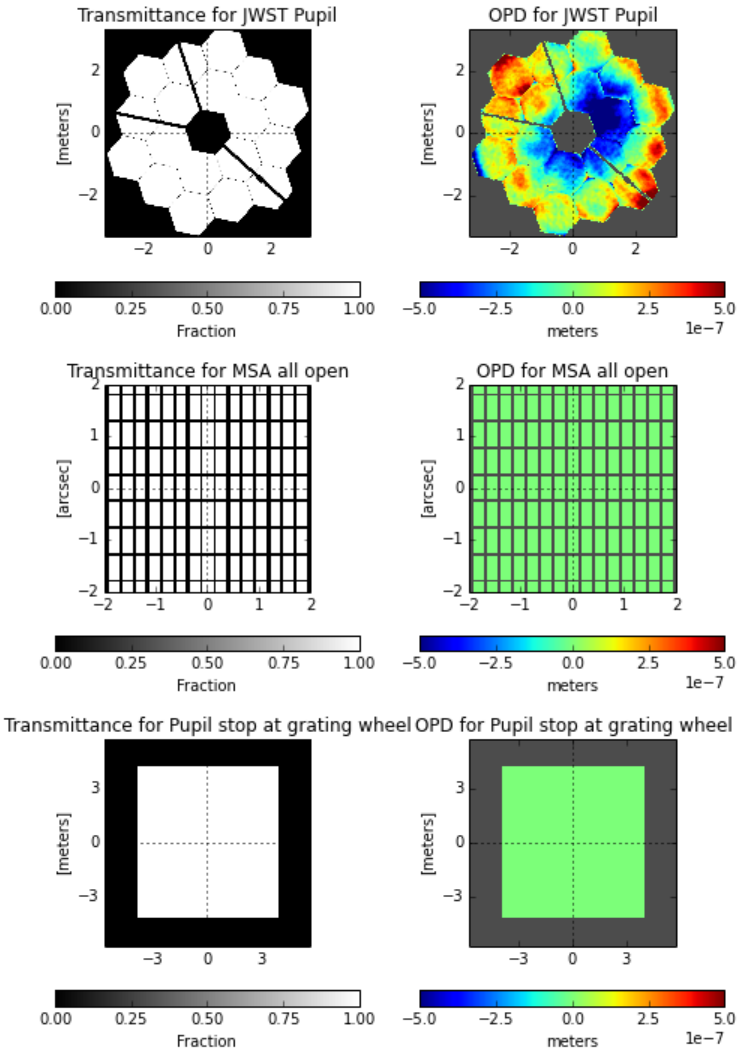
8.2.1 NIRSpec fixed slits

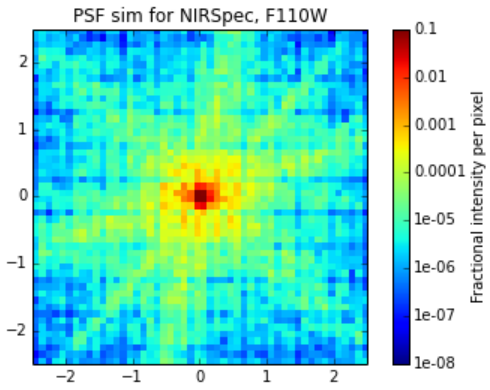
```
plt.figure(figsize=(8, 12))
nspec = webbpsf.NIRSpec()
nspec.image_mask = 'S200A1' # 0.2 arcsec slit

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

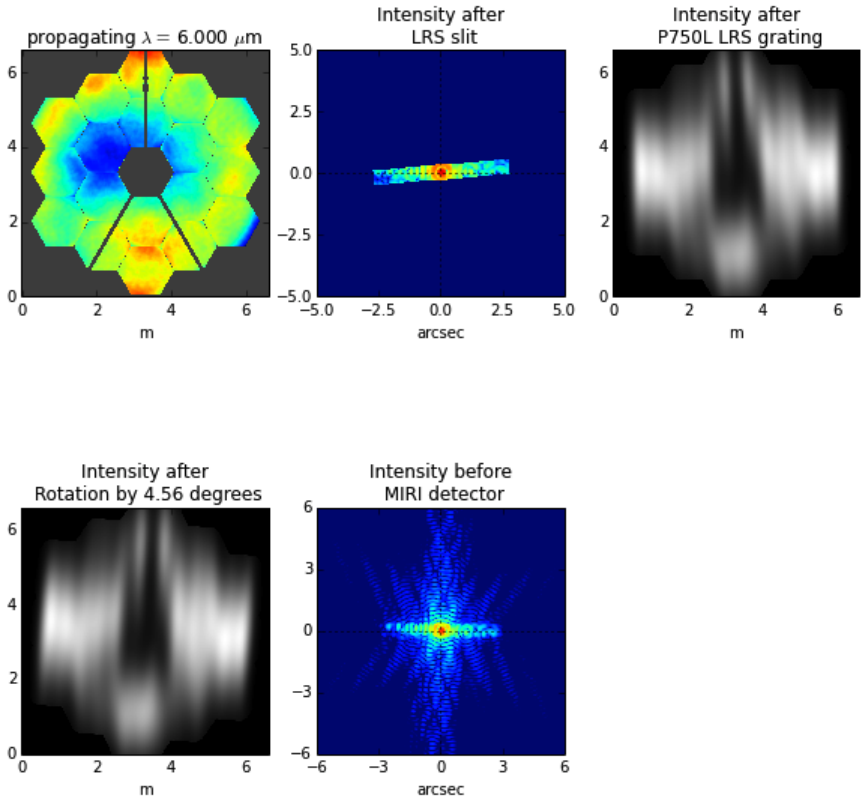
```
plt.figure(figsize=(8, 12))
ns = webbpsf.NIRSpec()
ns.image_mask='MSA all open'
ns.display()
plt.savefig('example_nirspec_msa_optics.png')
msapsf = ns.calc_psf(monochromatic=2e-6, oversample=8, rebin=True)
webbpsf.display_psf(msapsf, ext='DET_SAMP')
```


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

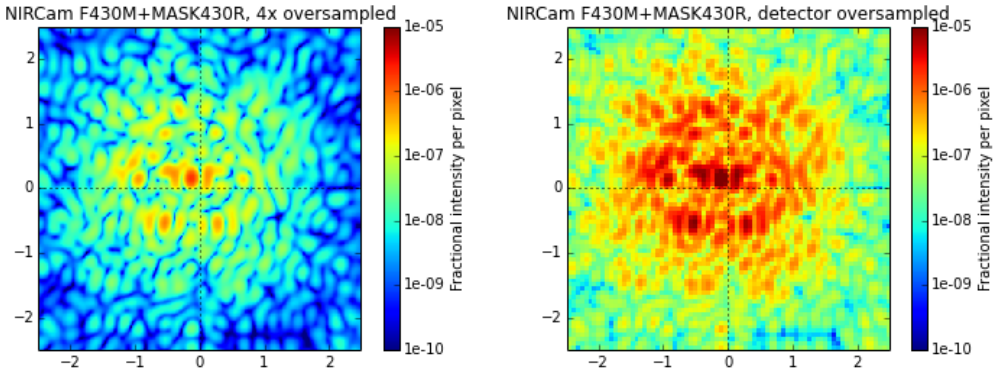
```
miri = webbpsf.MIRI()
miri.image_mask = 'LRS slit'
miri.pupil_mask = 'P750L'
psf = miri.calc_psf(monochromatic=6.0e-6, display=True)
```



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)
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),
```

(continues on next page)

(continued from previous page)

```
            ('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


        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␣
```

(continues on next page)

(continued from previous page)

```
↪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)



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')
```

(continues on next page)

(continued from previous page)

```
    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

    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_')
```

(continues on next page)

(continued from previous page)

```
            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

        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)
```

(continues on next page)

(continued from previous page)

```
        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")

        # plot shifted pupil PSF - detector sampled
```

(continues on next page)

(continued from previous page)

```
        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

NINE

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.

CHAPTER

TEN

ADVANCED USAGE

10.1 Detailed API Reference

10.1.1 webbpsf Package

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/

Functions

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)	Set up a WebbPSF instrument instance to have a modifiable OTE wavefront error OPD via an OTE linear optical model (LOM).
instrument(name)	This is just a convenience function, allowing one to access instrument objects based on a string.
measure_centroid([HDUlist_or_filename, ext, ...])	Measure the center of an image via center-of-mass
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 synphot Spectrum object from a user-friendly spectral type string.
system_diagnostic()	return various helpful/informative information about the current system.

enable_adjustable_ote

webbpsf.**enable_adjustable_ote**(*instr*)

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

Parameters

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

Returns

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

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')
```

Parameters

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

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.

