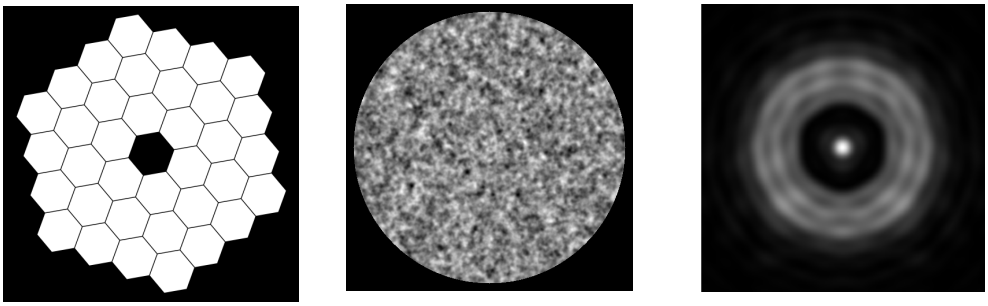
PROPER

An Optical Propagation Library for
IDL, Python, & Matlab



Available from proper-library.sourceforge.net

Developed by John Krist
Matlab conversion by Gary Gutt
Python conversion by Navtej Saini with Nikta Amiri and Luis Marchen

NASA Jet Propulsion Laboratory
California Institute of Technology

Version 3.2.7

15 June 2022

Acknowledgements

The author wishes to thank the following for discussions, suggestions, nagging, etc., that helped make this code possible: (JPL) Dwight Moody, Karl Stapelfeldt, John Trauger, Joe Green, David Palacios, Phil Dumont, Stuart Shaklan; (STScI) Anand Sivaramakrishnan; (Space Telescope Science Institute) Russ Makidon; (LLNL) Lisa Poyneer; Christian Marois, Luc Gilles (TMT). Thanks to Volker Tolls (Smithsonian Astrophysical Observatory) for help with the Windows installation instructions and provided DLLs. Thanks to James Tappan, Sunip Mukherjee, Shannon Zareh, Bryn Jeffries, AJ Riggs, Yinzi Xin, and Roser Juanola for bug reports (and even suggested fixes).

Development of PROPER was funded by the NASA Terrestrial Planet Finder Coronagraph project at JPL. Conversions to Matlab and Python were funded by the NASA WFIRST coronagraph project at JPL.

Legal Notices

Table of Contents

Changes

Version 3.2.x

A large number of bug fixes, mostly to the Python and Matlab versions. Added the KEEP_THREADS option to the IDL and Matlab versions of PROP_RUN_MULTI; this allows parallel processes created in PROP_RUN_MULTI to remain in existance for use in later calls, avoiding the overheads in creating new processes; these can be released using the new PROP_FREE_THREADS routine or calling PROP_RUN_MULTI without the KEEP_THREADS option. PROP_SET_ANTIALIASING was added to allow the user to set the subsampling factor used for antialiasing the edges of all shapes. In the IDL and Python versions, the number of threads used for the optional FFTW and Intel MKL FFT routines for single (PROP_RUN) and multiple (PROP_RUN_MULTI) processes was adjusted, and the loading of user-generated FFTW wisdom was put into PROP_BEGIN. Rotation of ellipses was added to PROP_ELLIPSE, PROP_ELLIPTICAL_APERTURE, and PROP_ELLIPTICAL_OBSCURATION (modification based on code by A.J. Riggs). In the Python version, PROP_FIT_ZERNIKES was only fitting the last requested Zernike when obscured Zernikes were used (due to an indentation error – stupid Python!). Version 3.2.1 (Python only) PROP_FIT_ZERNIKES was ignoring the values for XC and YC, if provided. Version 3.2.2 (Python v3.x only) Fixed bug in PROP_DM that caused a crash in Python v3.8 when calling fftconvolve due to incompatible array types. Version 3.2.4 (Python only) Fixed bug in cubic convolution C code that was checking the bounds of the output array instead of the input array, leading to interpolation errors when the input array was larger than the output array. Version 3.2.5: Change Rayleigh distance factor from 2 to 1 to provide more stable results. Version 3.2.5a: Update to manual only. Version 3.2.6: Added OMIT keyword to PROP_HEX_WAVEFRONT; Version 3.2.6a: fixed bug in Python version of PROP_HEX_WAVEFRONT (values of x and y aperture centers were being ignored); Version 3.2.7: Added INFLUENCE_FUNCTION_FILE parameter to PROP_DM to allow the user to specify their own actuator influence function; (Matlab) Changed PROP_RUN_MULTI to define its stack of images based on the image dimension returned by PROP_RUN, making it consistent with the IDL & Python versions.

Version 3.1.x

Bug fixes in the Python version, notably a memory leak in when using the Intel FFT. Astropy is now used for FITS i/o in both Python 2.7 and 3.x versions. The TABLE option now works in the Python version. In IDL and Python, the behavior of PROP_FFTW_WISDOM has changed; the routine will now compute FFTW wisdom for two pre-determined number of threads (different number for single and multiprocessing), and it uses the MEASURE test for determining wisdom (the option to specifiy the number of threads was removed). The wisdom is loaded during PROP_BEGIN rather than every time an FFT is computed. Fixed bug in the Matlab version of PROP_MAGNIFY. Fixed bug in Python 3.x version of PROP_DM that caused erroneous 1 pixel shift of DM surface map due to assumed integer divide being done as a floating divide. **Version 3.1.1** (Python) Fixed error in obscured Zernikes code (thanks to Gilles Orban de Xivry for reporting the error). **Version 3.1.2** (Python) Put in check in PROP_NOLL_ZERNIKES to see which version of scipy is present and load the factorial function from the correct module (thanks to Brandon Dube for pointing this out). **Version 3.1.3** (Python) The multiprocessing pool was not being closed, causing a memory leak. **Version 3.1.4** (Python) Fixed bug in PROP_IRREGULAR_POLYGON that caused crash when trying to draw a polygon beyond the maximum Y limit of the array. Version 3.1.5 (Python) Fixed bugs that caused some True/False optional parameters to be interpreted as True whenever the parameter was provided in the call to the routine, regardless of whether it was set to True or False; if the parameter was not provided, its correct default value was used. Changed the location where the flag files generated by PROP_USE_FFTI and PROP_USE_FFTW are written from the PROPER library directory to the user's home directory, along with the wisdom files generated by PROP_FFTW_WISDOM. This avoids needing to rerun those procedures anytime a new version of PROPER is installed.**Version 3.2 (IDL, Matlab)** Added the KEEP_THREADS option to PROP_RUN_MULTI to allow parallel processes to remain active and waiting rather than get destroyed once PROP_RUN_MULTI exits. Subsequent runs with KEEP_THREADS set will use these processes instead of creating new ones, saving overheads. PROP_FREE_THREADS was added to allow deletion of these processes.

Version 3.0d

This version fixes a bug in PROP_HEX_WAVEFRONT. According to the documentation, the segment numbering convention used when specifying aberrations on each segment does not change whether the central segment is set to be dark or not. However, in all versions of PROPER prior to this one, the numbering sequence would actually skip the central obscuration if it was dark. Thanks to Yinzi Xin for pointing this out. **Version 3.0d1 (Python only)**: Fixed a bug in PROP_PSD_ERRORMAP that was not correctly renormalizing the error map to the specified RMS value when the RMS keyword was set.

Version 3.0c

Bugs in the Python PROPER interpolation routines (PROP_CUBIC_CONV, PROP_MAGNIFY) that caused significant errors were fixed, along with a PROP_HEX_ZERNIKE bug (used by PROP_HEX_WAVEFRONT) that caused assignment of aberrations to the wrong segments. Thanks to Roser Parramon (GSFC) for reporting those errors. The output of the Python version of PROP_RUN_MULTI was inconsistent with that of PROP_RUN when the NOABS flag was set and the output was not complex, so this was fixed (thanks to Bryn Jeffries for reporting this and suggesting the fix). When using the Python version with PROP_RUN_MULTI and FFTW, a conflict would arise with multiple processes trying to write to the same FFTW wisdom file. This has been fixed (thanks to Bryn Jeffries for pointing this out). The Matlab version of PROP_MAGNIFY was fixed to prevent crashing when the output array size was not specified (thanks to A.J. Riggs at JPL for reporting this). The manual entry for PROP_HEX_WAVEFRONT was fixed to show the correct array ordering in Matlab for the Zernikes (that is, [segment_number, zernike_number]).

Version 3.0b

Bugs in the Python version were fixed (IDL to Python conversion errors). PROP_DM was causing a memory overload due to poor formatting of arrays. PROP_MAGNIFY, PROP_CUBIC_CONV, and PROP_RESAMPLE_MAP were fixed to match corrected cubic convolution interpolation parameters, along with cubic_conv*.c codes.

Version 3.0a

Bugs in the Matlab version of the examples were fixed, along with formatting. The manual entry for PROP_HEX_WAVEFRONT was modified to include a description of the segment numbering scheme. Legal notices were added to the codes.

Version 3.0

This is the introduction of the Python and Matlab versions of PROPER. PROP_FIT_ZERNIKES was changed to use least squares rather than iterative fitting, and a bug fixed that returned the fitted wavefront at the sampling of the shrunken wavefront rather than the original wavefront. MAX_FREQUENCY option was added to PROP_PSD_ERRORMAP to limit the spatial frequencies used in creating an aberration map. Fix to PROP_RECTANGLE to prevent crashing if the rectangle is beyond the array dimensions.

Version 2.0b,c

The orthographic projection of a tilted and/or rotated deformable mirror can now be included using PROP_DM and with the XTILT, YTILT, and ZTILT keywords. Version 2.0c fixes a bug in PROP_DM that caused N_ACT_ACROSS_PUPIL to be ignored (thanks to Gary Gutt for finding this).

Version 2.0a

IDL v8.4 introduced a new function called “lambda” that caused problems with the variable of the same name in PROP_RUN. This has been fixed.

Version 2.0

A new routine, PROP_RUN_MULTI, enables running multiple instances of a prescription in parallel. A number of routines have been optimized for improved speed. The FFTW interface has changed. Local optimization of a FFTW plan (i.e., wisdom) is not done automatically. If a local plan has not been generated using PROP_FFTW_WISDOM, then the default FFTW estimated plan will be used. The threaded FFTW routines are used by default. The FFTW interface now supports the Intel Math Kernel Library FFT. The PROP_MAGNIFY routine now handles complex values and a /QUICK switch has been added that provides faster results using IDL's built-in cubic convolution method. A number of bug fixes have been made (thanks to Lisa Poyneer and Christian Marois for pointing them out).

Version 1.1

The default behavior has changed regarding the wavefront phase offset that occurs during propagation. Prior to v1.1, a constant phase offset would be applied to the entire wavefront every time it was propagated some distance. For instance, if a wavefront was propagated over a distance equal to ½ the wavelength, then the phase at each point in the wavefront would be increased by a value of π radians. Because of phase wrapping, this offset would range over $\pm\pi$. This offset has caused confusion, especially since it was not documented. In most cases, it appears that not including this offset would be preferable, as it keeps the reference level of the wavefront at zero phase. Exceptions may include modeling systems in which the difference in length between two separate paths is important, like the arms of an interferometer. In versions 1.1 and later, the default action will be to NOT add the phase offset. In those situations where it is desired, the /PHASE_OFFSET switch can be specified in the call to PROP_RUN. Up to 22 Zernike polynomials are now available in PROP_HEX_WAVEFRONT, compared to the previous 11. Bug fixes were made.

Version 1.0

The CONIC and ASPHERIC lens options were removed. They produced unpredictable results, likely caused by the creation of a wavefront whose phase cannot be well fit with a sphere. They may be reintroduced at a later date. The PROP_HEX_APERTURE function was replaced with PROP_HEX_WAVEFRONT, which adds the capability to include aberrated segments. PROP_FIT_ZERNIKES now allows fitting an arbitrary number of unobscured Zernike polynomials. PROP_ZERNIKES can now multiply the wavefront by an amplitude error map comprising Zernike polynomials. Changed the default action in PROP_DM from smoothing to no smoothing, and changed the parameter switch from /NO_SMOOTH to /SMOOTH. The previous default was not sufficiently reliable. For those routines for which the amplitude error level was specified using the AMPLITUDE=*value* keyword, the meaning of *value* has changed. Previously, it specified the mean amplitude level. Now, it specifies the maximum amplitude level. The /NO_APPLY switch was added to some routines; this tells the routine to do whatever it does but not to modify the wavefront. This is useful for those instances when one wants the error map created by some routine without messing with the wavefront, for instance. For some routines, the /NOADD was replaced with /NO_APPLY. Users (especially under Windows) do not have to compile the external C routines to use the PROPER library. A version of the damped sinc interpolator is now provided written purely in IDL, and it will be used if the C version is not present. The C version, however, is about three times faster, though this is probably not an issue with most users.

Optical Propagation

Modeling the propagation of light through an optical system is usually done in one of two ways: (1) calculating the path of individual beams though the optical components (ray tracing), or (2) calculating the changes in the electromagnetic field as it travels (physical optics propagation, or POP). Ray tracing is typically used to design the system and determine its basic optical properties, such as magnification, aberrations, vignetting, etc. However, it cannot predict the effects of diffraction. Conversely, POP is completely concerned with how the electromagnetic wavefront is diffracted as it travels, but it does not determine things like aberration changes caused by a shifted component. Hybrid codes exist (e.g. MACOS) that combine the two methods – ray tracing to determine aberrations, beam sizes, and the like, and POP to compute the diffraction effects. Alternative propagation algorithms also exist that combine the two, such as the beam propagation method used in the commercial GLAD and Code V software.

Propagation Codes

A number of optical propagation codes exist. Many well-known commercial ray-tracing programs now include physical optics propagation (POP) calculations, including Code V and Zemax. Their POP systems are in addition to the simple far-field (Fraunhoffer) calculations they have always made. However, these packages are costly, have steep learning curves, and are not easy to integrate into exploratory modeling systems (e.g. wavefront control algorithm testing). Many POP programs have been developed by research institutions and companies for internal use and are typically not available to or designed for use by the public (BeamWarrior, MACOS, etc.). Free, publicly-available codes are few. Among these are LightPipes, which is a set of individual C programs that can be chained together to propagate a wavefront through a system. Another is Arroyo, a C++ library with an emphasis on atmospheric propagation for adaptive optics modeling.

Free and Easy Propagation with PROPER

PROPER is a library of optical propagation procedures and functions for the IDL (Interactive Data Language), Python, and Matlab environments. PROPER is intended for exploring diffraction effects in optical systems. It is a set of wavefront propagation tools – it is not a ray tracing system and thus is not suitable for detailed design work. An optical system is described by a series of PROPER library function and procedure calls within a user-written routine (hereafter called the *prescription*). These calls may be interleaved with additional user-written code, taking advantage of the wide range of mathematical, array processing, file input/output, and graphical routines available in the programming environment, providing a versatile system for modeling. PROPER includes procedures to create apertures and obscurations (circular, elliptical, rectangular, polygonal, and hexagonal arrays) and apply aberrations (low-order Zernikes polynomials, error maps defined by power-spectrum-density profiles, deformable mirrors, and user-defined wavefront error maps).