Parameters

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

Returns

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.

Parameters

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.

Parameters

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)

Examples

```
>>> 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

Parameters

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…

Classes

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.
RomanCoronagraph([mode, pixelscale, ...])	Roman Coronagraph Instrument
SpaceTelescopeInstrument([name, pixelscale])	A generic Space Telescope Instrument class.
UnsupportedPythonError	
WFI()	WFI represents the Roman mission's Wide Field Imager.

Conf

class webbpsf.**Conf**

Bases: 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: *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.

The detectors are named as FGS1, FGS2 but may synonymously also be referred to as GUIDER1, GUIDER2 for compatibility with DMS convention

Attributes Summary

detector	Detector selected for simulated PSF

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.

JWInstrument

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

Bases: *SpaceTelescopeInstrument*

Superclass for all JWST instruments

Attributes Summary

aperturename	SIAF aperture name for detector pixel to sky coords transformations
pupilopd	Filename *or* fits.HDUList for JWST pupil OPD.
telescope	

Methods Summary

calc_psf([outfile, source, nlambda, ...])	Compute a PSF.
get_opd_file_full_path([opdfilename])	Return full path to the named OPD file.
get_optical_system([fft_oversample, ...])	Return an OpticalSystem instance corresponding to the instrument as currently configured.
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.
set_position_from_aperture_name(aperture_name)	Set the simulated center point of the array based on a named SIAF aperture.
visualize_wfe_budget([slew_delta_time, ...])	Display a visual WFE budget showing the various terms that sum into the overall WFE for a given instrument

Attributes Documentation

aperturename

SIAF aperture name for detector pixel to sky coords transformations

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.

Parameters

source
[synphot.spectrum.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.

Returns

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.

Notes

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

get_opd_file_full_path(*opdfilename=None*)

Return full path to the named OPD file.

The OPD may be:

- a local or absolute path,
- or relative implicitly within an SI directory, e.g. $WEBBPSF_PATH/NIRCam/OPD
- or relative implicitly within $WEBBPSF_PATH

This function handles filling in the implicit path in the latter cases.

get_optical_system(*fft_oversample=2*, *detector_oversample=None*, *fov_arcsec=2*, *fov_pixels=None*, *options=None*)

Return an OpticalSystem instance corresponding to the instrument as currently configured.

When creating such an OpticalSystem, you must specify the parameters needed to define the desired sampling, specifically the oversampling and field of view.

Parameters

fft_oversample
[int] Oversampling factor for intermediate plane calculations. Default is 2

detector_oversample: int, optional
By default the detector oversampling is equal to the intermediate calculation oversampling. If you wish to use a different value for the detector, set this parameter. Note that if you just want images at detector pixel resolution you will achieve higher fidelity by still using some oversampling (i.e. *not* setting `oversample_detector=1`) and instead rebinning down the oversampled data.

fov_pixels
[float] Field of view in pixels. Overrides fov_arcsec if both set.

fov_arcsec
[float] Field of view, in arcseconds. Default is 2

Returns

osys
[poppy.OpticalSystem] an optical system instance representing the desired configuration.

interpolate_was_opd(*array*, *newdim*)

Interpolates an input 2D array to any given size.

Parameters

array: float
input array to interpolate

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

Returns

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.

Parameters

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'.

Returns

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

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.

visualize_wfe_budget(*slew_delta_time=<Quantity 14. d>*, *slew_case='EOL'*, *ptt_only=False*, *verbose=True*)

Display a visual WFE budget showing the various terms that sum into the overall WFE for a given instrument

Compares a WebbPSF instrument instance with the JWST optical budget for that instrument

Parameters

inst
[webbpsf.JWInstrument] A JWST instrument instance

slew_delta_time
[astropy.Quantity time] Time duration for thermal slew model

slew_case
[basestring] 'BOL' or 'EOL' for beginning of life or end of life thermal slew model. EOL is about 3x higher amplitude

ptt_only
[bool] When decomposing wavefront into controllable modes, use a PTT-only basis? The default is to use all controllable pose modes. (This is mostly a leftover debug option at this point, not likely useful in general)

verbose
[bool] Be more verbose

MIRI

class webbpsf.**MIRI**

Bases: *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).

Special Options:

The 'coron_shift_x' and 'coron_shift_y' options offset a coronagraphic mask in order to produce PSFs centered in the output image, rather than offsetting the PSF. This is useful for direct PSF convolutions. Values are in arcsec. ` miri.options['coron_shift_x'] = 3 # Shifts mask 3" to right; or source 3 " to left. `

Attributes Summary

filter	Currently selected filter name (e.g.

Attributes Documentation

filter

Currently selected filter name (e.g. F200W)

NIRCam

class webbpsf.**NIRCam**

Bases: *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.

Similarly, SIAF aperture names are automatically chosen based on detector, filter, image mask, and pupil mask settings. The auto-selection can be disabled by setting auto_aperturname = False. SIAF aperture information is mainly used for coordinate transformations between detector science pixels and telescope V2/V3.

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) `

Similarly, the 'coron_shift_x' and 'coron_shift_y' options will offset the mask in order to produce PSFs centered in the output image, rather than offsetting the PSF. This is useful for direct PSF convolutions of an image. Values are in arcsec. These options move the mask in the opposite sense

as nc.options['bar_offset']. ` nc.options['coron_shift_x'] = 3 # Shifts mask 3" to right, equivalent to source 3" to left. `

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	
aperturename	SIAF aperture name for detector pixel to sky coords transformations
channel	
detector	Detector selected for simulated PSF
filter	Currently selected filter name (e.g.
image_mask	Currently selected image plane mask, or None for direct imaging
module	
pupil_mask	Currently selected Lyot pupil mask, or None for direct imaging

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

aperturename
SIAF aperture name for detector pixel to sky coords transformations

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)

image_mask
Currently selected image plane mask, or None for direct imaging

module

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

NIRISS

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

Bases: *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	
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: *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

RomanCoronagraph

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

Bases: `RomanInstrument`

Roman Coronagraph Instrument

Simulates the PSF of the Roman 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.

Parameters

mode
[str] Roman Coronagraph Instrument observing mode. If not specified, the __init__ function will set this to a default mode 'CHARSPC_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	List of available filter names for this instrument
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_position
The pixel position in (X, Y) on the detector

filter
Currently selected filter name

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

List of available filter names for this instrument

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

SpaceTelescopeInstrument

class webbpsf.**SpaceTelescopeInstrument**(*name=''*, *pixelscale=0.064*)

Bases: `Instrument`

A generic Space Telescope Instrument class.

Note: Do not use this class directly - instead use one of the specific instrument subclasses!

This class provides a simple interface for modeling PSF formation through the instrument, with configuration options and software interface loosely resembling the configuration of the instrument hardware mechanisms.

This module currently only provides a modicum of error checking, and relies on the user being knowledgable enough to avoid trying to simulate some physically impossible or just plain silly configuration (such as trying to use a FQPM with the wrong filter).