The PROPER routines are provided as source code, so you can see what they are actually doing, debug your (and perhaps the PROPER) procedures, and see ways to improve things (let the author know, please!). Because it is possible for someone to make modifications to the source code, it is important to grab the official, “clean” version from the proper-library.sourceforge.net website.

Propagating the PROPER Way

The PROPER routines implement common Fourier transform algorithms (angular spectrum & Fresnel approximation) to propagate a wavefront in near-field and far-field conditions. The procedures automatically determine which algorithm to use depending on the properties of the *pilot beam*, an on-axis Gaussian beam that is analytically traced through the system. This process follows the method described by Lawrence (Applied Optics and Optical Engineering, V. 11 [1992]), which is also used by other programs (e.g. ZEMAX and GLAD). The ZEMAX manual

(available with the ZEMAX demo from www.zemax.com) contains a concise chapter on physical optics propagation that describes this method as well. It is very briefly described here, without the mathematics, so that the user will be familiar with the constraints imposed by the PROPER routines.

Representing the Wavefront

A propagating electromagnetic field is described as a wavefront with varying phase and amplitude across its surface. A perfectly collimated, unaberrated beam has uniform phase across its wavefront as measured in a plane perpendicular to the direction of propagation, and the phase advances with the distance traveled. In this case, the surface of constant phase is a plane. After this beam passes through a lens or is reflected by a curved mirror, the phase is no longer constant across the wavefront if measured in a plane. The surface of constant phase is now curved, or in other words, the wavefront now has a radius of curvature (at least as measured near the optical axis, also called the paraxial region). For example, a planar wavefront (collimated beam) reflected by a concave parabolic mirror will become a spherical wavefront (converging beam) with some radius of curvature.

A wavefront's curvature can create problems when trying to represent the phase across it using a sampled grid in the computer. At some propagation distance the phase will change by more than one wave (2π radians) between two adjacent samples when measured in a planar grid (Figure 1). This causes numerical aliasing of the phase that leads to incorrect results. This problem can be reduced by the choice of propagators.

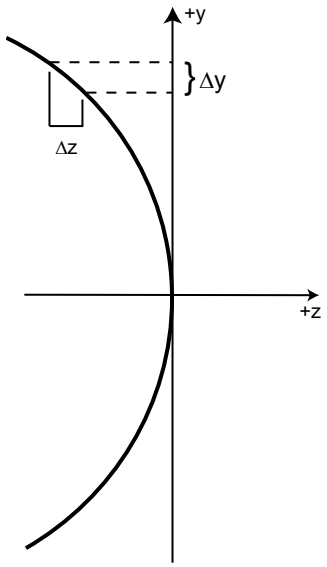


Figure 1. Schematic illustration of the problem of representing a wavefront with a curved surface of constant phase with a planar grid. In this example, a spherical wavefront, seen in cross-section, is expanding outward along the +z direction. The separation between sample points on the grid is Δy. Between two particular adjacent grid points in a plane, the phase of the wavefront changes by $2\pi\Delta z/\lambda$. If this phase change is greater than 2π radians, then the actual difference will be aliased to an erroneous value in the grid. Note that the degree of phase error depends on both the wavelength and the curvature of the wavefront.

Propagating in the Near and Far Fields

The goal of the methods used by PROPER is to choose a propagation algorithm that best accounts for the curvature of the wavefront to prevent phase aliasing. In an unaberrated system the wavefront is planar at the waist of the beam, which is at the pupil when the beam is collimated or at the focus when it is converging or diverging. Within some distance near the waist (the near field), the amount of wavefront curvature is low and the phase can be represented relative to a plane without significant aliasing. Away from the waist (into the far field), the wavefront gains curvature. At some point, it can no longer be sampled on a planar grid in the computer without aliasing.

The distance from the beam waist that defines where the near field ends and the far field begins must be chosen to prevent aliasing. There are various methods for choosing this distance, including using the Fresnel number. However, PROPER uses the Rayleigh distance, as described by Lawrence, which is determined using a Gaussian pilot beam that is analytically propagated through the system. A Gaussian beam is a convenient surrogate for the actual beam (at least in reasonably well-corrected systems) because its radius of curvature after propagation or after passing through a lens can be easily computed, as well as the diameter and location of its beam waist. These define the Rayleigh distance from the beam waist that specifies the boundary of the near and far fields, and thus which reference surface type (planar or spherical) is best used to minimize aliasing. Within the Rayleigh distance of the waist, the wavefront can usually be fit best with a planar reference surface, while outside it is best fit with a curved surface. The radius of a reference sphere is equal to the distance from the current position to the beam waist.

When propagating from one location to another within the near field, PROPER utilizes the angular spectrum algorithm (plane-to-plane, PTP, transform). To propagate from the beam waist in the near field to a location in the far, or vice-versa, the Fresnel method is used (spherical-to-waist (STW) or waist-to-spherical (WTS) wavefront transforms). Propagation between two points in the far field is done by first propagating to the beam waist (STW) and then to the new location (WTS). The scale between sample points in the wavefront grid remains constant when PTP is used. It changes proportionally with the distance propagated when STW or WTS is used, in order to maintain nearly-constant sampling of the beam. Thus, the scale will vary considerably during propagation though a multi-element system.

Conventions Assumed by the PROPER Routines

The PROPER routines propagate an electric field through an unfolded system in which all components lie on a straight line. A positive distance indicates that the field is being propagated forward through the system. A curved mirror will alter the phase distribution of the field but will not change the direction of the beam. The field phase also does not change sign due to reflection by a mirror, nor does the coordinate system. A positive value in an aberration map indicates advancement of the phase at that location relative to the default phase (i.e. the aberration map is added to the wavefront).

The PROPER coordinate system assumes that the first element of the wavefront array is in the lower left corner when the array is viewed. +X is towards the right and +Y is up. The wavefront origin is at the center of the array. Zernike aberrations are defined in azimuth from the +X axis to the +Y.

PROPER by default does not add a phase offset to the wavefront as it is propagated. Thus, the wavefront maintains a zero-phase reference value. This can be altered by specifying the *PHASE_OFFSET* switch in the call to **PROP_RUN** or **PROP_RUN_MULTI**. This may be important when modeling the separate arms of an interferometer with a path difference between the two.

Comparing PROPER Results with Those from Other Programs

The conventions used by PROPER may differ from those of other programs that do diffraction calculations, such as Zemax, Code V, GLAD, and others. So far, PROPER results have been compared to those only from Zemax. Some differences between the two are described here.

In Zemax (and most other optical design programs) an optical system is described by transmitting or reflecting surfaces that may be tilted or offset from the optical axis, forming a three-dimensional layout in which the beam can go back and forth through space (i.e. a reflection changes the direction of the beam). The PROPER routines instead assume an unfolded, linear layout, and a curved mirror does not change the direction of the beam but only its phase distribution.

Zemax utilizes ray tracing to compute the aberrations at each surface, be it a mirror or lens, including the effects of various surface shapes (conic, aspheric, anamorphic, etc.). PROPER routines assume that a lens introduces a purely radially-quadratic phase change. In Zemax, the phase of a wavefront changes sign after a reflection, while in PROPER the sign remains the same. Zemax uses ray tracing to compute the best-fit radius for the reference surface, while

PROPER simply assumes that it is equal to the distance from the current position of the wavefront in the system to the beam waist. Zemax draws binary obscurations (either 1 or 0), while PROPER draws obscurations with antialiased edges. In Zemax (and possibly other programs), the wavefront propagated using physical optical propagation methods includes a phase offset equal to the distance propagated (and is possibly phase wrapped). For instance, if the wavefront is propagated by ½ the wavelength, then the phase term at all points will include a π offset. PROPER (v1.1 and later) does not include this offset and maintains a zero phase wavefront offset (unless the *PHASE_OFFSET* switch is set in **PROP_RUN**).

Verification of PROPER's algorithms

As part of a NASA study on modeling coronagraphs, the accuracy of PROPER was compared for test cases against more mathematically rigorous computational methods, such as Rayleigh-Sommerfeld diffraction. The results show excellent agreement. A copy of the report is available at:

https://exoplanets.nasa.gov/exep/files/exep/krist_TDEM_milestone1_report_revised.pdf

Conventions Used in this Manual

IDL specific information is noted in blue, **Python** in red, and **Matlab** in green.

When the calling syntax of a PROPER routine is shown, optional parameters are enclosed in square brackets:

routine_name, required_parameter [, optional_parameter]

Routines may also have optional keywords and/or switches. A *keyword* is assigned a value; for example:

[, OPTIONAL_KEYWORD=value]

In IDL a *switch* is set like so:

[, /OPTIONAL_SWITCH]

which is equivalent to:

[, OPTIONAL_SWITCH=1]

or in Python

[, OPTIONAL_SWITCH=True]

In Matlab, switches are indicated by the name without any associated value. For example, to create an amplitude (rather than phase) error using Zernike polynomials normalized over a 0.05 meter radius, one might do this:

```
wf = prop_zernikes( wf, znum, zval, 'RADIUS', 0.05, 'AMPLITUDE' );
```

In this example, AMPLITUDE is a switch and RADIUS is an assigned-value keyword.

Be sure not to confuse optional parameters which are in italic square brackets with non-italic brackets that are part of the Matlab syntax for a vector of values.

Keywords and switches may be specified in any order in IDL and Python, but other parameters must be in the order shown in the calling syntax.

In Matlab all keywords and switches must come after the positionally-dependent parameters, and case is unimportant.

Python PROPER routines may return multiple parameters that can be accessed using a tuple. In such cases these are denoted using parentheses, which should not be confused with square brackets that indicate optional values:

(*wavefront, sampling*) = proper.prop_run(…)

Matlab routines may return multiple parameters in brackets. For example, one must always accept the wavefront structure return value from **prop_zernikes**:

```
wf = prop_zernikes( wf, znum, zval );
```

but the routine also returns the wavefront map that is the sum of the Zernikes, if the user wishes to use it:

```
[ wf, zmap ] = prop_zernikes( wf, znum, zval );
```

Getting Help from within the Environment

The calling sequence of a PROPER routine can be obtained within the execution environment using:

IDL: doc_library, 'prop_circular_aperture'

Python: help(prop_circular_aperture)

Matlab: help prop_circular_aperture

Installing and Setting Up the PROPER Package

IDL

Two IDL libraries need to be installed, PROPER and the IDL Astronomy User's Library, which provides the routines for FITS file input and output. The Astronomy Library can be downloaded from idlastro.gsfc.nasa.gov. Both libraries should be installed in directories of their own. If the user already has a directory dedicated to other IDL routine libraries, then they can be in subdirectories there. It is recommended that PROPER be installed separately by each user and not be in a system directory.

Except for one or two PROPER routines written in C that need to be compiled during installation, all of the procedures and functions in the PROPER and Astronomy User's libraries are written in IDL and distributed as source code. It is assumed that the user already has IDL installed and functioning properly.

Installing under UNIX/Linux/MacOS IDL

NOTE: The following instructions assume IDL version 6 or later has been installed. Earlier versions may require changes to the way the C routines are compiled and how the `IDL_PATH` environment variable is defined.

Go to step 2 if the Astronomy User's Library is already installed.

(1) Install the IDL Astronomy User's Library. Create a directory (or subdirectory in the user's IDL library directory) called astrolib and download the library into there. The library is distributed as a gzipped compressed tar file in one of two forms: all routines in one directory (astron.tar.gz) or routines in individual directories sorted by category (astron.dir.tar.gz) – it makes no difference in execution which is chosen. Uncompress and then untar the file:

```
gunzip astron.dir.tar.gz
tar xvf astron.dir.tar
```

(2) Create a directory (or subdirectory in the user's IDL library directory) for the PROPER library and download it into there. Uncompress and untar the file:

```
gunzip proper_idl_v3.2.tar.gz
tar xvf proper_idl_v3.2.tar
```

There is one file written in C that needs to be compiled on the user's machine. This contains code for damped sinc interpolation that cannot be efficiently performed in IDL. To compile it, enter the PROPER library directory, start up IDL, and issue the following commands:

```
.run prop_compile_c
prop_compile_c
```

This should compile the code on the most common UNIX and Linux platforms. If it does not, the user should consult the **MAKE_DLL** procedure description in the IDL Reference Manual and make the necessary changes described there to the code in `prop_compile_c.pro`. A version of this routine written in IDL is also provided and will be used if the user is unable to successfully compile the C one, but the IDL version is slower.

(3) Following the form of the examples below (replacing */directory/idl_libs* with the appropriate path), add the directories containing the Astronomy Users' and PROPER libraries to the IDL search path and define the PROPER_LIB_DIR environment variable. If both libraries are in subdirectories of the same directory, then that top level directory can be specified with a "+" preceding it to instruct IDL to search any of its subdirectories. If not, then each directory needs to be specified, separated by colons.

If using the C shell, add to the `.cshrc` file:

```
setenv IDL_PATH "+/directory/idl_libs:<IDL_DEFAULT>"
setenv PROPER_LIB_DIR "/directory/idl_libs/proper"
```

If using the Bourne shell, add to the `.bash_profile` file:

```
IDL_PATH="+/directory/idl_libs:<IDL_DEFAULT>"
PROPER_LIB_DIR=/directory/idl_libs/proper
export IDL_PATH PROPER_LIB_DIR
```

The user's environment needs to be reinitialized for these settings to take effect. The cleanest way is to log out and log back in again.

In IDL v6.2 and later you can set the IDL library path permanently from within IDL using the **PREF_SET** routine, rather than specifying it in your startup file:

```
pref_set, 'IDL_PATH', '+/directory/idl_libs:<IDL_DEFAULT>', /COMMIT
```

If you use this, you still need to define the PROP_LIB_DIR path in your .cshrc or .bash_profile file.

Installing under MS Windows IDL

NOTE: The author does not have access to a Windows machine running IDL, so everything here may be wrong or out of date!

The installation process under Windows is not as clean as it is on Unix-based systems. Currently, support for calling the external C routines provided with the PROPER library is not provided on Windows-based systems. There are two such routines: an interface to the FFTW or Intel math libraries to provide faster Fourier transforms than IDL's and a routine that does damped sinc interpolation. In the first case, PROPER will use the much slower IDL Fourier transform on Windows systems. In the second, a damped sinc interpolator routine written in IDL will be used that is slower than the C-based one. Functionality is not lost under Windows – execution is simply slower than it would be on a system using the fast FFTs and the C interpolator. NOTE: The author does not have access to IDL on a Windows machine, and thus cannot provide any support for such.

NOTE: The following instructions assume IDL version 6 or later has been installed. Earlier versions may require changes to the way the C routines are compiled and how the `IDL_PATH` environment variable is defined.

(1) Install the IDL Astronomy User's Library. Create a subdirectory in the user's IDL library directory. The library is distributed as a zip-compressed archive file, astron.zip. Using a zip uncompressor (or Window's built-in uncompressor), unzip the file's contents into the astrolib directory.

(2) Create a subdirectory for the PROPER library and download it into there. The library is distributed as a zip-compressed archive file. Using a zip uncompressor (or Window's built-in uncompressor), unzip the file's contents into the PROPER directory.