The instrument constructors do not take any arguments. Instead, create an instrument object and then configure the `filter` or other attributes as desired. The most commonly accessed parameters are available as object attributes: `filter`, *`image_mask`*, *`pupil_mask`*, `pupilopd`. More advanced configuration can be done by editing the SpaceTelescopeInstrument.options dictionary, either by passing options to `__init__` or by directly editing the dict afterwards.

Attributes Summary

aperturename	SIAF aperture name for detector pixel to sky coords transformations
detector	Detector selected for simulated PSF
detector_list	Detectors on which the simulated PSF could lie
detector_position	The pixel position in (X, Y) on the detector, relative to the currently-selected SIAF aperture subarray.
image_mask	Currently selected image plane mask, or None for direct imaging
options	A dictionary capable of storing other arbitrary options, for extensibility.
pupil_mask	Currently selected Lyot pupil mask, or None for direct imaging
telescope	

Methods Summary

get_optical_system([fft_oversample, ...])	Return an OpticalSystem instance corresponding to the instrument as currently configured.
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.

Attributes Documentation

aperturename

SIAF aperture name for detector pixel to sky coords transformations

detector

Detector selected for simulated PSF

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

detector_list

Detectors on which the simulated PSF could lie

detector_position

The pixel position in (X, Y) on the detector, relative to the currently-selected SIAF aperture subarray. By default the SIAF aperture will correspond to the full-frame detector, so (X,Y) will in that case be absolute (X,Y) pixels on the detector. But if you select a subarray aperture name from the SIAF, then the (X,Y) are interpreted as (X,Y) within that subarray.

Please note, this is X,Y order - **not** a Pythonic y,x axes ordering.

image_mask

Currently selected image plane mask, or None for direct imaging

`options = {}`

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

This is a superset of the options provided in `poppy.Instrument.options`.

Parameters

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

source_offset_theta
[float] Position angle for that offset, in degrees CCW.

pupil_shift_x, pupil_shift_y
[float] Relative shift of the intermediate (coronagraphic) pupil in X and Y relative to the telescope entrance pupil, expressed as a decimal between -1.0-1.0 Note that shifting an array too much will wrap around to the other side unphysically, but for reasonable values of shift this is a non-issue. This option only has an effect for optical models that have something at an intermediate pupil plane between the telescope aperture and the detector.

pupil_rotation
[float] Relative rotation of the intermediate (coronagraphic) pupil relative to the telescope entrace pupil, expressed in degrees counterclockwise. This option only has an effect for optical models that have something at an intermediate pupil plane between the telescope aperture and the detector.

rebin
[bool] For output files, write an additional FITS extension including a version of the output array rebinned down to the actual detector pixel scale?

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

jitter_sigma
[float] Width of the jitter kernel in arcseconds (default: 0.006 arcsec, 1 sigma per axis)

parity
[string "even" or "odd"] You may wish to ensure that the output PSF grid has either an odd or even number of pixels. Setting this option will force that to be the case by increasing npix by one if necessary. Note that this applies to the number detector pixels, rather than the subsampled pixels if oversample > 1.

force_coron
[bool] Set this to force full coronagraphic optical propagation when it might not otherwise take place (e.g. calculate the non-coronagraphic images via explicit propagation to all optical surfaces, FFTing to intermediate pupil and image planes whether or not they contain any actual optics, rather than taking the straight-to-MFT shortcut)

no_sam
[bool] Set this to prevent the SemiAnalyticMethod coronagraph mode from being used when possible, and instead do the brute-force FFT calculations. This is usually not what you want to do, but is available for comparison tests. The SAM code will in general be much faster than the FFT method, particularly for high oversampling.

`pupil_mask`

Currently selected Lyot pupil mask, or None for direct imaging

`telescope = 'Generic Space Telescope'`

Methods Documentation

`get_optical_system`(*fft_oversample=2*, *detector_oversample=None*, *fov_arcsec=2*, *fov_pixels=None*, *options=None*)

Return an OpticalSystem instance corresponding to the instrument as currently configured.

When creating such an OpticalSystem, you must specify the parameters needed to define the desired sampling, specifically the oversampling and field of view.

Parameters

fft_oversample
[int] Oversampling factor for intermediate plane calculations. Default is 2

detector_oversample: int, optional
By default the detector oversampling is equal to the intermediate calculation oversampling. If you wish to use a different value for the detector, set this parameter. Note that if you just want images at detector pixel resolution you will achieve higher fidelity by still using some oversampling (i.e. *not* setting `oversample_detector=1`) and instead rebinning down the oversampled data.

fov_pixels
[float] Field of view in pixels. Overrides fov_arcsec if both set.

fov_arcsec
[float] Field of view, in arcseconds. Default is 2

Returns

osys
[poppy.OpticalSystem] an optical system instance representing the desired configuration.

`psf_grid`(*num_psfs=16*, *all_detectors=True*, *save=False*, *outdir=None*, *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.

Parameters

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.

outdir
[str] If "save" keyword is set to True, your file will be saved in the specified directory. Default of None will save it in the current directory

outfile
[str] If "save" keyword is set to True, your file will be saved as {outfile}_det.fits. Default of None will save it 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.

Returns

gridmodel
[photutils GriddedPSFModel object or list of objects] 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.

UnsupportedPythonError

exception webbpsf.**UnsupportedPythonError**

WFI

class webbpsf.**WFI**

Bases: RomanInstrument

WFI represents the Roman mission's Wide Field Imager.

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).

Attributes Summary

detector	Detector selected for simulated PSF
filter	Currently selected filter name (e.g.
mode	The current WFI mode.
pupil	The path to the FITS file containing pupil information for the detector/filter combination sent from the WFI class.
pupil_mask	The corresponding mask for the current filter.

Methods Summary

lock_aberrations(aberration_path)	This function loads user provided aberrations from a file and locks this instrument to only use the provided aberrations (even if the filter or mode change).
lock_pupil(pupil_path)	Prevents dynamic updates of the path to the proper pupil file on any changes to the selected detector or filter.
lock_pupil_mask(pupil_mask)	Prevents dynamic updates of the pupil mask on any change to the selected filter.
unlock_aberrations()	Releases the lock on the detector aberration file location and loads the default file.
unlock_pupil()	Undoes the effects of lock_pupil() by resetting the class to its default state of updating the pupil whenever a detector or filter is changed.
unlock_pupil_mask()	Undoes the effects of lock_pupil_mask() and resets the class to its default state of updating the pupil mask whenever the filter is changed.

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.

filter

Currently selected filter name (e.g. F200W)

mode

The current WFI mode. Cannot be directly set by the user.

pupil

Filename *or* fits.HDUList for the pupil mask.

pupil_mask

The corresponding mask for the current filter. Cannot be directly set by the user.

Methods Documentation

lock_aberrations(*aberration_path*)

This function loads user provided aberrations from a file and locks this instrument to only use the provided aberrations (even if the filter or mode change).

To release the lock and load the default aberrations, use unlock_aberrations(). To load new user provided aberrations, call this function with the new path.

To load custom aberrations, please provide a csv file containing the detector names, field point positions and Zernike values. The file should contain the following column names/values (comments in parentheses should not be included):

- sca (Detector number)

- wavelength (μm)
- field_point (field point number/ID for SCA and wavelength,

starts with 1) - local_x (mm, local detector coords) - local_y (mm, local detector coords) - global_x (mm, global instrument coords) - global_y (mm, global instrument coords) - axis_local_angle_x (XAN) - axis_local_angle_y (YAN) - wfe_rms_waves (nm) - wfe_pv_waves (waves) - Z1 (Zernike phase NOLL coefficients) - Z2 (Zernike phase NOLL coefficients) - Z3 (Zernike phase NOLL coefficients) - Z4 (Zernike phase NOLL coefficients)

.

Please refer to the default aberration files for examples. If you have the WebbPSF data installed and defined, you can get the path to that file by running the following:

```
>>> from webbpsf import roman
>>> wfi = roman.WFI()
>>> print(wfi._aberration_files["imaging"])
```

Warning: You should not edit the default files!

lock_pupil(*pupil_path*)

Prevents dynamic updates of the path to the proper pupil file on any changes to the selected detector or filter. Instead, the path remains locked on whichever `pupil_path` was provided here.

WARNING: This is non-standard usage of the WFI class and may lead to unexpected behavior.

Parameters

pupil_path
[string] The custom path to your pupil file.

lock_pupil_mask(*pupil_mask*)

Prevents dynamic updates of the pupil mask on any change to the selected filter. Instead, the pupil mask remains locked on whichever *pupil_mask* was provided here.

WARNING: This is non-standard usage of the WFI class and may lead to unexpected behavior.

Parameters

filter
[string] See WFI.pupil_mask_list for a list of valid pupil masks.

unlock_aberrations()

Releases the lock on the detector aberration file location and loads the default file.

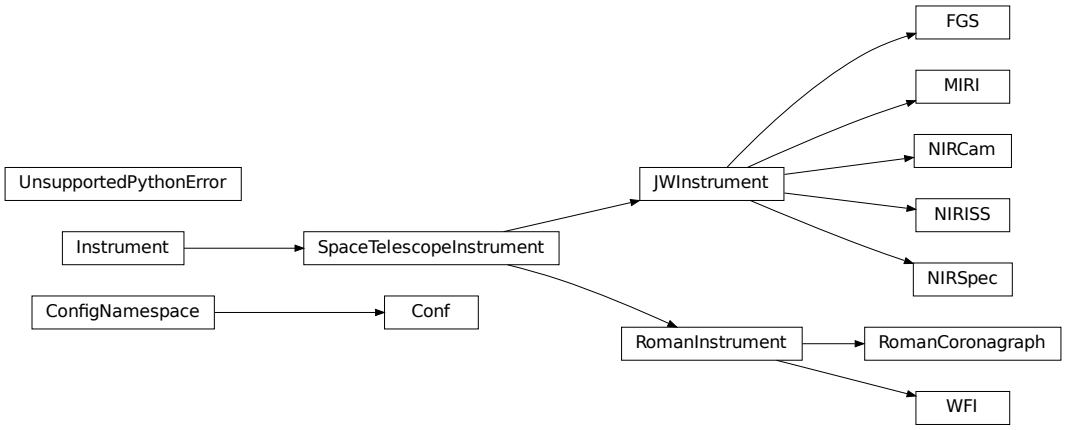
unlock_pupil()

Undoes the effects of lock_pupil() by resetting the class to its default state of updating the pupil whenever a detector or filter is changed. If necessary, it also sets the proper pupil for the current detector/filter combination.

unlock_pupil_mask()

Undoes the effects of lock_pupil_mask() and resets the class to its default state of updating the pupil mask whenever the filter is changed.

Class Inheritance Diagram



10.2 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
```

(continues on next page)

(continued from previous page)

```
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
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
---------------------------------------------------------------------
```

10.3 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.

10.3.1 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.

10.3.2 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.

10.3.3 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)
```

(continues on next page)

(continued from previous page)

```
     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)
     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)
        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.

10.4 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 created numpy arrays which have this property. While I expected using this would make the transforms faster, in fact I see significantly better performance when using unaligned arrays. (The speed difference becomes larger as array size increases, up to 2x!) This is unexpected and not understood, so it may vary by machine and I suggest one ought to test this on different machines to see if it is reliable.

10.4.1 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

10.4.2 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
```

(continues on next page)

(continued from previous page)

```
   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.

10.4.3 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
```

10.4.4 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
```

(continues on next page)

(continued from previous page)

```
        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
        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
```

10.4.5 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
 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
```

(continues on next page)

(continued from previous page)

```
        Planning took   0.147264        0.227571 s
        Executing took  4.640368        0.528359 s
```

10.4.6 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
poppy     : INFO       Calculation completed in 0.082 s
poppy     : INFO     PSF Calculation completed.
```

CHAPTER

ELEVEN

APPENDICES AND REFERENCE

11.1 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

11.2 WebbPSF Multifield Model for OTE WFE

This page documents the model for the field dependence of the OTE WFE as implemented in WebbPSF.

Background: Wavefront error is in general somewhat different at each of the different points in the image field. These wavefront errors are inherent in the design of the optical system, even when the system is aligned perfectly.

As a result, one can think of the system's field-varying wavefront error, as being dependent on four coordinates,

$where (x, y) are the lateral coordinate of the wavefront in the system's pupil plane and (u, v) are the lateral coordinates of the image$

However, in WebbPSF (and other diffraction computation tools) is is necessary to know the wavefront distribution across a sampled version of the entire pupil,

$Most tools assume that the field variations are constant over the region of interest, typically a single field point. Optical designs of$

$$\phi(x, y) = \sum_j \alpha_j Z_j(x, y) .$$

$The set of Zernike polynomials is a popular choice here.$

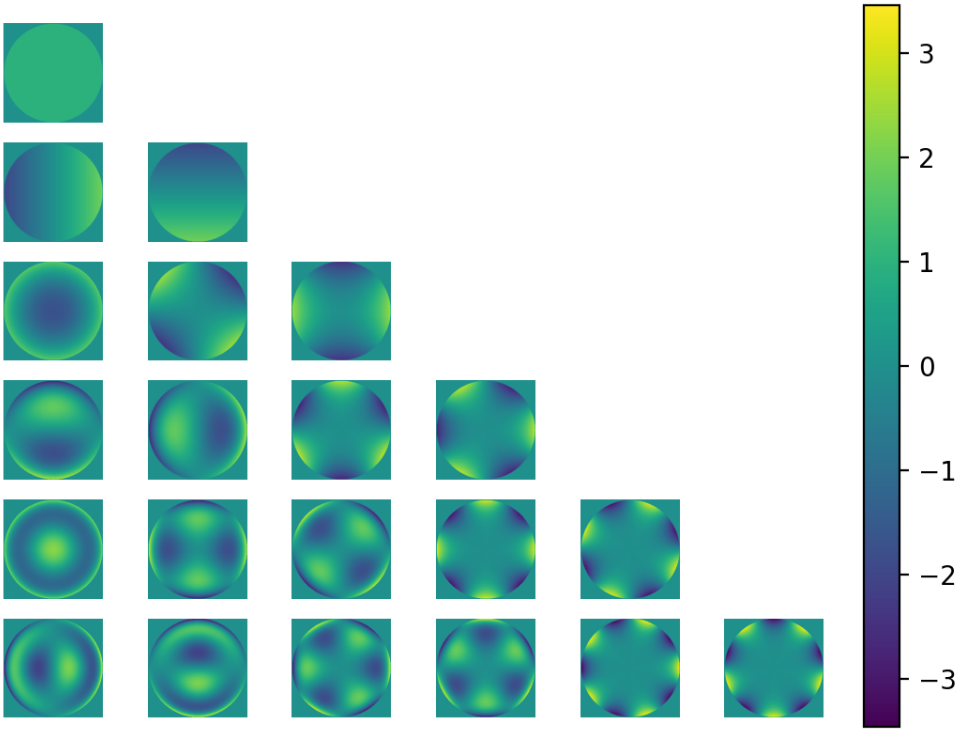


Figure 1: The first 21 Zernike polynomials, using the ordering and normalization of R.J. Noll, JOSA 66, 207-211 (1976).

When dealing with systems where there are significant field-dependent aberrations, particularly large-field systems like JWST, a single field assumption is often inadequate to represent the performance of the optical system. The most direct approach to handling this is to compute the wavefront using an optical analysis program on-demand for each field point of interest. This is a very computationally expensive approach. An alternate approach is to store a large ensemble of wavefronts, discrete sampling various field points, and perform multidimensional interpolation to compute the wavefront at the desired field point. This is storage intensive and somewhat computationally expensive. A third approach, used here, is to use a polynomial expansion using Zernikes polynomials where the polynomial coefficients $\alpha_j\,(u, v)$ vary with field. Thus the wavefront is computed using expression

$$\phi(x, y; u, v) = \sum_j$$

$The field varying characteristic of the polynomial coefficients is captured by performing a second polynomial expansion on each o$

$where : math : `\beta_{j,k}` is the coefficient of this second expansion corresponding to the : math : `j^{th}` Zernike term and : math : `L$

of interest are typically rectangular and the Legendres are orthonormal over a rectangular region.