(3) Set the environment variables to point to the library directory. Go to **Start->Control Panel** and double-click on **System**. The **Systems Properties** window will pop up. Select **Advanced System Settings** and then the **Advanced** tab and click on the **Environment variables** button. Under **System Variables**, click on **New** to add a new variable. In the new window, in the **Variable name** box enter **PROPER_LIB_DIR** and in the **Variable value** box enter the full path to the library directory (e.g. **C:\RSI\user contributed\proper** or whatever your equivalent directory is). Press **Ok** to dismiss the window, then press **Ok** in the **Environment Variables** window, and kill the **Control Panel** window. If you are running IDL, save all your files and restart IDL so that the next instance of IDL can incorporate the new information.

Python

See the following sections on using the Intel versions of Numpy & Scipy and using FFTW or Intel's FFT for improved performance.

There are separate distributions of Python PROPER, one for Python v2.7 and another for Python v3.x. Python PROPER requires the following external packages: *numpy* (≥1.8), *astropy* (≥1.3), and *scipy* (≥0.14). Also, if you want to run some of the demos that display images to the screen, you will need *matplotlib* as well. Consult your software distribution's documents about obtaining these packages.

Create a directory (or subdirectory in the user's directory) for the PROPER library and download it into there. Choose the file appropriate for your Python version. Uncompress and untar the file, or in Windows extract the files from the corresponding zip archive:

Python 2.7

```
gunzip proper_v3.*_python_2.7.tar.gz
tar xvf proper_v3.*_python_2.7.tar
```

Python 3.x

```
gunzip proper_v3.*_python_3.x.tar.gz
tar xvf proper_v3.*_python_3.x.tar
```

Path setup

Option 1

This is the preferred method, as it also compiles some additional C codes that otherwise would not be compiled with the 2nd option.

To set up PROPER execute the following command in the terminal window while in the PROPER library directory (you may need to specify `python3` to use v3.x if your OS defaults to v2.7 and you are using the v3.x distribution):

```
python setup.py install
```

This will install Python PROPER in the user's site packages directory. These are:

Linux & MacOS X (non-framework)	`~/.local/lib/python*`
MacOS X (framework)	`~/Library/Python/X.Y`
Windows	`%APPDATA%\Python`

The user, rather than system, location used because compiled shared libraries and some temporary files are written to the PROPER directory.

Option 2

Alternatively, PROPER can be used by running the following commands within Python:

```
import sys
sys.path.insert(0, '/path/to/PROPER')
prop_compile_c()
```

Then, in your program import PROPER like so:

```
import proper
```

Matlab

Installing PROPER under Matlab is fairly straightforward. The PROPER library directory can be added to the search path using the *Set Path* option under the *Environment* tab in Matlab's integrated development environment. Once this is done the PROPER routines can be used in any Matlab program.

There is one file written in C that needs to be compiled on the user's machine. This contains code for damped sinc interpolation that cannot be efficiently performed in Matlab. To compile it, enter the PROPER library directory, start up Matlab, and issue the following command:

```
mex prop_szoom_c.c
```

This should compile the code on the most common platforms, as long as you have an appropriate C compiler installed.

Matlab array indices start with 1, while in IDL and Python they start at 0.

Increasing Speed in Python using Intel Optimized Numpy & Scipy

PROPER routines make use of Numpy and Scipy functions that are not optimized for multithreaded performance in the standard distributions of those packages. There are, however, Intel-specific versions of those that include highly-optimized code for vector and multithreaded calculations. Such functions include FFTs, exponentials, convolutions, etc. If you are going to do long-term PROPER calculations, you may benefit from installing these packages. They are all freely available.

Intel-optimized numpy and scipy packages exist. There are stand-alone versions available through pip, but they are older versions of numpy and scipy. The best choice is to use the conda packages. Check out https://anaconda.org/intel

Increasing Speed using FFTW or the Intel Math Library (IDL pre-v8.8.1, Python)

Note: IDL v8.8.1 uses by default the Intel Math Kernel Library, so you can ignore this section if using that version or later.

The PROPER routines rely heavily on the Fast Fourier Transform to propagate the wavefront. When repeatedly running through a prescription (modeling iterative wavefront control with a deformable mirror, for instance), much of the time is spent doing FFT. The default FFT algorithms in IDL (pre-v8.8.1) and Numpy are not very fast compared to some others, notably FFTW and the Intel Math Kernel Library (MKL). If the user will be making heavy use of PROPER, especially on large arrays, it is strongly advised that one of these packages be used. Neither is distributed with PROPER and must be downloaded separately. ***Matlab uses a version of FFTW that is fast, so there are no additional PROPER interfaces provided for other FFTs for it.*** Note that activating the FFTW or Intel FFTs will only affect PROPER routines.

FFTW (Fastest Fourier Transform in the West) is a free library containing efficient and optimized FFT functions. It is used by PROPER if the **PROP_USE_FFTW** routine has been called (which permanently sets PROPER to use FFTW, even over different sessions). In IDL the FFTW library is called via some interface C code and the **CALL_EXTERNAL** function. In Python the pyFFTW package is used. Calling **PROP_USE_FFTW** with the *DISABLE* switch will revert back to the default FFT (IDL's or Numpy's). Note that FFTW can be optimized further in IDL for a given machine by using "wisdom" (see the FFTW section below for more details).

The option is also provided to use Intel's MKL FFT. While it used to be available only for purchase, there is now a free version of MKL available from *software.intel.com*. It will be used if the **PROP_USE_FFTI** has been called to enable it. In IDL the Intel math library is called via some interface C code and the **CALL_EXTERNAL** function. Python PROPER uses the **ctypes** package to call the library. Calling **PROP_USE_FFTI** with the *DISABLE* switch will revert back to the default FFT (IDL's or Numpy's).

> NOTE: If you install the Intel-optimized version of Numpy/Scipy, you do not need to use PROP_USE_FFTI or PROP_USE_FFTW. The Intel-optimized FFT included in that package will be used.

As a demonstration of the improvement provided by the FFTW and MKL over IDL's built-in function, the FFTs of double-precision, complex arrays of different dimensions were computed with each on a dual Xeon (E5-2680, 2.7 GHZ, 8 cores/16 threads per CPU) workstation with 96 GB of RAM. Shown below are the elapsed time in seconds per array (based on 10 FFTs). Note that the speed of FFTW improves considerably for larger arrays when the plan (wisdom) is pre-computed. (Note: the no-wisdom FFTW actually is faster for 8192 arrays than for 4096 ones).

Dimension	IDL FFT	FFTW (no wisdom)	FFTW (wisdom)	Intel MKL FFT
1024 x 1024	0.095 s	0.028 s	0.018 s	0.002 s
2048 x 2048	0.804 s	0.115 s	0.028 s	0.012 s
4096 x 4096	2.820 s	0.592 s	0.079 s	0.046 s
8192 x 8192	9.956 s	0.365 s	0.274 s	0.183 s

NOTE: In Python, if you already have an installation of PROPER v3.1.4 or earlier and your have enabled Intel FFT or FFTW with PROP_USE_FFTI or PROP_USE_FFTW, or you have generated FFTW wisdom with PROP_FFTW_WISDOM, then you need to re-enable those and/or generate new wisdom. The flag and wisdom files used in v3.1.5 and later are stored in the user's home directory rather than the PROPER library directory.

PROPER and FFTW Wisdom Optimizations

> ***NOTICE:*** *The optimization ("wisdom") used by FFTW for a given system can change if a system library or the FFTW library is updated. The wisdom files produced with*

PROP_FFTW_WISDOM *using the previous libraries may not be appropriate for the new environment and could potentially cause significant degradations in speed or accuracy. A new wisdom file needs to be generated for each grid size after such updates.*

In IDL, FFT flag and wisdom files are stored in the PROPER library directory. If you delete that directory and install a new version of PROPER, you will need to rerun ***PROP_USE_FFTI*** *or* ***PROP_USE_FFTW*** *&* ***PROP_FFTW_WISDOM****. In v3.1.5 and later versions of PROPER for Python, these files are stored in the user's home directory and are not overwritten when a new version is installed.*

The FFTW routines do not automatically know the most efficient way to compute an FFT for a given situation (array size, threads, processor type, etc.). To do so, they first compute a few FFTs to decide which algorithm to use, creating a plan or, as it is sometimes described, "gathering wisdom". Once wisdom is attained for a particular situation, calls to the FFT routines will immediately result in the optimal method being used as long as the program is running. Once the program stops running, this wisdom will be lost unless it is saved. Unsaved wisdom would result in the overhead of determining the optimal method each time the program executes. **NOTE: It can take minutes to compute the wisdom for large grid sizes.**

Once the FFTW interface has been installed, wisdom for a given array size can be obtained using the PROPER routine **PROP_FFTW_WISDOM**. The wisdom is saved to a file in the user's home directory (Python) or the PROPER library directory (IDL). Separate wisdom must be obtained for each grid size, which will generate a separate wisdom file (for example in Python, ".proper_1024pix1threads_wisdomfile", ".proper_2048pix1threads_wisdomfile"). The only parameter to the routine is the grid size (one dimension of it); for example, to generate wisdom for a 2048 × 2048 grid:

IDL:
```
prop_fftw_wisdom, 2048
```

Python:
```
proper.prop_fftw_wisdom( 2048 )
```

The wisdom files are only valid for a particular system. When a new computer is used, a new wisdom file should be generated. Likewise, if new libraries are installed, it may also be necessary to recompile the FFTW package and delete the previous wisdom files.

Installing the PROPER FFTW interface in IDL

Windows

The FFTW interface to IDL is not supported under Windows in this version of PROPER (the author does not have IDL on a Windows machine and so cannot test it).

Unix/Linux/MacOS

NOTE: The FFTW library provided as a standard downloadable package (e.g., libfftw3 from Ubuntu) is usually not compiled to support threads or advanced hardware, so you need to compile the FFTW library from scratch.

The FFTW library can be downloaded from www.fftw.org. The PROPER interface requires FFTW Version 3 or later. Instructions are provided in the FFTW documentation regarding the library compilation and installation procedure, which consists of running a configuration script, making the library, and then installing it. You may want to install and link to a version of FFTW in your own personal directory rather than replacing the system's. How to do so is beyond the scope of this manual, but if you do this, be sure to modify PROP_COMPILE_FFTW.PRO in the PROPER library directory and change the link paths for the FFTW libraries.

When running the FFTW `configure` script, certain options should be enabled in order to work with PROPER and to optimize the speed of the FFT. By default, FFTW is compiled for double precision, which is what PROPER expects; PROPER will not work with a single-precision FFTW library. Shared libraries need to be created using the "`--enable-shared`" option (by default, shared libraries are not created). If the code will be running on a multiprocessor computer, then "`--enable-threads`" needs to be specified as well. If the code will be running on processors that support SSE2 extensions, then "`--enable-sse2`" should also be included (plain SSE extensions will not help because they do not support double precision reals). An example is:

```
./configure --enable-threads --enable-shared --enable-sse2
```

The FFTW library functions are called by a few C routines provided in the PROPER distribution and which are accessed using the **PROP_FFTW** procedure (which only accesses the FFTW complex-to-complex transforms). These C routines must be compiled prior to use using the **PROP_COMPILE_FFTW** command (note that this only compiles the interface routines, not the FFTW library itself).

On the IDL command line, simply type (while in the PROPER library directory)

```
prop_compile_fftw
```

Next, the PROPER_FFTW_WISDOM_FILE environment variable needs to be defined. This specifies the name of a file (including the full directory path) that will be created by the FFTW library to contain the "wisdom" accumulated for optimizing the FFT for a particular system (see the next section for more on FFTW wisdom). The directory that will contain this file must be writable by the user.

If using the C shell, add to the `.cshrc` file:

```
setenv PROPER_FFTW_WISDOM_FILE "/directory/filename"
```

If using the Bourne shell, add to the `.bash_profile` file:

```
PROPER_FFTW_WISDOM_FILE=/directory/filename
export PROPER_FFTW_WISDOM_FILE
```

The user's environment needs to be reinitialized for these settings to take effect. The cleanest way is to log out and log back in again.

The PROPER routines also need to be told to use the FFTW library FFTs rather than IDL's. This is done by calling **PROP_USE_FFTW** in IDL:

```
prop_use_fftw
```

which will create a tiny file in the directory pointed to by PROPER_LIB_DIR (the user must have write permission in that directory). Whenever **PROP_RUN** is called, it checks for that file to determine which FFT to use. This action is permanent until it is disabled:

```
prop_use_fftw, /disable
```

Installing the PROPER Intel FFT Interface in IDL

The Intel MKL FFT library functions are accessed via C routines using the **PROP_FFTI** procedure (which only accesses the FFT complex-to-complex transforms). These routines must be compiled prior to use using the **PROP_COMPILE_FFTI** command. With the MKL installed, on the IDL command line simply type (while in the PROPER library directory)

```
prop_compile_ffti
```

Once these routines have been successfully compiled, PROPER needs to be instructed to use them. To do so, on the IDL command line type:

```
prop_use_ffti
```

To disable using them, use:

```
prop_use_ffti, /disable
```

NOTE: If both the FFTW and MKL FFT routines are enabled, then the MKL routine will be used.

Installing the PROPER FFTW interface in Python

NOTE: If you are installing a new version of PROPER and already have a version 3.1.4 or earlier, you will need to run PROP_USE_FFTW and rerun PROP_FFTW_WISDOM, since the files associated with those will not exist in the new distribution on your machine.

pyFFTW is a wrapper for FFTW in Python. More information about this package and installation instruction can be obtained from *https://www.github.com/pyFFTW/pyFFTW*. **PROP_USE_FFTW** enables pyFFTW in PROPER:

```
proper.prop_use_fftw()
```

Once FFTW is enabled, the wisdom file for the appropriate array size can be created using **PROP_FFTW_WISDOM**. For example, to generate wisdom file for a 2048 × 2048 grid:

```
proper.prop_fftw_wisdom(2048)
```

Note that the wisdom files are only valid for a particular system. When a new computer is used, a new wisdom file should be generated. The FFTW interface first checks whether the wisdom file for a specific gridsize exists in the user's home directory (e.g. for an array size of 512 it checks if ".512pix_wisdomfile" exists). If the wisdom file exists, it is imported. The wisdom files are saved in the user's home directory.

Installing the PROPER Intel MKL FFT interface in Python

NOTE: Anytime you install a new version of PROPER, you will need to run PROP_USE_FFTI, since the file associated with those will not exist in the new distribution on your machine.

Python PROPER uses the *ctypes* package to import the Intel library functions. **PROP_USE_FFTI** enables the MKL FFT libraries in PROPER. The libraries loaded are:

```
Linux:    ctypes.cdll.LoadLibrary('libmkl_rt.so')
MacOS:    ctypes.cdll.LoadLibrary("libmkl_rt.dylib")
Windows:  ctypes.cdll.LoadLibrary("mk2_rt.dll")
```

Some benchmarks

To compare run times for different numbers of parallel processes and against different languages, a complex PROPER prescription was executed. It called the same sequence of PROPER routines in IDL, Python, and Matlab. These included drawing shapes, using deformable mirrors, reading optical surface error maps, etc. In each language the prescription was run in parallel using PROP_RUN_MULTI for a varying number of wavelengths and a combination of 2048 x 2048 and 4096 x 4096 grid sizes.

All runs were done on a Ubuntu 18.04 system with dual Xeon 6240 Gold CPUs (2.6 GHz base clock, 3.9 GHz turbo clock). Each CPU has 18 physical cores plus 18 hyperthreads, for a total of 72 available threads). It had 256 GB RAM (though the runs used only a fraction of that).

PROPER's interface to the optimized Intel FFT was used for IDL. More than 36 wavelengths could not be run at once in IDL, with crashes occuring in the subprocesses due to allocation errors (solution unknown).

The Intel distributions of Numpy and Scipy were used with Python v3.6 (Python 2.7 was about 20% slower for some reason with the same setup). A test using the interface to FFTW with pre-generated wisdom files showed it to be about 5% faster than the Intel FFTs for these particular conditions.

The IDL and Matlab times do not include overheads for process allocation.

Matlab includes FFTW by default.

Mean time (sec) per wavelength
(total time / N_λ)

Nλ	IDL v8.6.1	Python v3.6	Matlab R2019a
1	49.0	62.0	10.8
9	8.7	10.5	8.7
27	5.5	4.7	3.3
36	5.4	4.1	2.7
46		3.5	2.7
66		3.1	2.4

Note how well Matlab is optimized by default. It has multithreaded operators and uses FFTW with well-chosen settings by default. IDL also has multithreaded operators and basic functions, but they are not as well optimized (IDL's FFT is abysmally slow, necessitating the need for PROPER's Intel FFT interface). Python is slowest for a single processes and does not surpass IDL until 27 or more wavelengths are run in parallel. Note that Python does not have any intrinsic multithreaded operators or functions, nor does the standard Numpy & Scipy distribution.

PROPER Routines by Category

Note: Some "behind-the-scenes" routines are not listed here

Prescription Definition and Execution Routines

prop_begin	Define initial beam properties and create wavefront array
prop_define_entrance	Define entrance pupil and renormalize wavefront to unit intensity
prop_end	Terminate propagation sequence
prop_end_savestate	Terminate saving state information and clean up state files
prop_free_threads	(IDL, Matlab) Release processes created by PROP_RUN_MULTI
prop_init_savestate	Initialize state saving system
prop_is_statesaved	Check if state applicable to current run exists
prop_run	Execute a prescription
prop_run_multi	Execute multiple instances of a prescription in parallel
prop_state	Read in saved state if it exists, else save current state

Wavefront Phase and Amplitude Modifying Routines

The primary intended use of the PROPER package is simulating the sensing and control of wavefront errors. Thus, there are a number of routines that involve modifying the phase and amplitude of the wavefront (aperture and obscuration mask routines are listed in another category):

prop_add_phase	Add a phase error map to the current wavefront
prop_divide	Divide the wavefront amplitude by a value or 2D array
prop_dm	Modify the wavefront using a deformable mirror
prop_errormap	Read in an error map from a file and apply it to the current wavefront
prop_hex_wavefront	Create hexagonal array of aberrated hexagonal segments
prop_lens	Alter wavefront curvature due to a lens or mirror
prop_multiply	Multiply the wavefront amplitude by a value or 2D array
prop_propagate	Propagate the wavefront a specified distance
prop_psd_errormap	Create an error map defined by a power spectral density profile
prop_zernikes	Add phase aberrations defined by Zernike polynomials

Query Functions

A number of functions are available for determining the current characteristics of the wavefront being propagated and the current propagation state. Note that the results from some of these routines are only valid at the focus of an unaberrated system, as indicated in their function descriptions in the PROPER Routines Reference Section.

prop_get_amplitude	Return the amplitude portion of the current wavefront
prop_get_beamradius	Get the current radius of the pilot beam
prop_get_distancetofocus	Get the distance from the current location to the focus (beam waist)
prop_get_fratio	Get the current focal ratio of the pilot beam
prop_get_gridsize	Get the size of the wavefront array grid
prop_get_nyquistsampling	Get the Nyquist sampling criterion for the current wavefront
prop_get_phase	Return the phase portion of the current wavefront
prop_get_refradius	Get the current reference surface radius
prop_get_sampling	Get the sampling of the wavefront in meters
prop_get_sampling_arcsec	Get the sampling of the wavefront in arcseconds
prop_get_sampling_radians	Get the sampling of the wavefront in radians
prop_get_wavefront	Return the complex-valued wavefront array
prop_get_wavelength	Get the wavelength of the propagation in meters
prop_is_statesaved	Check if state applicable to current run is saved

Shape Drawing, Aperture & Obscuration Pattern Routines

Routines are provided that return an image containing a filled shape or multiply the wavefront by a mask with a certain shape. The edges of the shapes are antialiased (the value of a pixel along the edge of a shape is proportional to the area of the pixel covered by that shape, ranging from 0.0 to 1.0).

prop_circular_aperture	Multiply the wavefront by a circular aperture (dark outside)
prop_circular_obscuration	Multiply the wavefront by a circular obscuration (dark inside)
prop_ellipse	Return an image containing a filled ellipse
prop_elliptical_aperture	Multiply the wavefront by an elliptical aperture (dark outside)
prop_elliptical_obscuration	Multiply the wavefront by an elliptical obscuration (dark inside)
prop_hex_wavefront	Create hexagonal array of aberrated hexagonal segments
prop_irregular_polygon	Return an image containing a filled convex irregular polygon
prop_polygon	Return an image containing a filled polygon
prop_rectangle	Return an image containing a filled rectangle
prop_rectangular_aperture	Multiply the wavefront by a rectangular aperture (dark outside)
prop_rectangular_obscuration	Multiply the wavefront by a rectangular obscuration (dark inside)
prop_rounded_rectangle	Return an image containing a rounded rectangle aperture mask
prop_set_antialiasing	Define subsampling used to antialias shape edges

Error Map Input & Output Routines

IDL has a wide array of file input/output routines, and the Astronomy User's Library has a number of procedures for reading and writing images in commonly used scientific formats (e.g. FITS). The PROPER routines listed here are expressly intended for reading and writing wavefront error maps.

prop_errormap	Read an error map from a FITS file and apply it to the wavefront
prop_readmap	Read an error map from a FITS file and return it as an image
prop_writemap	Write out a phase or amplitude error map to a FITS file

Utility Routines

prop_compile_ffti	Compile the IDL interface to the Intel MKL library
prop_compile_fftw	Compile the IDL interface to the FFTW library
prop_fftw_wisdom	Generate an FFTW wisdom file for a given grid size
prop_fit_zernikes	Fit Zernike polynomials to an error map
prop_magnify	Resize an image using damped sinc interpolation
prop_noll_zernikes	Generate a table of Noll-ordered Zernike polynomials
prop_print_zernikes	Print a table of Noll-ordered Zernike polynomials
prop_radius	Return array of distances of wavefront array elements from optical axis
prop_resamplemap	Resample an error map using cubic convolution interpolation
prop_rotate	Rotate and/or shift an image via interpolation
prop_shift_center	Shift the center of an array to the lower left of the array
prop_use_ffti	Enables/disables use of Intel MKL FFT routines
prop_use_fftw	Enables/disables use of FFTW routines

Other Routines

prop_8th_order_mask	Multiply the wavefront by an 8^{th}-order occulting mask
prop_pixellate	Integrate a sampled image onto square pixels

Defining and Running a PROPER Prescription of a System

Definition Requirements

The convention used in this manual is that a *prescription* is a user-written procedure that contains a series of calls to PROPER library routines that describe propagation through an optical system. The prescription must follow all of the rules of the language for a routine definition (e.g. in IDL it must begin with a "`pro routine_name`" declaration and conclude with an **end** statement, comments begin with a semicolon, etc.). The routine can also take advantage of all of the facilities that the language provides, like math, file input/output, and graphics functions. The prescription itself is not intended to be called directly by the user, but rather only by **PROP_RUN** or **PROP_RUN_MULTI**.

The fundamental data set that the PROPER routines modify is the complex-valued wavefront array that is created and initialized by calling **PROP_BEGIN**. As described in a following section, this array is contained within a structure that includes information regarding the state of the wavefront. The wavefront array should only be accessed through PROPER routines; direct modification of it by the user could result in unexpected errors, and the structure members may change names or purpose between different versions of the software.

At a minimum, the prescription must include the following procedure calls (see the PROPER Routines Reference section for more details on each routine):

PROP_BEGIN
: Initiates the PROPER routines. The user passes the wavelength at which to propagate, the initial diameter of the beam in meters, the size of the computational grid (*n*, an integer that is preferably a power of two, so that the wavefront array has dimensions of *n* by *n* elements), and the ratio of the initial beam diameter to the grid diameter (see the section on sampling to see how this affects the propagation results). It then creates a wavefront array with all elements set to 1.0 (uniform amplitude, no phase error).

PROP_DEFINE_ENTRANCE
: Establishes that the current wavefront array represents the wavefront at the entrance aperture. This is typically called immediately after the blank wavefront initialized by **PROP_BEGIN** is multiplied by the entrance aperture pattern. The wavefront array is renormalized to have a total intensity of 1.0.

PROP_END
: Concludes the propagation sequence by computing the intensity of the wavefront array (or returning the complex amplitude if the *NOABS* switch is set); unless the amplitude of the wavefront was reduced during propagation (via additional apertures or modification of the amplitude), the total intensity should be 1.0. **PROP_END** also returns the sampling of the result in meters, which must be passed back to **PROP_RUN** as described later.

IDL

Prescription routine declaration

All IDL prescription routines must have the same parameter declaration sequence (no more nor fewer parameters are allowed):

```
pro routine_name, wavefront, wavelength, gridsize, sampling, PASSVALUE=variable
```

Here *routine_name* is name of the prescription routine and can be anything that is a valid IDL routine name. The routine name and the root name of the file containing the routine must be the same (e.g. if the routine name is `telescope` then its code must be in a file called `telescope.pro`). *wavefront* is a variable in which the final propagation result is returned as a two-dimensional, double-precision array. *wavelength* is the wavelength *in meters* at which to propagate (**NOTE: even though the wavelength provided to PROP_RUN or PROP_RUN_MULTI is in microns, it is internally converted to meters before being passed to the prescription**). *gridsize* is the size of the wavefront array grid (*gridsize* by *gridsize* elements), which must be a power of two (e.g., 512, 1024, 2048). *sampling* is a variable in which the sampling in meters is to be returned to **PROP_RUN** (this value is returned by **PROP_END** and the user is responsible for ensuring that it is passed back). The *PASSVALUE* keyword is described below.

Here are some example prescription routine declarations:

```
pro telescope, wavefront, wavelength, gridsize, sampling, PASSVALUE=optval
pro camera, psf, lambda, n, sampling, PASSVALUE=passvalue
```

Prescription return values

The prescription function must finish by calling **PROP_END** to extract the wavefront array (either real or complex-valued, depending on the prescription) from the wavefront structure and obtain its sampling in meters. Both are then returned; for example, something like this:

```
...
prop_end, wavefront, sampling_m
return
end
```

Optional prescription parameters using PASSVALUE

The *PASSVALUE* keyword must be included in the prescription routine declaration, but its use is optional when calling the routine. It allows the user to pass an additional parameter to the prescription by assigning a value to the keyword when calling **PROP_RUN** (common blocks may be used instead, but these do not work when running in parallel using **PROP_RUN_MULTI**). In the prescription routine declaration, *PASSVALUE* is set to a variable that can be accessed within the routine. Multiple values may be passed via this method using a structure. Here is an example prescription declaration:

```
pro simple_prescription, wavefront, wavelength, gridsize, sampling, PASSVALUE=optval
```

In this case, the prescription will access the passed values through the variable *optval*. The name of the variable can be any valid IDL name, *optdiam* rather than *optval*, for instance. If *PASSVALUE* was not assigned a value in the call to **PROP_RUN**, then its corresponding variable in the prescription will be undefined. If, for instance, the optional passed value is not used in the prescription, the *PASSVALUE* keyword can be omitted in the call to **PROP_RUN,** e.g.:

```
prop_run, 'simple_prescription', psf, lambda, gridsize
```

Before trying to use a passed value, the IDL n_elements function can be used to check if the assigned variable actually has been assigned a value. Here is an altered snippet of the example above that allows the diameter of the entrance aperture to be optionally defined by *optval*, otherwise it defaults to 1 meter:

```
…
if ( n_elements(optval) eq 0 ) then diam = 1.0d else diam = optval
…
```

This routine can then be passed the diameter (2.0 meters in this case) during the call to **PROP_RUN**:

```
prop_run, 'simple_prescription', psf, lambda, sampling, PASSVALUE=2.0
```

Suppose that both the entrance aperture diameter and the focal ratio of the lens should be optionally passed to the prescription. This can be done using a multiple-member structure:

```
pars = {diam:2.0d, focal_ratio:25.0d}
prop_run, 'simple_prescription', psf, lambda, sampling, PASSVALUE=pars
```

The code in `simple_prescription` would then look like this:

```
…
if ( n_elements(optval) eq 0 ) then begin
        diam = 1.0d
        focal_ratio = 15.0d
endif else begin
        diam = optval.diam
        focal_ratio = optval.focal_ratio
endelse
```

Matlab

Prescription routine declaration

All Matlab prescription functions must have the same parameter declaration sequence (no more nor fewer parameters are allowed):

```
function [wavefront, sampling] = routine_name( wavelength, gridsize, optval )
```

routine_name is the name of the prescription function and can be anything that is a valid Matlab name. The name and the root name of the file containing the routine must be the same (e.g. if the routine name is `telescope` then its code must be in a file called `telescope.m`). *wavelength* is the wavelength *in meters* at which to propagate (**NOTE: even though the wavelength provided to PROP_RUN or PROP_RUN_MULTI is in microns, it is internally converted to meters before being passed to the prescription**). *gridsize* is the size of the wavefront array grid (*gridsize* by *gridsize* pixels). The *optval* optional parameters are described below.

Here are some example prescription routine declarations:

```
function [wavefront, sampling] = telescope( wavelength, gridsize, optval )
function [psf, sampling] = camera( lambda, n, other_pars )
```

Prescription return values

The prescription function must finish by calling **PROP_END** to extract the wavefront array (either real or complex-valued, depending on the prescription) from the wavefront structure and obtain its sampling in meters. These are returned as the *wavefront* array and wavefront *sampling* in meters.

Optional prescription parameters

The third parameter in a prescription's declaration is for receiving optional parameters that were passed to **PROP_RUN** or **PROP_RUN_MULTI** using the *PASSVALUE* keyword/value pair. Multiple values may be passed via this method using a structure. Here is an example prescription declaration:

```
function [psf, sampling] = simple_prescription( wavelength, gridsize, pars )
```

In this case, the prescription will access the optional parameter through the variable `pars`. The name of the variable can be any valid Matlab name. If the optional parameter is not provided in the call to **PROP_RUN** or **PROP_RUN_MULTI**, then its corresponding variable in the prescription routine will be undefined.