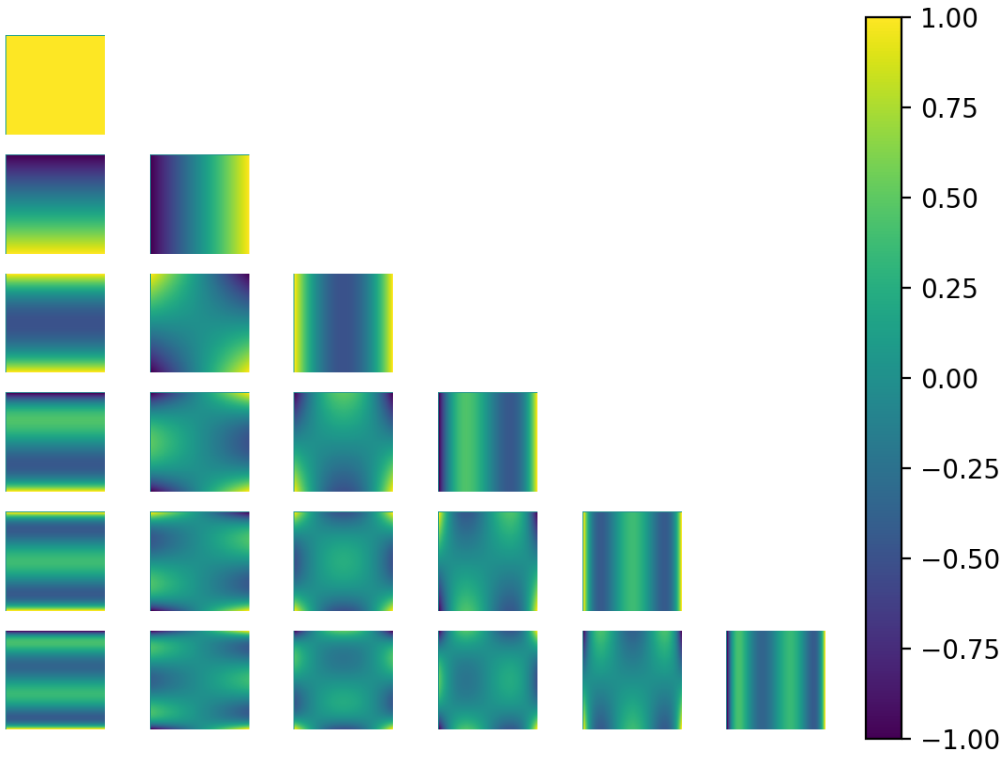


Figure 2: The first 21 Legendre polynomials.

Thus field dependence of the optical system of interest can be entirely characterized by a relatively small, two-dimensional matrix of $\beta_{j,k}$ coefficients. These coefficients have been precomputed for use in WebbPSF by using the most up-to-date CodeV models of the JWST optical telescope assembly (OTE). For the highest fidelity, a set of these coefficients is computed separately for each of the instrument fields.

11.3 Field Dependence in JWST

To build our field dependence model in WebbPSF, we have employed the as-built CodeV model of JWST derived from the nominal optical design for JWST, measurements of the figure of individual components, and end-to-end measurements of the OTE wavefront error conducted in the course of the JWST test campaign. In the figure below we show the wavefront at a selection of field points, generally in the corners of the instrument fields. The variation of wavefront with field angle is quite evident.

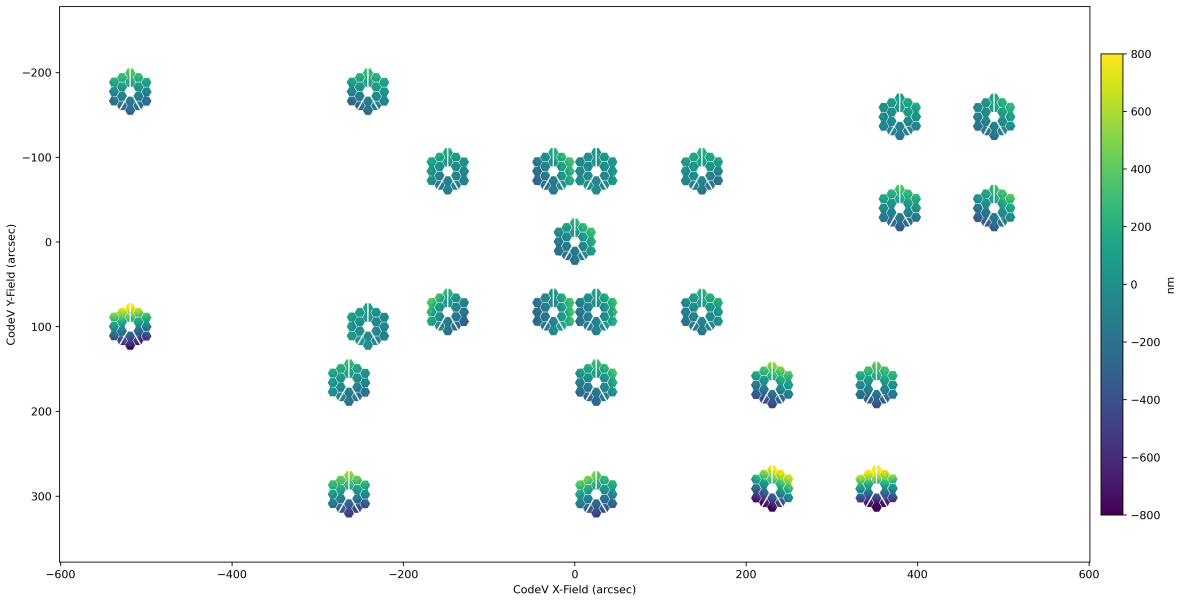


Figure 3: CodeV modeled wavefronts. Note that the X and Y field angles in this figure are analagous to the (u, v) coordinates in our above notation.

For practical use with WebbPSF, we are interested in the variation of the wavefront with field from the nominal wavefront defined in WebbPSF. To facilitate these, we subtract the wavefront modeled by CodeV at its (0,0) field point. In JWST (v_2, v_3) field coordinates, this is at the location (0, -468 arcsec). The following figure shows the field-dependent wavefronts at the same points as above but with the nominal wavefront subtracted and plotted in the (v_2, v_3) coordinate system. Residual piston, tip and tilt terms are also removed from the wavefront and are not modeled.