The optional parameters are provided to the prescription in **PROP_RUN** like so:

```
pars.diam = 2.0;
pars.focal_ratio = 25.0;
psf = prop_run( 'simple_prescription', lambda, sampling, 'PASSVALUE', pars );
```

Note that the *PASSVALUE* keyword is used to indicate that the next parameter in the call is a structure to pass to the prescription. If the optional parameter is not used in the prescription, it can be omitted in the call to **PROP_RUN,** e.g.:

```
[psf, sampling] = prop_run( 'simple_prescription', lambda, gridsize );
```

Before trying to use an optional passed value, the Matlab `nargin` function can be used to check if three parameters have been provided to the prescription (one less than was provided to **PROP_RUN** because the filename is not passed to the prescription). Here is an altered snippet of the example above that allows the diameter of the entrance aperture to be optionally defined by `optval`, otherwise it defaults to 1 meter:

```
if nargin ~= 3
        diam = 1.0;
        focal_ratio = 14.0;
else
        diam = pars.diam;
        focal_ratio = pars.focal_ratio;
end
```

A Note on Matlab Parameter Passing

Matlab does not have named keywords or switches as IDL or Python does, so the Matlab PROPER routines have a mixture of positionally-dependent parameters (required and optional), along with parameter name/value pairs. For example, PROP_ZERNIKES has three positionally-depended parameters along with some named optional parameters:

wavestruct = **prop_zernikes**(*wavestruct, zernike_num, zernike_val [, 'AMPLITUDE']*
[, 'EPS', obscuration_ratio] [, 'NAME', string] [, 'NO_APPLY'] [, 'RADIUS', value]);

[*wavestruct, map_array*] = **prop_zernikes**(*wavestruct, zernike_num, zernike_val [, 'AMPLITUDE']*
[, 'EPS', obscuration_ratio] [,' NAME', string]
[, 'NO_APPLY'] [, 'RADIUS', value]);

The three required parameters are the wavefront structure (wavestruct), an array of Zernike polynomial indices (zernike_num), and a corresponding array of Zernike polynomial coefficients (zernike_val). If obscured Zernike polynomials are to be used, the central obscuration ratio must be provided using the EPS keyword and value, but must do so only after the first 3 parameters are given. Some of these parameters are switches (flags) that do not need to be set to any value. For example, to specify that the aberration map should be returned but not applied to the wavefront, one might have a call that looks like this:

```
[wf, zermap] = prop_zernikes( wf, znum, zval, 'NO_APPLY' );
```

Python

Prescription routine declaration

All Python prescription functions must have the same parameter declaration sequence (no more nor fewer parameters are allowed):

```
def routine_name( wavelength, gridsize, optval ):
```

routine_name is the name of the prescription function and can be anything that is a valid Python name. The name and the root name of the file containing the routine must be the same (e.g. if the routine name is *telescope* then its code must be in a file called *telescope.py*). *wavelength* is the wavelength *in meters* at which to propagate (**NOTE: even though the wavelength provided to PROP_RUN or PROP_RUN_MULTI is in microns, it is internally converted to meters before being passed to the prescription, for historical reasons**). *gridsize* is the size of the wavefront array grid (*gridsize* by *gridsize* pixels), which must be a power of two (e.g., 512, 1024, 2048). The *optval* optional parameter is described below in the *PASSVALUE* section.

Here are some example prescription routine declarations:

```
def telescope( wavelength, gridsize, optval ):
def camera( lambda, n, other_pars ):
```

Prescription return values

The prescription function must finish by calling **PROP_END** to extract the wavefront array (either real or complex-valued *numpy* array, depending on the prescription) from the wavefront structure and obtain its sampling in meters. Both are then returned as a tuple; for example, something like this:

```
…
wavefront, sampling_m = proper.prop_end( wavefront_struct )
return ( wavefront, sampling_m )
```

Optional prescription parameters using PASSVALUE

The third parameter in a prescription's declaration is for receiving an optional parameter that may be provided in the call to **PROP_RUN** or **PROP_RUN_MULTI** via the *PASSVALUE* keyword. Multiple values may be passed using a dictionary (or list of dictionaries for **PROP_RUN_MULTI**). Suppose we have a prescription in which the diameter of a circular mask can optionally be specified at run time instead of using its default value. Its declaration might look like this:

```
def simple_prescription( wavelength, gridsize, PASSVALUE={'mask_radius':0.02} ):
```

Running this prescription with the mask radius set to 0.01 would then be done like so, using *PASSVALUE*:

```
(psf, sampling) = proper.prop_run( 'simple_prescription', lambda, gridsize,
                        PASSVALUE={'mask_radius':0.01} )
```

In the prescription the value of the parameter passed by *PASSVALUE* is accessed like so:

```
wavefront = proper.prop_circular_aperture( wavefront, PASSVALUE['mask_radius'] )
```

Note that values can only be passed down to the prescription via this method, and not returned back up to **PROP_RUN**.

If the optional parameter is not provided in the call to **PROP_RUN** or **PROP_RUN_MULTI** via the *PASSVALUE* variable then a default value should be defined. For instance, the optional passed value is not passed**,** e.g**.**:

```
(psf, sampling) = proper.prop_run( 'simple_prescription', lambda, gridsize )
```

In this case, the default value for the mask radius will be used.

Suppose instead we wanted to be able to specify the inner and outer radii of a mask. We could pass these two values in a dictionary. Our modified routine would then have the following definition:

```
def simple_prescription( wavelength, gridsize, PASSVALUE={'inner_radius':0.01,
       'outer_radius':0.8} ):
```

and running it with the inner and outer mask radii specified would look like this:

```
(psf, sampling) = proper.prop_run( 'simple_prescription', lambda, gridsize,
       PASSVALUE={'inner_radius':0.005, 'outer_radius':0.5} )
```

Fundamental PROPER Routines: PROP_LENS & PROP_PROPAGATE

The two procedures that will do most of the work in typical PROPER prescriptions are **PROP_PROPAGATE** and **PROP_LENS**. **PROP_PROPAGATE** causes the wavefront to travel a specified distance in meters. It decides, depending on the properties of the pilot beam, which propagation method to use (angular spectrum or Fresnel). Its calling sequence is

IDL **prop_propagate**, *wavefront, dz [, label] [, /TO_PLANE]*

Python **proper.prop_propagate(** *wavefront, dz [, label] [, TO_PLANE=True/False]* **)**

Matlab *wavefront_out* = **prop_propagate(** *wavefront_in, dz [, 'SURFACE_NAME', label] [, 'TO_PLANE']* **);**

where *wavefront* is the variable containing the wavefront structure, *dz* is the distance to propagate the wavefront, and the optional *label* will be printed if specified (e.g. "Propagating to secondary mirror"). The propagation distance is positive to proceed forward through the system or is negative to back up. Note that it makes no sense to propagate the wavefront before applying an aperture function; otherwise the edge of the array will act as the edge of the aperture. PROPER has a number of routines that can create and apply aperture masks of various shapes.

As the name suggests, **PROP_LENS** alters the phase of the wavefront as would a thin lens or curved mirror. Its calling sequence is

IDL **prop_lens**, *wavefront, focal_length [, label]*

Python **proper.prop_lens(** *wavefront, focal_length [, label]* **)**

Matlab *wavefront_out* = **prop_lens(** *wavefront_in, focal_length [, label]* **);**

where *wavefront* is the variable containing the wavefront structure, *focal_length* is the focal length of the lens or mirror in meters, and the optional parameter *label* is a string containing the name of the lens. If *label* is defined, for instance, to be 'primary mirror', then the message "Applying lens at primary mirror' will be printed as the lens is applied to the wavefront (unless the *QUIET* switch was set in the call to **PROP_RUN**). A convex lens (equivalent to a concave mirror) will have a positive focal length, while a concave lens (convex mirror) has a negative one.

A Simple Example Prescription

Here is an example prescription that simply creates a 1 meter diameter circular aperture and lens and propagates the wavefront to the focus. The incoming wavefront is assumed to be collimated (uniformly flat). The prescription returns the modulus-squared of the wavefront at the focus (e.g. the point spread function). In this prescription, the focal length of the lens is 15 meters. The entrance pupil diameter (e.g. the initial beam size) is set to occupy half of the wavefront grid diameter (`beam_ratio=0.5`). The code is included in the *examples* subdirectory in the PROPER directory.

IDL:

```
pro simple_prescription, wf, wavelength, gridsize, sampling, PASSVALUE=optval

diam = 1.0d                              ;-- entrance aperture diameter in meters
focal_ratio = 15.0d
focal_length = diam * focal_ratio
beam_ratio = 0.5

prop_begin, wf, diam, wavelength, gridsize, beam_ratio

prop_circular_aperture, wf, diam/2   ;-- 0.5 meter radius circular aperture
prop_define_entrance, wf
prop_lens, wf, focal_length

prop_propagate, wf, focal_length

prop_end, wf, sampling

return
end
```

Python:

```
import proper


def simple_prescription(wavelength, gridsize):

    diam = 1.0
    focal_ratio = 15.0
    focal_length = diam * focal_ratio
    beam_ratio = 0.5

    wfo = proper.prop_begin(diam, wavelength, gridsize, beam_ratio)

    proper.prop_circular_aperture(wfo, diam/2)
    proper.prop_define_entrance(wfo)
    proper.prop_lens(wfo, focal_length)

    proper.prop_propagate(wfo, focal_length)

    (wfo, sampling) = proper.prop_end(wfo)

    return (wfo, sampling)
```

Matlab:

```
function [wf, sampling] = simple_prescription( wavelength, gridsize, optval )

diam = 1.0;
focal_ratio = 15.0;
focal_length = diam * focal_ratio;
beam_ratio = 0.5;

wf = prop_begin( diam, wavelength, gridsize, beam_ratio );

wf = prop_circular_aperture( wf, diam/2 );
wf = prop_define_entrance( wf );
wf = prop_lens( wf, focal_length );

wf = prop_propagate( wf, focal_length );

[wf, sampling] = prop_end( wf );

end
```

Running the Prescription

A prescription is executed with the **PROP_RUN** or **PROP_RUN_MULTI** command. Note that every time these are called they compile the prescription code file. This may lead to unexpected results if the user is editing the prescription's code while that prescription is being executed multiple times by looping calls to **PROP_RUN**; the next iteration through the loop will compile the latest version of the file. **PROP_RUN** will be described here. See the section on **PROP_RUN_MULTI** for details on running a prescription in parallel.

The calling sequence for **PROP_RUN** is:

IDL: **prop_run**, *prescription*, *result, wavelength, gridsize [, sampling_m]*
[, PASSVALUE=value] [, /PHASE_OFFSET] [, /PRINT_INTENSITY]
[, /QUIET] [, /TABLE] [, /VERBOSE]

Python: (*result, sampling_m*) = **proper.prop_run**(*prescription*, *wavelength, gridsize*
[, PASSVALUE=value] [, PHASE_OFFSET=True/False]
[, PRINT_INTENSITY=True/False] [, QUIET=True/False]
[, TABLE=True/False] [, VERBOSE=True/False])

Matlab: *result = - OR -*
[*result, sampling_m*] =
prop_run(*prescription*, *wavelength, gridsize [, 'PASSVALUE', value]*
[, 'PHASE_OFFSET'] [, 'PRINT_INTENSITY'] [, 'QUIET']
[, 'TABLE'] [, 'VERBOSE']);

In this call, *prescription_name* is a string specifying the name of the prescription, *result* is a variable in which the result of the propagation is returned, *wavelength* is the wavelength *in microns* at which to run the prescription, *gridsize* is the dimension of the wavefront grid array (*gridsize* by *gridsize*, where *gridsize* is a power of two), and *sampling* is a variable in which the sampling of the result in meters is returned (*sampling* is optional and will only contain a value if the prescription bothers to return one). Use of the optional *PASSVALUE* keyword was described in an earlier section. The other parameters are detailed in the **PROP_RUN** entry in the Routines Reference.

NOTICE: *The wavelength must be specified in microns when calling* ***PROP_RUN****.* ***PROP_RUN*** *will convert it to meters when calling the prescription. The prescription must assume that it is called with the wavelength specified in meters.*

To execute the example routine shown in the previous section, call **PROP_RUN** like this:

IDL:
```
prop_run, 'simple_prescription', psf, 0.5, 512, dx
```

Python:
```
(psf, dx) = proper.prop_run( 'simple_prescription', 0.5, 512 )
```

Matlab:
```
[psf, dx] = prop_run( 'simple_prescription', 0.5, 512 );
```

In this particular example, the wavelength is set to 0.5 μm, the wavefront grid size to 512 by 512 pixels, the wavefront intensity is returned in the variable *psf*, and the sampling in meters is returned in *dx*. The result of running this prescription is a 512 by 512 array containing the point spread function at the focus of a circular lens.

Some Things to Note in IDL

When it is necessary to be as accurate as possible, distances (propagation distances, focal lengths, wavelengths, etc.) should be specified as double precision values. In IDL, such values are denoted by using a "d" at the end of the number or instead of "e" for scientific notation exponents. Examples are:

```
1.5d
3.44d-2
4.2d10
```

IDL will get confused when "d" is used without proper spacing in equations (e.g. `3.1d-2-5`). In such cases, spaces are required to make the intent clear (e.g. `3.1d-2 - 5`).

It may be useful to force execution to stop at some point within a prescription so that variables, including the wavefront, can be examined interactively. For example:

```
…
stop
…

IDL> phase = prop_get_phase(wavefront)
IDL> plot, phase(*,256)
```

When Things Crash in IDL…

If a prescription crashes during execution, including within a PROPER library routine, by default IDL will halt at the code line where the error occurred. This allows the user to investigate why the routine crashed by interactively checking variable values. Note that many routines from other libraries, including those distributed with IDL, will instead return to the caller if an error occurs because they have set `on_error` or `on_ioerror`.

If the user chooses to rerun a prescription after it has crashed without first exiting IDL, be sure to issue the IDL command "`retall`", which will return the system all the way back to the main IDL program level. Be sure to check for any open files that need to be closed (with the command "`help, /files`"). If a save state (described in a following section) is active when a program crashes, the user should exit IDL, delete the temporary file(s) the save state system created, and restart.

The Wavefront Array Structure

The common parameter to all of the PROPER routines that propagate or modify the wavefront is the wavefront structure. This is a structure created by **PROP_BEGIN** that contains the double-precision, complex-valued, two-dimensional wavefront array (member name *wavefront*). The center of the wavefront is always at the lower-left corner of the array, consistent with the location of the zero-spatial-frequency element assumed by the Fourier transform routine. Modifications to the wavefront array using the PROPER library routines take this offset into account, shifting the center of any user-defined map (wavefront center assumed to be at the center of the map) to the corner before adding-to or multiplying-by the wavefront array. **PROP_END** shifts the wavefront origin to the center of the array.

The wavefront structure also has a number of members that contain information related to the wavefront array and the current propagation state, such as the current sampling, wavelength, reference surface characteristics, and so on. The values that may be of interest to the user can be obtained using the PROPER query library functions. It is important that the user avoid making any modifications or references to these member values directly. Altering these values can

lead to erroneous results, and there is no guarantee that the member names or types will remain constant in future versions of the PROPER library.

Sampling

Figuring out the wavefront array and initial beam dimensions necessary to obtain adequate sampling at a given surface (an intermediate one or the final image plane) is one of the more problematic issues when using a combination of near-field and far-field diffraction propagators, as is done in the PROPER routines. In the near-field (near the focus or, in a nearly-collimated beam near a pupil), the angular spectrum propagation method is used, and the sampling of the wavefront remains constant with propagation distance. In the far-field, away from focus, the Fresnel propagator is used, and the sampling changes proportionally with propagation distance to maintain a nearly-constant number of samples across the beam. Note that the wavelength affects where the switch is made between near and far-field propagators.

The wavefront is sampled using a finite square grid with dimensions specified by the user when **PROP_BEGIN** is called. The user must also specify the ratio of the initial beam diameter to the grid width (the default is 0.5). The beam should occupy only a portion of the wavefront array, with sufficient zero padding between its edges and the array's to reduce numerical artifacts introduced by the Fourier transforms. The combination of the grid and beam sizes, along with the wavelength and the location of the surface in the system, determines the sampling. Note that by changing the wavelength just a little, it is possible under some circumstances to drastically change the sampling, usually when propagating near the focus.

Setting the *TABLE* switch when calling **PROP_RUN** will cause it to print out a table listing the sampling at each surface.

In a well-corrected, conventional system (e.g. a telescope), the sampling, Δ, at the focus can be easily determined:

$$\Delta = \frac{F\lambda D}{N}$$

where D is the entrance pupil diameter and N is the grid width (both in the same units – meters, pixels, whatever), λ is the wavelength, and F is the focal ratio of the beam when approaching the focus. Increasing the beam diameter relative to the grid size (e.g. improving sampling of the beam) results in coarser sampling of the image at the focus (though the image extends to a greater angle), and vice-versa. Ideally, the image at the focus should be sampled at, or more finely than, the Nyquist criterion, which theoretically (in the absence of numerical artifacts and with a distribution of infinite extent), allows the value of a point at any location between the samples to be perfectly determined using sinc interpolation. The required sampling to meet this is

$$\Delta_{Nyquist} = \frac{F\lambda}{2}$$

For astronomers, the equivalent spacing in arcseconds is (for entrance pupil diameter D):

$$\Delta''_{Nyquist} = \frac{\lambda \cdot 360 \cdot 3600}{4\pi D}$$

This sampling is achieved when the entrance pupil diameter is half of the grid width. In reality, numerical artifacts and the limited number of samples can result in interpolation errors when the image is Nyquist sampled; interpolation using a slightly finer image sampling can reduce these errors, especially near the sharp core of a point spread function.

If an aperture contains thin obscurations (e.g. vanes that hold the secondary mirror in a reflecting telescope, or gaps between mirror segments), care must be taken to ensure that those structures are sufficiently sampled. This can be

done by specifying a combination of adequately large grid and initial beam sizes. It is advisable to have at least two samples across an obscuration or subaperture.

Polychromatic Imaging

A prescription routine computes a purely monochromatic result, which is fine if a laser or extremely narrow bandpass filter is used. To simulate a polychromatic result, say the image seen with a white-light source and broad bandpass filter, multiple monochromatic images must be generated using separate calls to **PROP_RUN** or use the **PROP_RUN_MULTI** routine to generate them in parallel. The images can then be added together with weights corresponding to the transmission of the system.

The issue of sampling is important when simulating a polychromatic image. Unless the user specifies a beam diameter/grid width ratio when calling **PROP_BEGIN**, the PROPER routines will compute a monochromatic image that is Nyquist-sampled for its particular wavelength. Adding a bunch of Nyquist-sampled, multi-wavelength images together will create an invalid result. The monochromatic images must, in the end, have the same sampling (in terms of meters per pixel). This can be achieved in two ways. The first is to specify, for each wavelength, a beam diameter/grid width ratio when calling **PROP_BEGIN** that provides the desired sampling. If this method is used, then the focal plane sampling should be chosen to be equal-to or finer-than the Nyquist sampling criterion at the shortest wavelength. The diameter of the entrance aperture pattern relative to the grid should not grow too large, which would result in wrap-around numerical artifacts. The alternative method is to compute a Nyquist-sampled (or finer) image at each wavelength and resample each one to a common grid. This can be done using **PROP_MAGNIFY** (with the *CONSERVE_FLUX* switch set) or **PROP_PIXELLATE**. This has some computational overhead due to the resampling, but the author considers this the safest solution.

PROPER Accuracy

As part of a NASA study of the modeling of coronagraphs, the accuracy of PROPER was determined relative to the results produced using a more rigorous (and substantially slower) algorithms. Please refer to the Milestone 1 results report of the NASA TDEM study “Assessing the performance limits of internal coronagraphs through end-to-end modeling” by Krist et al. (available at http://exep.jpl.nasa.gov/technology).

Running multiple instances of a prescription in parallel with PROP_RUN_MULTI

By default, PROPER runs a single instance of a prescription. If, for example, you want to propagate a wavefront at different wavelengths through a system, you would run a PROPER prescription separately using **PROP_RUN** for each wavelength in series. In another case, you may want to see how poking different deformable mirror actuators changes the field in the final plane, so again you would run the simulation multiple times, once per poke. With current computers having multiple CPUs and/or cores, this does not take full advantage of the parallel processing capabilities that can speed execution by multiple factors. While a number of IDL, Python, and Matlab routines (along with FFTW and the Intel Math Library FFT) take advantage of multithreading for parallel operations, there are usually some unused processing cycles left waiting to be exploited. Of course, one could always run multiple sessions at once, but starting the runs and then combining the results could be cumbersome. Also, on IDL and Matlab licenses with a limited number of sessions, this could eat up all the slots and prevent other users from running.

PROP_RUN_MULTI can be used in place of **PROP_RUN** to execute multiple instances of a prescription at once from within a single session. It can run a simulation simultaneously at different wavelengths or with different optional parameters, as specified by an array of *PASSVALUE* entries. In general, most prescriptions need little or no modification to make use of this capability.

As an example of the potential savings in run time, a prescription was run for 1, 5, and 9 wavelengths on a Linux workstation with dual Xeon CPUs (10 cores/20 threads per CPU), 96 GB of RAM, and using the Intel Math Library FFT. A single wavelength took 17.9 seconds. Running each sequentially would take 89.5 sec to do 5 wavelengths and 161.1 sec to do 9. Running 5 in parallel using **PROP_RUN_MULTI** took 33.5 sec (6.7 sec/wavelength) and 9 took 51.3 sec (5.7 sec/wavelength), speed improvements of 2.7x and 3.1x, respectively. Running even more wavelengths would eventually reduce the speedup factors.

PROP_RUN_MULTI makes use of the *IDL_IDLBRIDGE* object in IDL, the *multiprocessing* package in Python, and the *parpool* function in Matlab. Note that it requires the Parallel Processing Toolbox in Matlab, which is an extra-cost option.

Using PROP_RUN_MULTI

The calling sequence to **PROP_RUN_MULTI** is very similar to that of **PROP_RUN**:

IDL: **prop_run_multi**, *prescription*, *result, wavelength, gridsize [, sampling_m]*
[, /NO_SHARED_MEMORY] [, PASSVALUE=value] [, /PHASE_OFFSET] [, /QUIET]

Python: (*result, sampling_m*) = **proper.prop_run_multi**(*prescription*, *wavelength, gridsize*
[, NCPUS=value] [, PASSVALUE=value] [, PHASE_OFFSET=True/False]
[, QUIET=True/False])

Matlab: *result = - OR -*
[*result, sampling_m*] =
prop_run_multi(*prescription*, *wavelength, gridsize [, 'PASSVALUE', value]*
[, 'PHASE_OFFSET'] [, 'QUIET']);

Note that the *TABLE* and *VERBOSE* options that are available for **PROP_RUN** are not allowed here. The *wavelength* may be an array of wavelengths (in microns) and/or *optional_value* may be an array of optional parameters, including an array of structures. If both *wavelength* and *optional_values* are arrays, then both must have the same number of entries; in this case *optional_value[i]* is used for *wavelength[i]*. All entries will be run simultaneously in parallel. The *wavefront* variable will be a 3-D array with the 3rd dimension corresponding to the respective wavelength and/or parameter value entries. The *sampling* variable contains the returned final plane samplings as a vector (it is dependent on the user to return these values).

Avoid time-expensive process creation overheads in IDL & Matlab with the KEEP_THREADS option

By default, every time **PROP_RUN_MULTI** is called it will create parallel processes, run a copy of the prescription in each, gather the results, and then delete the processes. In IDL and Matlab, each process is a separate instantiation of IDL or Matlab, and creating the process incurs the overhead of starting and ending the program, including accessing the license server for each process. This can take a significant amount of time (a second or so per process) if the license server is on another machine (the server is accessed both when creating and freeing processes). This can add tens of seconds of just process creation overheads for each call to **PROP_RUN_MULTI**, and thus minutes if multiple **PROP_RUN_MULTI** calls are made from within a program.

When running **PROP_RUN_MULTI** multiple times from within a program, the repeated process creation/destruction overheads can be avoided using the KEEP_THREADS option. The first time **PROP_RUN_MULTI** is called with *KEEP_THREADS* set the required number of processes will be created, with the unavoidable overheads. However, when that instance of **PROP_RUN_MULTI** is finished, the processes will be kept alive for the next iteration, which will have no process creation overhead (as long as *KEEP_THREADS* is set when calling **PROP_RUN_MULTI**). As long as the number of required processes is equal to or less than the initial one, you can keep running **PROP_RUN_MULTI** without creating new ones. The processes will stay alive until either (1) they are freed using **PROP_FREE_THREADS**; (2) **PROP_RUN_MULTI** is called without *KEEP_THREADS* (the old processes will be deleted); or (3) you exit IDL or Matlab. Note that exiting the routine but not IDL or Matlab will not delete the processes. In Matlab the processes may also be terminated automatically if nothing happens for some amount of time.

As an example of how *KEEP_THREADS* can help, a PROPER prescription was run for 9 wavelengths sequentially using **PROP_RUN** and then in parallel using **PROP_RUN_MULTI**, without and with *KEEP_THREADS* set. As shown in the table below, there is a substantial advantage to using *KEEP_THREADS* for multiple calls (and yes, Matlab is faster running sequentially than once in parallel due to the overheads involved and Matlab's multithreaded instrinsic functions optimizations).

	Matlab	IDL	Python
Sequentially, no threads	24 s	84 s	99 s
Parallel, w/o KEEP_THREADS	36 s	39 s	20 s
1st run with KEEP_THREADS	25 s	29 s	
2nd run with KEEP_THREADS	13 s	18 s	
3rd run with KEEP_THREADS	13 s	19 s	

Note for Matlab users: Program behavior is likely to be unstable if you create your own processes in addition to those created in **PROP_RUN_MULTI.** Using *gcp* to control processes rather than **PROP_RUN_MULTI** will likely lead to a crash. If a parallel processing program crashes, it is best to restart Matlab than try to recover inside it, as PROPER's process allocation system will likely get confused.

Examples

The following contrived example prescription begins at a 48 mm diameter deformable mirror with an array of 48 × 48 actuators spaced by 1 mm and a circular aperture. A lens is located at the plane of the DM to focus the wavefront. The array of DM strokes is an element of the *optval* structure that is passed using the *PASSVALUE* keyword. This structure also includes *use_dm*, which specifies when to use the DM. This code and that which follows are available in the *examples* subdirectory of the PROPER directory.

IDL:

```
pro multi_example, wavefront, lambda_m, n, sampling, PASSVALUE=optval

diam = 0.048d
pupil_ratio = 0.25
fl_lens = 0.48d
n_actuators = 48     ;-- number of DM actuators in each dimension

prop_begin, wavefront, diam, lambda_m, n, pupil_ratio
   prop_circular_aperture, wavefront, diam/2
   prop_define_entrance, wavefront
   if ( optval.use_dm ) then begin
            dm_xc = n_actuators / 2.0
            dm_yc = n_actuators / 2.0
            dm_spacing = 1.0e-3
            prop_dm, wavefront, optval.dm, dm_xc, dm_yc, dm_spacing
   endif
   prop_lens, wavefront, fl_lens

prop_propagate, wavefront, fl_lens

prop_end, wavefront, sampling, /NOABS

return
end
```

Python:

```
import proper
import numpy as np

def multi_example(lambda_m, n, PASSVALUE = {'use_dm': False, 'dm': np.zeros([48,48],
       dtype = np.float64)}):

    diam = 0.048
    pupil_ratio = 0.25
    fl_lens = 0.48
    n_actuators = 48     # number of DM actuators in each dimension

    wfo = proper.prop_begin(diam, lambda_m, n, pupil_ratio)
    proper.prop_circular_aperture(wfo, diam/2)
    proper.prop_define_entrance(wfo)
    if PASSVALUE['use_dm']:
        dm_xc = n_actuators/2.
        dm_yc = n_actuators/2.
        dm_spacing = 1.e-3

        proper.prop_dm(wfo, PASSVALUE['dm'], dm_xc, dm_yc, dm_spacing)

    proper.prop_lens(wfo, fl_lens)

    (wfo, sampling) = proper.prop_end(wfo, NOABS = True)

    return (wfo, sampling)
```

Matlab:

```
function [wavefront, sampling] = multi_example(lambda_m, n, optval)
  diam = 0.048;
  fl_lens = 0.48;
  n_actuators = 48;
  pupil_ratio = 0.25;

  wavefront = prop_begin(diam, lambda_m, n, pupil_ratio);
  wavefront = prop_circular_aperture(wavefront, diam / 2.0);
  wavefront = prop_define_entrance(wavefront);
  if (nargin > 2) & (optval.use_dm == 1)
    dm_xc = fix(n_actuators / 2.0);
    dm_yc = fix(n_actuators / 2.0);
    dm_spacing =  1.0d-3;
    wavefront = prop_dm(wavefront, optval.dm, dm_xc, dm_yc, dm_spacing);
  end
  wavefront = prop_lens(wavefront, fl_lens);

  wavefront = prop_propagate(wavefront, fl_lens);

  [wavefront, sampling] = prop_end(wavefront, 'noabs');

end
```

Suppose we want to create a polychromatic point spread function over $\lambda = 0.5 – 0.7$ μm. We can do this by combining monochromatic PSFs generated in parallel at wavelengths spanning the passband. Because the final field is at focus, where the image scale is proportional to wavelength, we will need to resample each field to the same physical scale (here, 1.5 μm / pixel). We also poke a couple of actuators on the DM, just for fun.