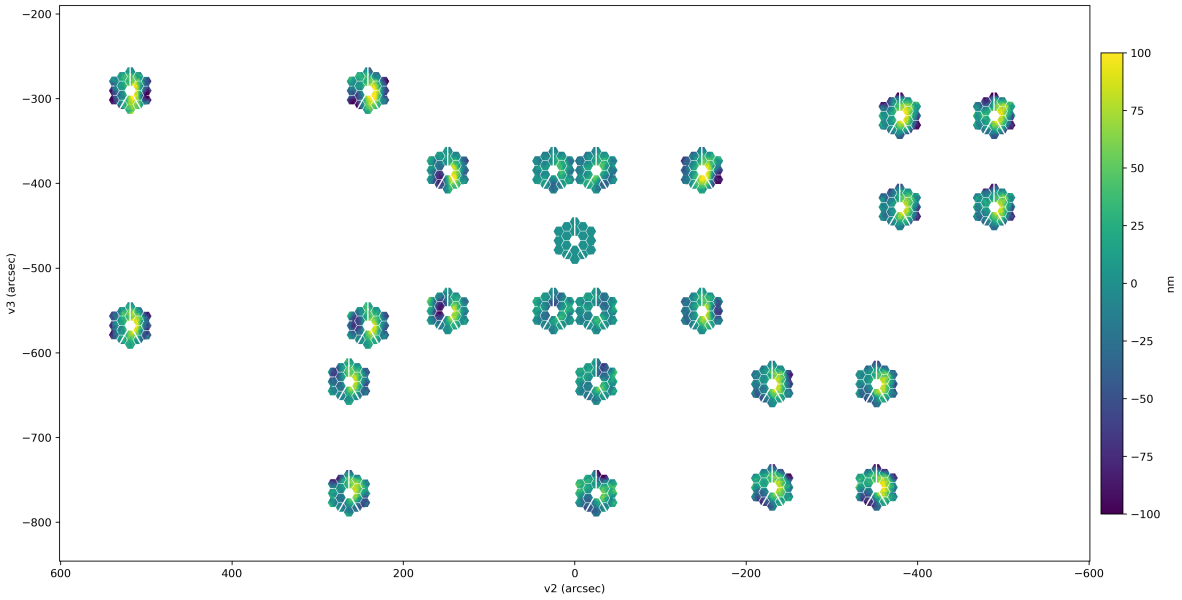


Figure 4: CodeV modeled wavefronts with nominal wavefront and piston, tip and tilt removed. JWST (v_2, v_3) is the lateral coordinate system used here.

If we then use our polynomial-based WebbPSF model to calculate the wavefront variation at the same field points we have the following result.

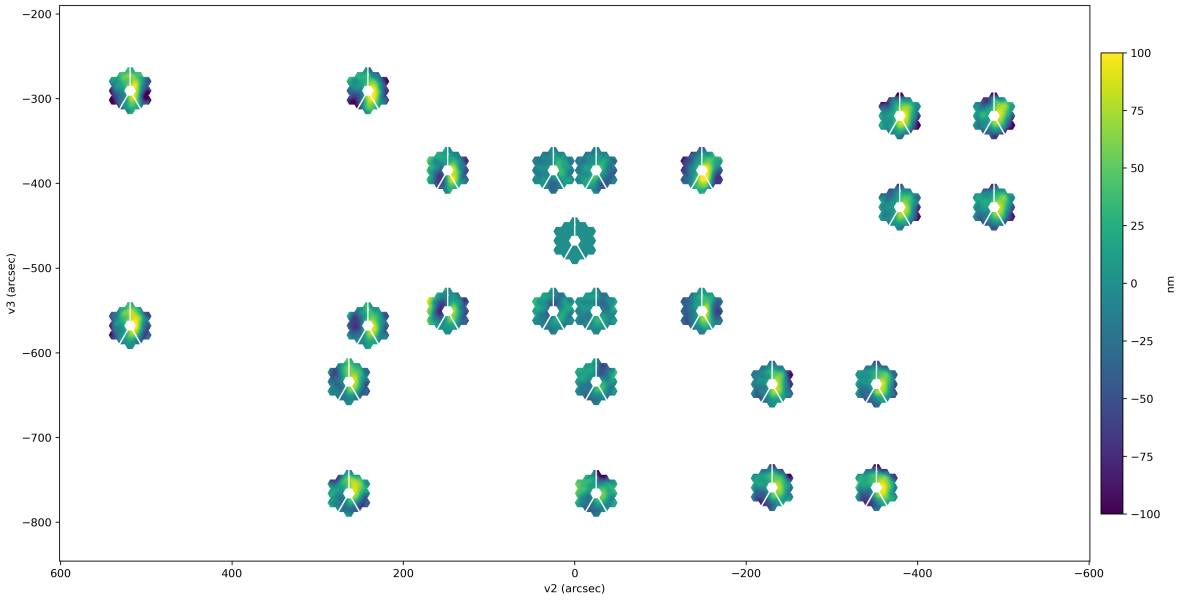


Figure 5: WebbPSF modeled wavefronts with nominal wavefront and piston, tip and tilt removed. JWST (v_2, v_3) is the lateral coordinate system used here.

11.4 Model usage in WebbPSF

The model is implemented in the `_get_field_dependence_nominal_ote()` method of the `OTE_Linear_Model_WSS` class in the opds.py file of WebbPSF. WebbPSF will apply the field-dependence model automatically when used in the ordinary way. The following illustrates the calculation of a PSF that uses the field-dependent model

First, do some setup

[ ]:
```
%pylab inline
matplotlib.rcParams['image.origin'] = 'lower'
import webbpsf
import astropy.units as u
webbpsf.setup_logging()
```

Set up an instance of a WebbPSF NIRCam object with the F444W filter

[2]:
```
nc = webbpsf.NIRCam()
nc.filter='F444W'
nc.detector = 'NRCA5'
```

```
[webbpsf] NIRCam aperture name updated to NRCA1_FULL
[webbpsf] NIRCam pixel scale switched to 0.063000 arcsec/pixel for the long wave channel.
[webbpsf] NIRCam aperture name updated to NRCA5_FULL
```

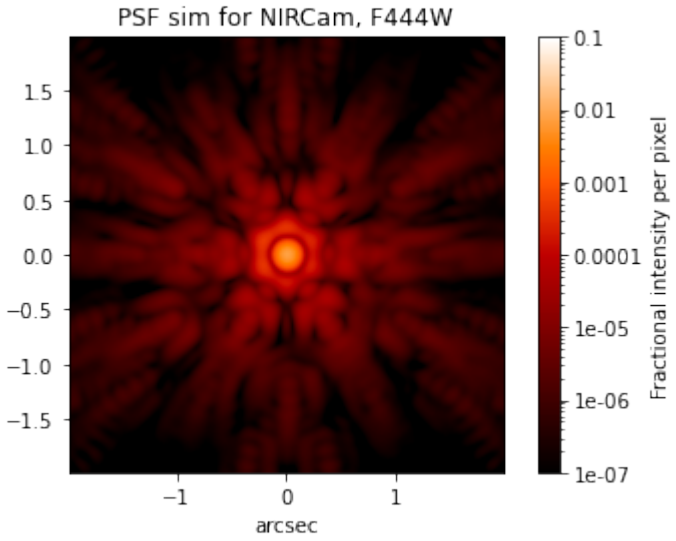
Now we call the calc_psf method. This includes WebbPSF's data for the pupil wavefront and the apply the field-dependent perturbation calculated using our polynomial model. Note the logging indicating that the field dependence has been applied.