IDL:

```
;-- program testmulti1.pro

lambda_min = 0.5
lambda_max = 0.7
nlambda = 9
gridsize = 1024
npsf = 256
final_sampling = 1.5e-6

;-- generate array of wavelengths

lambda_um = dindgen(nlambda) / (nlambda-1) * (lambda_max - lambda_min) + lambda_min

;-- create DM pattern (a couple of 0.1 micron pokes)

optval = {use_dm:1, dm:dblarr(48,48)}
optval.dm(20,20) = 0.2e-6
optval.dm(25,15) = 0.2e-6

;-- generate monochromatic fields in parallel

prop_run_multi, 'multi_example', fields, lambda_um, gridsize, sampling, PASSVALUE=optval

;-- resample fields to same scale, convert to PSFs

psfs = dblarr(npsf,npsf,nlambda)
for i = 0, nlambda-1 do begin
    mag = sampling(i) / final_sampling
    field = prop_magnify(fields(*,*,i), mag, npsf, /CONSERVE)
    psfs(0,0,i) = abs(field)^2
endfor

psf = total(psfs,3) / nlambda   ;-- add PSFs together

end
```

Python:

```
import proper
import numpy as np

def testmulti1():
    lambda_min = 0.5
    lambda_max = 0.7
    nlambda = 9
    gridsize = 256
    npsf = 256
    final_sampling = 1.5e-6

    # generate array of wavelengths
    wavelength = np.arange(nlambda) / (nlambda - 1.) * (lambda_max - lambda_min) + lambda_min

    # Create DM pattern (a couple of 0.1 micron pokes)
    optval = {'use_dm': True, 'dm': np.zeros([48,48], dtype = np.float64)}
    optval['dm'][20,20] = 0.2e-6
    optval['dm'][15,25] = 0.2e-6

    # generate monchromatic fields in parallel
    (fields, sampling) = proper.prop_run_multi('multi_example', wavelength, gridsize,
                              PASSVALUE = optval)

    # resample fields to same scale, convert to PSFs
    psfs = np.zeros([nlambda, npsf, npsf], dtype = np.float64)
    for i in range(nlambda):
        mag = sampling[i] / final_sampling
        field = proper.prop_magnify(fields[i,:,:], mag, npsf, CONSERVE = True)
        psfs[i,:,:] = np.abs(field)**2

    # add PSFs together
    psf = np.sum(psfs, axis = 0) / nlambda

    return

if __name__ == '__main__':
    testmulti1()
```

Matlab:

```
  final_sampling = 1.5d-6;
  gridsize = 1024;
  npsf = 256;
  nlambda = 9;
  lambda_min = 0.5;
  lambda_max = 0.7;

  % Generate array of wavelengths (um)

  lambda_um = lambda_min + (lambda_max-lambda_min) * [0:(nlambda-1)]/(nlambda-1);

  % Create Deformable Mirror pattern (a couple of 0.1 micron pokes)

  dm = zeros(48, 48);
  dm(21, 21) = 0.2d-6;
  dm(16, 26) = 0.2d-6;
  use_dm = 1;
  optval = struct('use_dm', use_dm, 'dm', dm);

  % Generate monochromatic fields in parallel

  [fields,sampling] = prop_run_multi('multi_example',lambda_um,gridsize,'passvalue',optval);

  % Resample fields to same scale, convert to Point Spread Functions

  psfs = zeros(npsf, npsf, nlambda);
  for i = 1 : nlambda
```

```
    mag = sampling(i) / final_sampling;
    field = prop_magnify(fields(:, :, i), mag, 'size_out', npsf, 'conserve');
    psfs(1:npsf, 1:npsf, i) = abs(field).^2;
  end

  psf = sum(psfs, 3) / nlambda; % Add PSFs together
```

Note that the complex-valued electric field at each wavelength is resampled onto a 256 × 256 grid using **PROP_MAGNIFY**. The interpolation tends to be more accurate when used on the complex field rather than the intensity of that field. The *CONSERVE* switch tells **PROP_MAGNIFY** to conserve energy (intensity). The monochromatic PSFs are averaged together at the end to create the broadband PSF.

We could have also specified a different DM pattern for each wavelength, or just have different DM patterns for one wavelength. In either case, we would not want to combine them into a single PSF. Here's a similar code, but this time with one wavelength and different DM patterns (cosine ripples with different periods). Note that *optval* is an array of structures, each containing the *use_dm* flag and the DM actuator stroke array *dm*.

IDL:

```
;-- program testmulti2.pro

lambda = 0.6
gridsize = 1024

;-- create different DM ripple patterns (50 nm amplitude)

npatterns = 3
optval = replicate({use_dm:1, dm:dblarr(48,48)}, npatterns)
x = dindgen(48) / 47 * (2*!pi) # replicate(1,48)

for i = 1, npatterns do optval(i-1).dm = 5.0e-8 * cos(4*x*i)

;-- generate monochromatic fields in parallel

prop_run_multi, 'multi_example', fields, lambda, gridsize, sampling, PASSVALUE=optval

end
```

Python:

```
import proper
import numpy as np

def testmulti2():
    wavelength = 0.6
    gridsize = 256

    # create different DM ripple patterns (50 nm amplitude)
    npatterns = 3
    optval = []
    for i in range(3):
        optval.append({'use_dm': True, 'dm': np.zeros([48,48], dtype=np.float64)})
    x = np.dot((np.arange(48.)/47 * (2*np.pi)).reshape(48,1), np.ones([1,48],
       dtype = np.float64))

    for i in range(npatterns):
        optval[i]['dm'] = 5.e-8 * np.cos(4*x*(i+1))

    # generate monochromatic field in parallel
    (fields, sampling) = proper.prop_run_multi('multi_example', wavelength, gridsize,
       PASSVALUE = optval)

    return
if __name__ == '__main__':
    testmulti2()
```

Matlab:

```
gridsize  = 1024;
npatterns = 3;
lambda = 0.6;

% Create different Deformable Mirror ripple patterns (50 nm amplitude)

dm = zeros(48, 48);
[x, y] = meshgrid(2.0d0 * pi * [0 : 47] / 47.0, ones(1, 48));
use_dm = 1;
optval = struct('use_dm', use_dm, 'dm', dm);

for i = 1 : npatterns
  optval(i).dm = 5.0d-8 * cos(4.0d0 * x * i);
  optval(i).use_dm = 1;
end

% Generate monochromatic fields in parallel
[fields,sampling] = prop_run_multi('multi_example',lambda,gridsize,'passvalue',optval);
```

Limitations

- Optional parameters can be passed to the prescription using the *PASSVALUE* keyword, but the prescription cannot return values to the caller via the same variables (**PROP_RUN** does allow this).
- Each instance of the prescription runs in a separate process, and memory, including common blocks, is not shared. Any changes to system variables apply only within each instance.
- The number of simultaneous processes should be less than (probably no more than half) the number of available processors. At some point the overheads involved defeat any advantages of parallelizing more instances. Error messages about unable to allocate resources means you are using too many instances.
- The runs may execute and finish in a different order than they were called. Any screen output produced by them may be randomly interleaved. The returned values are in the same order as the input values, however.
- The SAVESTATE features do not work properly when running in parallel.
- IDL: Unix (Linux, Mac OSx, etc.) versions of IDL prior to v8.3 require an active X Windows session to run the IDL_IDLBRIDGE objects used by PROP_RUN_MULTI. X windows must be running even if the machine has just a terminal interface. In order to run on such systems without an X display, or where X forwarding is not possible over the connection, it is necessary to create a virtual display. The procedure for doing this is described later. IDL versions later than v8.3 do not require X Windows for IDL_IDLBRIDGE to work.
- IDL: On Unix, parallel runs cannot be halted using Control-C in IDL. The only way to stop them is to do a Control-Z to stop IDL, and kill the IDL job, or wait until they finish. All prescriptions should be debugged first using **PROP_RUN**.

Running PROP_RUN_MULTI remotely (Unix IDL before v8.3)

As noted before, Unix versions of IDL before v8.3 require an active X Windows session to run the IDL_IDLBRIDGE objects that **PROP_RUN_MULTI** uses to implement parallel processing. This is no problem if you are logged into your own workstation, but if you log into another machine remotely (e.g., via *ssh*) you must also enable X forwarding (e.g., ssh –Y remote.machine.edu) even if you are not planning on producing any graphical output.

If you cannot do X forwarding (e.g., your firewall is not set up for it, perhaps), you will need to set up a virtual X display on that machine using *Xvfb*. This is not installed by default on many systems, so you must first get it. On Fedora, for instance, you can do this (requires root access or as sudo):

```
yum install Xvfb
```

You can then create a virtual X Windows frame buffer and execute IDL with one command:

```
xvfb-run idl
```

You can also use `xvfb-run` to execute jobs in the background or in batch.

Save States

NOTE: Save states must not be used when running in parallel with PROP_RUN_MULTI.

Suppose you are modeling a telescope with multiple mirrors, a deformable mirror (DM) for wavefront control, and a camera, and you are looking at how the image changes as you modify the DM actuator settings. The only optical surface that is varying in this case is the DM. If you are continually looping through the prescription (trying to iteratively correct a static wavefront error, for instance), it may be computationally wasteful, especially if a large grid size is used, to propagate the wavefront all the way through the entire system every time you change the DM settings if nothing else prior to the DM changes. A better solution would be to save the wavefront to a file just before encountering the DM when you first propagate through the system. During later iterations, rather than starting from the beginning, you can instead read in the wavefront, modify it with the DM, and continue propagating through the rest of the system.

PROPER has a simple system called *save states* that makes it relatively easy to do this, handling the bookkeeping that would otherwise make this a cumbersome process. The procedure outline of save state usage goes as follows:

1) In the top-level program containing the loop that repeatedly calls **PROP_RUN** to execute the prescription routine, the save state system is initialized by calling **PROP_INIT_SAVESTATE** before the loop.

2) At some point in the prescription routine, after **PROP_BEGIN** has been called and before the wavefront is first propagated, a call to **PROP_IS_STATESAVED** determines if a save state for the current wavelength has already been generated. If one has, then the user can bypass the initial bit of propagation and jump (via a goto or an if-then block) to the point where things begin to change.

3) In the prescription routine, the point where the wavefront should be saved, or read in if the state was previously saved, is identified by a call to **PROP_STATE**. After this, the propagation through the rest of the code occurs as usual.

4) After the looping **PROP_RUN** calls have finished in the top-level program, a call to **PROP_END_SAVESTATE** turns off the save state system and deletes the wavefront files that were written.

Here is a simple example of using save states. Suppose that we have a prescription routine that describes a system with two identical lenses positioned so that the first lens forms an image at an intermediate focus, where we might fiddle with the wavefront in some way. To avoid recomputing propagation through the first lens, we use save states (the user may be afraid of goto's, but the author is not…):

IDL:

```
pro example_system, wavefront, wavelength, gridsize, sampling, PASSVALUE=optval

diam = 1.0d
lens_fl = 20.0d
beam_ratio = 0.5

prop_begin, wavefront, diam, wavelength, gridsize, beam_ratio

if ( prop_is_statesaved(wavefront) ne 0 ) then goto, skip_first_lens

prop_circular_aperture, wavefront, diam/2
prop_define_entrance, wavefront
prop_lens, wavefront, lens_fl, '1st lens'
prop_propagate, wavefront, lens_fl, 'intermediate focus'
```

```
        skip_first_lens:
        prop_state, wavefront

        ;-- we are now at the intermediate focus, so pretend that
        ;-- we do something to the wavefront here and continue on

        prop_propagate, wavefront, lens_fl, 'second lens'
        prop_lens, wavefront, lens_fl, 'second lens'
        prop_end, wavefront

        return
        end
```

Python:

```
    import proper

    def example_system(wavelength, gridsize):

        diam = 1.
        lens_fl = 20.
        beam_ratio = 0.5

        # Define the wavefront
        wfo = proper.prop_begin(diam, wavelength, gridsize, beam_ratio)

        if proper.prop_is_statesaved(wfo) == False:
            proper.prop_circular_aperture(wfo, diam/2)
            proper.prop_define_entrance(wfo)
            proper.prop_lens(wfo, lens_fl, '1st lens')
            proper.prop_propagate(wfo, lens_fl, 'intermediate focus')

        proper.prop_state(wfo)

        # we are now at the intermediate focus, so pretend that
        # we do something to the wavefront here and continue on
        proper.prop_propagate(wfo, lens_fl, 'second lens')
        proper.prop_lens(wfo, lens_fl, 'second lens')
        (wfo, sampling) = proper.prop_end(wfo)

        return (wfo, sampling)
```

Matlab:

```
function [wavefront, sampling] = example_system(wavelength, gridsize)
  diam = 1.0;
  lens_fl = 20.0;
  beam_ratio = 0.5;

  wavefront = prop_begin(diam, wavelength, gridsize, beam_ratio);

  if (prop_is_statesaved(wavefront) == 0)
    wavefront = prop_circular_aperture(wavefront, diam / 2.0 );
    wavefront = prop_define_entrance(wavefront);
    wavefront = prop_lens(wavefront, lens_fl, '1st lens');
    wavefront = prop_propagate(wavefront, lens_fl, 'surface_name', 'intermediate focus');
  end

  wavefront   = prop_state(wavefront);

  % We are now at the intermediate focus, so pretend that
  % we do something to the wavefront here and continue on.

  wavefront = prop_propagate(wavefront, lens_fl, 'surface_name', 'second lens');
  wavefront = prop_lens(wavefront, lens_fl, 'second lens');

  [wavefront, sampling] = prop_end(wavefront);
end
```

We can now have a routine that repeatedly calls example_system:

IDL:

```
pro run_example, wavelength, gridsize

prop_init_savestate

for i = 0, 10 do begin
       prop_run, 'example_system', psf, wavelength, gridsize

       ;-- let us pretend that we now do something useful with
       ;-- this iteration's PSF and then compute another
endfor

prop_end_savestate

return
end
```

Python:

```
import proper

def run_example(wavelength, gridsize):

    proper.prop_init_savestate()

    for i in range(11):
        (psf, sampling) = proper.prop_run('example_system', wavelength, gridsize)

        #-- let us pretend that we now do something useful with
        #-- this iteration's PSF and then compute another

    proper.prop_end_savestate()

if __name__ == '__main__':
    run_example(0.5, 512)
```

Matlab:

```
function run_example( wavelength, gridsize )

prop_init_savestate;

for it = 0 : 10
    psf  = prop_run( 'example_system', wavelength, gridsize );

    % Let us pretend that we now do something useful with
    % this iteration's PSF and then compute another.
end

prop_end_savestate;

end
```

When **PROP_STATE** is first called, it writes the current wavefront structure and ancillary information to a file. A separate file is created for each propagation wavelength, and all of the files are deleted when **PROP_END_SAVESTATE** is called. The filenames begin with the wavelength, followed by random numbers, and ending with "_prop_savestate" (e.g. "550000000_4547_prop_savestate"). If the user exits before **PROP_END_SAVESTATE** is called, then these files will stay around until the user manually deletes them.

Apertures and Obscurations

Overview

The PROPER library contains procedures that create aperture and obscuration masks for a variety of shapes. Circles, rectangles, ellipses, and polygons can be drawn. Some routines multiply the wavefront by the apertures while others return an image of the aperture that can be further modified by the user and then multiplied by the wavefront using **PROP_MULTIPLY**. A series of calls to these aperture functions can create complex patterns.

The edges of the shapes are antialiased. The value of a pixel along the edge is set to fraction of the pixel area covered by the shape, computed by subdividing the pixel into 11 by 11 subpixels (this subsampling factor can be adjusted using **PROP_SET_ANTIALIASING**). This reduces high-frequency noise that would be created if these pixels were set to just one or zero.

For most PROPER shape-drawing routines, the size of the shape can be specified in meters or as a fraction of the size of the beam at that location if the */NORM* switch is set. Note that the beam size is determined from the Gaussian pilot beam, which provides a good diameter estimate for an unaberrated system but will be increasingly inaccurate as aberrations increase.

Note that in IDL and Python, which have (0,0) starting array indices, the center of the wavefront is at (x,y) = (fix(n)/2, fix(n)/2). In Matlab, which starts at (1,1), the center is at (x,y) = (fix(n)/2+1, fix(n)/2+1).