[3]:
```
psf = nc.calc_psf(nlambda=5, fov_arcsec=4, display=False)
```

```
[  poppy] No source spectrum supplied, therefore defaulting to 5700 K blackbody
[  poppy] Computing wavelength weights using synthetic photometry for F444W...
[  poppy] PSF calc using fov_arcsec = 4.000000, oversample = 4, number of wavelengths = 5
[webbpsf] Creating optical system model:
[  poppy] Initialized OpticalSystem: JWST+NIRCam
[  poppy] JWST Entrance Pupil: Loaded amplitude transmission from /Users/gbrady/
↪Documents/Projects/WebbPSF/webbpsf-data/jwst_pupil_RevW_npix1024.fits.gz
[  poppy] JWST Entrance Pupil: Loaded OPD from /Users/gbrady/Documents/Projects/WebbPSF/
↪webbpsf-data/NIRCam/OPD/OPD_RevW_ote_for_NIRCam_requirements.fits.gz
[webbpsf] Field coordinates determined to be in NIRCam field
[webbpsf] Loading field dependent model parameters from /Users/gbrady/Documents/Projects/
↪WebbPSF/webbpsf-data/NIRCam/OPD/field_dep_table_nircam.fits
[webbpsf] Calculating field-dependent OPD at v2 = 1.435 arcmin, v3 = -8.224 arcmin
[  poppy] Added pupil plane: JWST Entrance Pupil
[  poppy] Added coordinate inversion plane: OTE exit pupil
[  poppy] Added pupil plane: NIRCamLWA internal WFE at V2V3=(1.43,-8.22)', near MIMF5
[  poppy] Added detector with pixelscale=0.063 and oversampling=4: NIRCam detector
[  poppy] Calculating PSF with 5 wavelengths
[  poppy]  Propagating wavelength = 3.97009e-06 m
[webbpsf]   Applying OPD focus adjustment based on NIRCam focus vs wavelength model
[webbpsf]   Modified focus from 3.23 to 3.970093517034068 um: 47.765 nm wfe
[webbpsf]   Resulting OPD has 92.385 nm rms
[  poppy]  Propagating wavelength = 4.21298e-06 m
[webbpsf]   Applying OPD focus adjustment based on NIRCam focus vs wavelength model
[webbpsf]   Modified focus from 3.23 to 4.212981352705411 um: 65.575 nm wfe
[webbpsf]   Resulting OPD has 75.663 nm rms
[  poppy]  Propagating wavelength = 4.45587e-06 m
[webbpsf]   Applying OPD focus adjustment based on NIRCam focus vs wavelength model
[webbpsf]   Modified focus from 3.23 to 4.455869188376753 um: 95.425 nm wfe
[webbpsf]   Resulting OPD has 49.371 nm rms
[  poppy]  Propagating wavelength = 4.69876e-06 m
[webbpsf]   Applying OPD focus adjustment based on NIRCam focus vs wavelength model
[webbpsf]   Modified focus from 3.23 to 4.698757024048096 um: 138.693 nm wfe
[webbpsf]   Resulting OPD has 28.912 nm rms
[  poppy]  Propagating wavelength = 4.94164e-06 m
[webbpsf]   Applying OPD focus adjustment based on NIRCam focus vs wavelength model
[webbpsf]   Modified focus from 3.23 to 4.941644859719438 um: 196.755 nm wfe
[webbpsf]   Resulting OPD has 67.598 nm rms
[  poppy]   Calculation completed in 0.832 s
[  poppy] PSF Calculation completed.
[webbpsf] Calculating jitter using gaussian
[  poppy] Calculating jitter using gaussian
[  poppy] Jitter: Convolving with Gaussian with sigma=0.007 arcsec
[  poppy]         resulting image peak drops to 0.992 of its previous value
[  poppy]  Adding extension with image downsampled to detector pixel scale.
[  poppy]  Downsampling to detector pixel scale, by 4
[  poppy]  Downsampling to detector pixel scale, by 4
```

Now we display the PSF

[4]:
```
webbpsf.display_psf(psf)
```



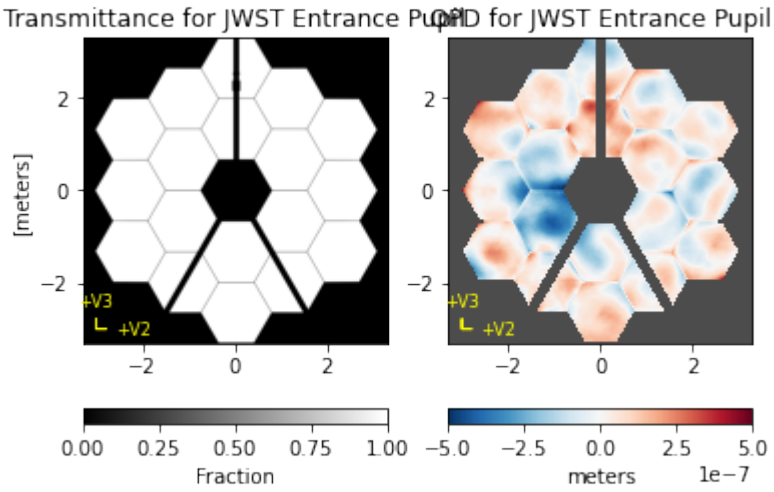
nbsphinx-code-borderwhite

We can display the pupil function used to compute this by getting the optical system object and using its visualization methods.

[5]:
```
opt_sys = nc.get_optical_system()
opt_sys.planes[0].display(what='both')
```

```
[webbpsf] Creating optical system model:
[  poppy] Initialized OpticalSystem: JWST+NIRCam
[  poppy] JWST Entrance Pupil: Loaded amplitude transmission from /Users/gbrady/
↪Documents/Projects/WebbPSF/webbpsf-data/jwst_pupil_RevW_npix1024.fits.gz
[  poppy] JWST Entrance Pupil: Loaded OPD from /Users/gbrady/Documents/Projects/WebbPSF/
↪webbpsf-data/NIRCam/OPD/OPD_RevW_ote_for_NIRCam_requirements.fits.gz
[webbpsf] Field coordinates determined to be in NIRCam field
[webbpsf] Loading field dependent model parameters from /Users/gbrady/Documents/Projects/
↪WebbPSF/webbpsf-data/NIRCam/OPD/field_dep_table_nircam.fits
[webbpsf] Calculating field-dependent OPD at v2 = 1.435 arcmin, v3 = -8.224 arcmin
[  poppy] Added pupil plane: JWST Entrance Pupil
[  poppy] Added coordinate inversion plane: OTE exit pupil
[  poppy] Added pupil plane: NIRCamLWA internal WFE at V2V3=(1.43,-8.22)', near MIMF5
[  poppy] Added detector with pixelscale=0.063 and oversampling=2: NIRCam detector
```

[5]:
```
(<AxesSubplot:title={'center':'Transmittance for JWST Entrance Pupil'}, ylabel='[meters]
↪'>,
 <AxesSubplot:title={'center':'OPD for JWST Entrance Pupil'}>)
```



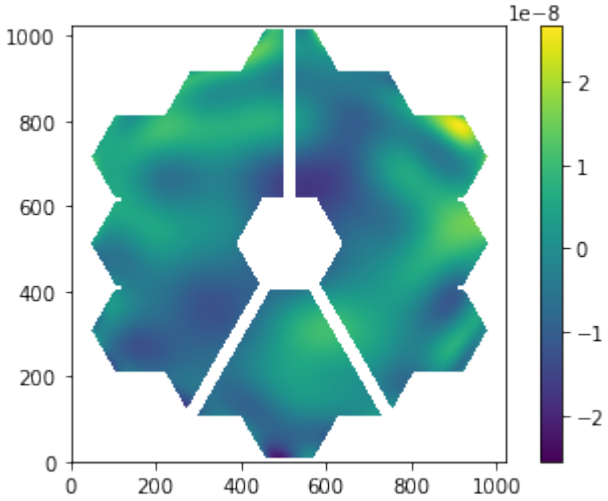
nbsphinx-code-borderwhite

We can also directly access the OPD perturbation using an instance of the OTE_Linear_Model_WSS class, which can take the field point v2,v3 as an argument. This allows us to compute the OPD perturbation at an arbitrary field point.

[6]:
```
# Define field coordinate in angular units
v2 = 0 * u.arcsec
v3 = -500 * u.arcsec
# Instantiate OTE_Linear_Model_WSS
linmod = webbpsf.opds.OTE_Linear_Model_WSS(v2v3=(v2,v3))
# Get the OPD perturbation
plot_data = linmod.opd
# Get the pupil transmission
data_mask = linmod.get_transmission(0) != 0
# Set it up so we don't plot anything outside of the pupil
plot_data[~data_mask] = np.nan
# Do a simple plot of the data using imshow
plt.imshow(plot_data)
plt.colorbar()
```

```
[  poppy] Unnamed OPD: Loaded amplitude transmission from /Users/gbrady/Documents/
↪Projects/WebbPSF/webbpsf-data/jwst_pupil_RevW_npix1024.fits.gz
[webbpsf] Field coordinates determined to be in NIRCam field
[webbpsf] Loading field dependent model parameters from /Users/gbrady/Documents/Projects/
↪WebbPSF/webbpsf-data/NIRCam/OPD/field_dep_table_nircam.fits
[webbpsf] Calculating field-dependent OPD at v2 = 0.000 arcsec, v3 = -500.000 arcsec
```

[6]:
```
<matplotlib.colorbar.Colorbar at 0x11d622f90>
```



nbsphinx-code-borderwhite

We can use this same methodology to calculate the OPD distribution at a bunch of field points and plot them all on the same axis

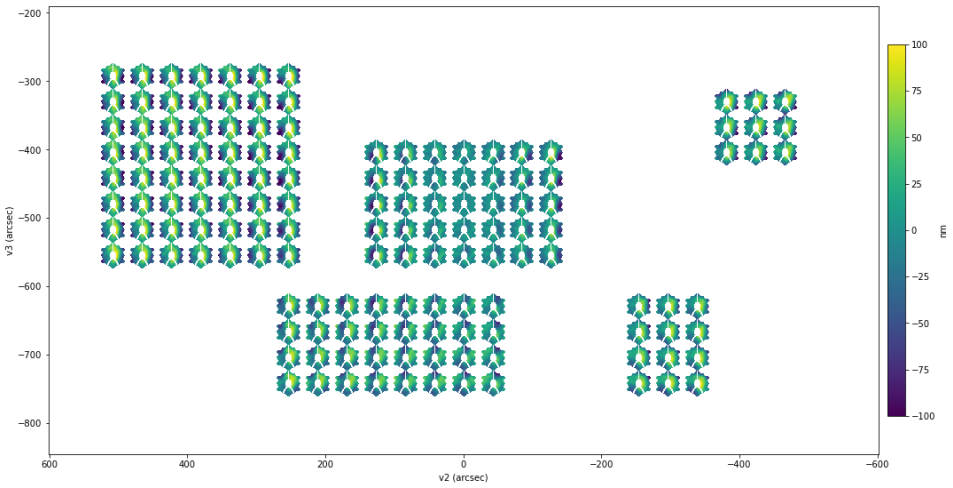
[7]:
```
# Minimize logging output becuase it's a little overbearing here.
webbpsf.setup_logging(level='ERROR')

# Set up the limits of the very large field region over which we will calculate the OPD
min_v2field = 550 * u.arcsec
max_v2field = -550 * u.arcsec
v2field_extent = max_v2field - min_v2field
v2field_center = (max_v2field + min_v2field)/2
max_v3field = (250 - 468) * u.arcsec
min_v3field = (-350 - 468) * u.arcsec
v3field_extent = max_v3field - min_v3field
v3field_center = (max_v3field + min_v3field)/2

# How many different field point will we sample the above region with
num_v2 = 27
num_v3 = 17

# Size of each idividual OPD plot
tile_size = 0.06
# Leave a border around the edge of the plot
border = 0.085/2

#Set up the initial figure axes
fig = plt.figure(figsize=(13, 7), facecolor='white')
ax = fig.add_axes([0, 0, 1, 1])
ax.set_xlim((v2field_center-0.5 * (v2field_extent/(1 - 2 * border))) / u.arcsec,
            (v2field_center+0.5 * (v2field_extent/(1 - 2 * border))) / u.arcsec)
ax.set_ylim((v3field_center-0.5 * (v3field_extent/(1 - 2 * border))) / u.arcsec,
            (v3field_center+0.5 * (v3field_extent/(1 - 2 * border))) / u.arcsec)
ax.set_xlabel('v2 (arcsec)')
ax.set_ylabel('v3 (arcsec)')
```

(continues on next page)

(continued from previous page)

```
# Loop over the desired field points
for v2 in np.linspace(min_v2field,max_v2field,num_v2):
    for v3 in np.linspace(min_v3field,max_v3field,num_v3):
        # Setting the OPD calculation inside a try block.  If the (v2,v3) isn't inside a
        # valid instrument field a ValueError will be thrown
        try:
            # Instance of OTE_Linear_Model_WSS object at the desired field point
            linmod = webbpsf.opds.OTE_Linear_Model_WSS(v2v3=(v2,v3))
            # Get the OPD and mask off the region where the data is valid
            plot_data = linmod.opd
            data_mask = linmod.get_transmission(0) != 0
            plot_data[~data_mask] = np.nan
            # Calculate where to place the plot of the OPD at the current field point
            offset = (border + (1 - 2 * border) * (v2 - min_v2field)/v2field_extent -␣
↪tile_size/2,
                      border + (1 - 2 * border) * (v3 - min_v3field)/v3field_extent -␣
↪tile_size/2)
            # Create an axis for the OPD plot and plot it
            ax = fig.add_axes([offset[0], offset[1], tile_size, tile_size])
            img = ax.imshow(plot_data*1e9, vmin=-100, vmax=100, origin='lower')
            ax.axis('off')
        except ValueError:   # We get here if we use a field point not within a valid␣
↪instrument field
            pass   # So we don't plot anything
# Create a colorbar
cbax = fig.add_axes([1 + border/4, 2 * border, border/2, 1 - 4 * border])
cbar = fig.colorbar(img, cax=cbax)
cbar.set_label('nm')
```

```
WebbPSF log messages of level ERROR and above will be shown.
WebbPSF log outputs will be directed to the screen.
```



nbsphinx-code-borderwhite

```
[ ]:
```

11.5 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.

11.5.1 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)

11.5.2 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.

11.5.3 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.

11.5.4 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.

11.5.5 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 prism 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).

11.5.6 Instrument + Filter Throughputs

Instrumental relative spectral responses are derived from the reference data files used by the Pandeia engine for the JWST Exposure Time Calculator, normalized to peak transmission=1.0 (because absolute throughput is not relevant for PSF calculations).

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.

11.6 Developer Notes: Releasing a new version of WebbPSF

11.6.1 Prerequisites

- Is the `develop` build passing on Travis? with all desired release items included?

11.6.2 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.
    1. This now means on Box. Upload to Box in the webbpsf shared data folder. Get the Box shared file URL.
    2. Update `docs/installation.rst` to have that new URL and updated filename in the appropriate location.
5. Update the shared copy on STScI Central Store:
    1. `cd` to `/grp/jwst/ote` and remove the `webbpsf-data` symlink
    2. Copy the archive into `/grp/jwst/ote/` and extract it to `/grp/jwst/ote/webbpsf-data`
    3. Rename the folder to `webbpsf-data-0.x.y`
    4. Create a symbolic link at `/grp/jwst/ote/webbpsf-data` to point to the new folder
6. Update the URL in `installation.rst` under *Installing the Required Data Files*

Details for using `make-data-sdist.sh`:

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/
```

11.6.3 Releasing new versions

If you are making a release for `poppy` at the same time as a release in WebbPSF, do that first. Update the dependency requirement to the new version of poppy, in `webbpsf/setup.cfg`.

When you are ready, proceed with the WebbPSF release as follows:

1. Get the `develop` branch into the state that you want, including all PRs merged, updated release notes. This includes all tests passing both locally and on Travis.
2. Tag the commit with `v`, being sure to sign the tag with the `-s` option. * `git tag -s v<version> -m "Release v<version>"`
3. Push tag to github, on `develop`

4. On github, make a PR from `develop` to `stable` (this can be done ahead of time and left open, until all individual PRs are merged into `develop`.).
5. After verifying that PR is complete and tests pass, merge it. (Once merged, both the `stable` and `develop` branches should match).
6. Release on Github:
    1. On Github, click on "[N] Releases".
    2. Select "Draft a new release".
    3. Specify the version number, title, and brief description of the release.
    4. Press "Publish Release".
7. Release to PyPI. This should now happen automatically on Travis. This will be triggered by a Travis build of a tagged commit on the `stable` branch, so it will happen automatically on the prior step for the PR into stable.
8. Release to AstroConda, via steps below.

11.6.4 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*.

11.6.5 Finishing the release

1. Email an announcement to `webbpsf-users@maillist.stsci.edu`

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

W

INDEX

M

N

O

P

R

S

T

U

V

W