Examples

Multiply the current wavefront by a circular aperture with a diameter equal to the beam diameter at the current location:

```
IDL:      prop_circular_aperture, wavefront, prop_get_beamradius(wavefront)*2
Python:   proper.prop_circular_aperture( wavefront, prop_get_beamradius(wavefront)*2 )
Matlab:   wavefront = prop_circular_aperture( wavefront, prop_get_beamradius(wavefront)*2 );
```

The same thing as above can be done using the *NORM* switch:

```
IDL:      prop_circular_aperture, wavefront, 1.0, /NORM
Python:   proper.prop_circular_aperture( wavefront, 1.0, NORM=True )
Matlab:   wavefront = prop_circular_aperture( wavefront, 1.0, 'NORM' );
```

Doing it the hard way, a circle can be drawn in an image that then multiplies the wavefront:

```
IDL:      circle = prop_ellipse( wavefront, 1.0, 1.0, /NORM )
          prop_multiply, wavefront, circle
Python:   circle = proper.prop_ellipse( wavefront, 1.0, 1.0, NORM=True )
          proper.prop_multiply( wavefront, circle )
Matlab:   circle = prop_ellipse( wavefront, 1.0, 1.0, 'NORM' );
          wavefront = prop_multiply( wavefront, circle );
```

For another example, multiply the wavefront by a circular aperture (diam = 2.4 meters) that has a circular central obscuration (diam = 0.79 meters), such as the shadow of the secondary mirror in a Cassegrain reflector telescope like Hubble. The secondary mirror is held by four vanes, which are represented here by two overlapping rectangles arranged perpendicularly to each other. The lengths of the rectangles are greater than the aperture diameter to ensure that they extended completely across the opening.

```
IDL:      prop_circular_aperture, wavefront, 2.4/2
          prop_circular_obscuration, wavefront, 0.79/2
          prop_rectangular_obscuration, wavefront, 0.0264, 2.5
          prop_rectangular_obscuration, wavefront, 2.5, 0.0264
```

Now, let's add three small (r = 7.8 cm) circular obscurations (almost) equally spaced from each other near the edge of the aperture:

IDL:
```
prop_circular_obscuration, wavefront, 0.078, -0.9066, -0.5538
prop_circular_obscuration, wavefront, 0.078, 0.0, 1.0705
prop_circular_obscuration, wavefront, 0.078, 0.9127, -0.5477
```

The above statements creates the pattern shown in Figure 2. This happens to be the obscuration pattern of the Hubble Space Telescope. The three circular obscurations are pads that hold the primary mirror in place.

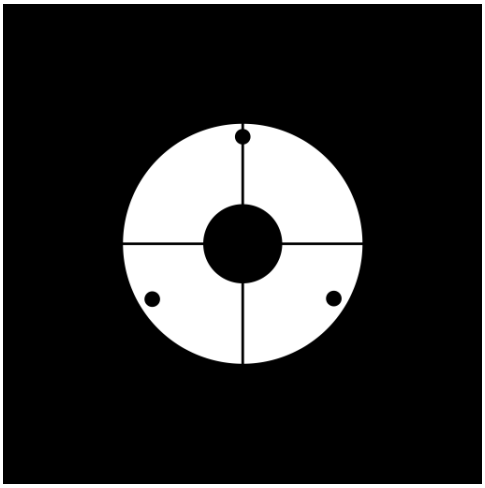


Figure 2. Obscuration pattern of the Hubble Space Telescope. The central obscuration is the shadow of the secondary mirror which is supported above the primary mirror by four vanes. The three small circles are pads that hold the primary mirror in place.

The drawing functions can be used to create complex shapes with varying transmissions:

IDL:
```
circle = prop_ellipse( wf, 1.0, 1.0, /NORM )
rectangle = prop_rectangle( wf, 0.5, 0.5, /NORM, /DARK ) * 0.5 + 0.5
aspect = 0.65
ring = prop_ellipse( wf, 0.85, 0.85*aspect, /NORM, /DARK )
ring = ring + prop_ellipse( wf, 0.70, 0.70*aspect, /NORM )
mask = circle * rectangle * ring
```

Python:
```
circle = proper.prop_ellipse( wf, 1.0, 1.0, NORM=True )
rectangle = proper.prop_rectangle( wf, 0.5, 0.5, NORM=True, DARK=True ) * 0.5 + 0.5
aspect = 0.65
ring = proper.prop_ellipse( wf, 0.85, 0.85*aspect, NORM=True, DARK=True )
ring = ring + prop_ellipse( wf, 0.70, 0.70*aspect, NORM=True )
mask = circle * rectangle * ring
```

Matlab:
```
circle = prop_ellipse( wf, 1.0, 1.0, 'NORM' );
rectangle = prop_rectangle( wf, 0.5, 0.5, 'NORM', 'DARK' ) * 0.5 + 0.5;
aspect = 0.65;
ring = prop_ellipse( wf, 0.85, 0.85*aspect, 'NORM', 'DARK' );
ring = ring + prop_ellipse( wf, 0.70, 0.70*aspect, 'NORM' );
mask = circle .* rectangle .* ring;
```

These statements create a clear circle with a square at the center that is ½ the amplitude of the circle (Figure 3). The square is surrounded by an elliptical ring with an aspect ratio of 0.65. When creating apertures with non-zero and non-unity values, remember that they represent *amplitude*, not *intensity* (square of amplitude), transmission.

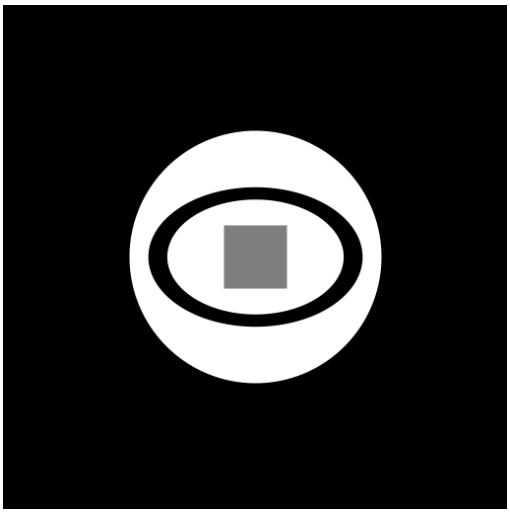


Figure 3. A more complex aperture function

Lenses and Mirrors

A lens or curved mirror alters the wavefront by introducing a radially-dependent phase change, causing the wavefront to converge, diverge, or become collimated. The routine **PROP_LENS** is used to reproduce the phase change created by a thin lens or curved mirror. A thin lens has a negligible thickness, so the difference in the indices of refraction between the lens material and the external medium are insignificant, there are no chromatic effects, and a ray enters and exits the lens at the same height. The lens assumed by **PROP_LENS** will take a collimated beam and create a spherically converging or diverging beam, or vice-versa. If the lens has a positive focal length, it is a convex lens (or the equivalent concave mirror) and will cause an incident collimated beam to converge. If the focal length is negative, it is a concave lens (convex mirror) that will cause a collimated beam to diverge as if it emanated from a point in front of the lens. In most cases, this default lens can be used when modeling well-corrected systems.

PROP_LENS makes no distinction between a lens and mirror. It simply creates a phase transformation that would be produced by an optical element with power. Positive focal lengths always indicate convex lenses and concave mirrors. Positive distances always indicate forward propagation through the system, from one surface (either real or virtual) to the next. These conventions are unlike those used in ray-tracing programs like Zemax or Code V where the sign signifying forward propagation changes after a reflection and the shape of a lens or mirror is defined relative to absolute spatial coordinates.

Aberrations

Zernike Polynomials

Classical low-order aberrations are often defined by Zernike polynomials. These form an orthogonal set of aberrations that include wavefront tilt, defocus, coma, astigmatism, spherical aberration, and others. The polynomials are functions of radius and azimuth angle within the aperture. It is common in optical modeling to use Zernike polynomials that have been normalized over the aperture (i.e. have a root-mean-square of 1.0 relative to a mean wavefront phase of 0.0). The coefficients to each polynomial specify the RMS amount of that aberration.

The PROPER routine **PROP_ZERNIKES** adds Zernike aberrations to the current wavefront, either as phase or amplitude errors. The user specifies which polynomials to add and the coefficient for each. The numbering scheme follows that established by Noll (J. Opt. Soc. Am., 66, 207 (1976)). Unobscured Zernikes up to an arbitrary number or the first 22 obscured Zernikes (those normalized over a centrally-obscured aperture) can be added. The routine **PROP_HEX_WAVEFRONT** can be used to add a hexagonal array of aberrated hexagonal segments.

The unobscured Zernike polynomial equations can be printed using **PROP_PRINT_ZERNIKES**. Zernike polynomials can be fit to a user-provided error map and aperture function using **PROP_FIT_ZERNIKES**.

User-Created or Measured Error Maps

The user may wish to apply an error map (either phase or amplitude) to the wavefront that was created on-the-fly (e.g. using an equation) or is an actual measurement of a surface (from an interferometer, for instance). **PROP_ADD_PHASE** takes a 2D array representing the wavefront or surface error in meters, converts it to phase error, and adds it to the phase term of the current wavefront. **PROP_MULTIPLY** and **PROP_DIVIDE** take an amplitude error map (with values ranging from 0.0 to 1.0) and multiply or divide the current wavefront by it.

These routines assume that the error map has the same size and sampling as the current wavefront array. The relevant values can be obtained using **PROP_GET_GRIDSIZE**, **PROP_GET_SAMPLING,** and **PROP_GET_BEAMRADIUS**. **PROP_ROTATE** can be used to rotate and/or shift a map and **PROP_MAGNIFY** to resample it. The optical axis is assumed to be at the center of the central pixel of the map.

PROPER also has routines that will read in an error map from a FITS file and resample it to match the sampling of the current wavefront array. **PROP_READMAP** will read in, shift (if necessary), and resample a map, but it will not apply it to the wavefront. **PROP_ERRORMAP** does all these as well as rotation and then applies the map to the wavefront (adding if wavefront or surface error, multiplying if amplitude error). **PROP_WRITEMAP** can be used to write an error map file containing header information used by **PROP_ERRORMAP**.

The Deformable Mirror

An optical system may use a deformable mirror (DM) to alter the phase of a wavefront, usually to correct for aberrations induced by the atmosphere, the optics themselves, or even the eye. Most major astronomical telescopes utilize DMs to correct for blurring caused by the atmosphere, sensing and correcting for low spatial frequency aberrations (currently <10 cycles/D, though progress is being made towards correcting higher frequency errors). In space, where atmospheric interference is not a problem, DMs will be used on telescopes designed for high-contrast imaging, such as the Terrestrial Planet Finder. To detect the light from a faint planet near a bright star, the wavefront must be as near-perfect as possible, so the errors in the optics due to manufacturing defects and thermally-induced variations need to be corrected. DMs can also be used to control amplitude errors by means of the Talbot effect.

The PROPER package includes a deformable mirror modeling routine, **PROP_DM**, which allows for realistic simulation of wavefront control. It assumes that the DM is a square array of regularly-spaced actuators that alter the height of a thin-facesheet mirror. DMs exist that have non-rectangular, irregularly-spaced actuators, and some have individual mirror segments for each actuator, such as micromirror arrays, but those are not modeled by **PROP_DM**.

The facesheet is a semi-rigid surface that exerts forces beyond the region controlled by a single actuator. It can pull up or down on neighboring actuators, resulting in surface heights that are not equal to the commanded displacements. Between actuators, facesheet tension can alter the surface in complicated ways. The *influence function* describes the deformation of the facesheet surface when a single actuator is pistoned above surrounding actuators. The function's shape depends on a number of factors, including the thickness and stiffness of the face sheet.

To reproduce the DM surface, a set of delta functions whose heights are equal to the values provided by the user in the DM z piston array are placed in a grid at the same sampling as the $1/10^{th}$ subsampled influence function. This array is then convolved with the influence function and then resampled via interpolation onto the wavefront array. This has been demonstrated to represent the behavior of a real DM well via testbed measurements.

IMPORTANT: For historical reasons, a negative piston provided to prop_dm results in a positive (away from the DM) displacement of the facesheet. The DM z piston values specify the DM facesheet deformation at the center of the actuator when a single actuator is pistoned, so a value of -1.0 would produce a maximum surface deformation of +1.0. However, due to the effects of the neighboring actuators and the rigidity of the facesheet, when multiple adjacent actuators are pistoned the net surface deformation will be greater or lower than the specified z piston value. Actuators will pull down or up on the others. Figure shows the DM surface when all but the edge actuators are pistoned by 1.0. The central flat region has a deflection of 1.43 because neighboring actuators are not pulling down those adjacent to them.

The pistons needed to achieve the desired surface can be derived by iteratively fitting the map of required surface heights with the influence-function-convolved actuator array. This can be done by providing the requested surface heights to **PROP_DM** and setting the *FIT* switch. Note that this is a crude fit, not a least-squares one.

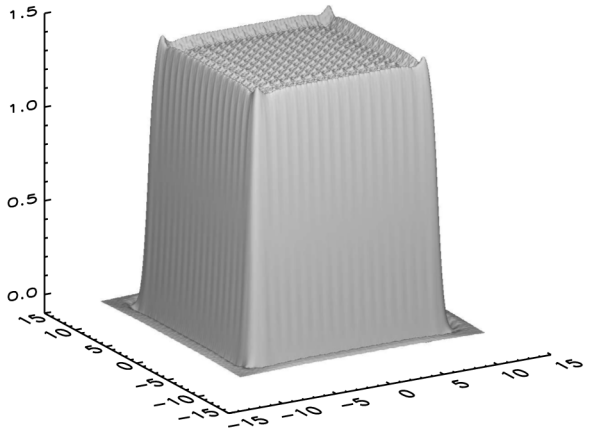


Figure 4. The simulated surface of a 20 x 20 actuator deformable mirror when the inner actuators are pistoned by 1.0 while the actuators along the outer edge are set to 0.0. The raised fence along the top edge and the depression along the bottom are caused by the rigidity of the facesheet and are what would be seen in a real DM. These effects are reproduced due to application of the influence function. Note that the mean deflection in the interior region is ~1.43, not 1.0.

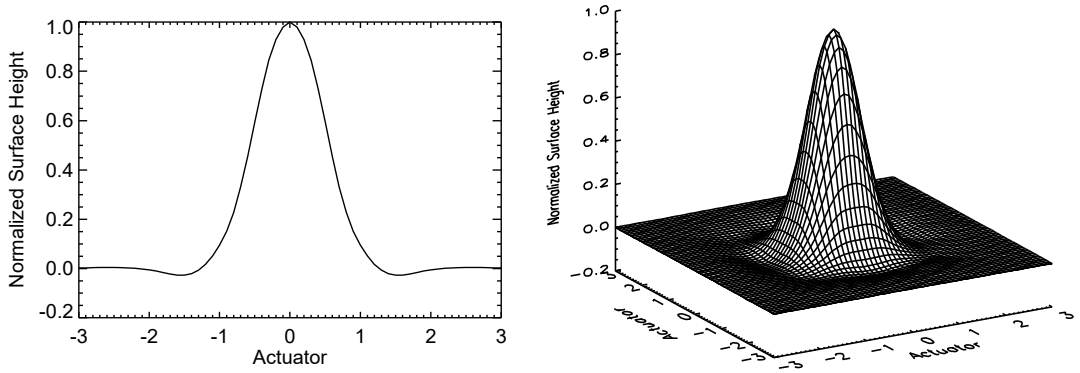


Figure 5. The measured average influence function for a Xinetics Photonex deformable mirror actuator. This represents the surface height when one actuator is pistoned above the local actuators

Defining the Influence Function

The default influence function (Figure 4) used in **PROP_DM** is for a Xinetics Photonex module actuator and was derived from measurements at the Jet Propulsion Laboratory and provided by John Trauger. It is sampled at 1/10th of the spacing between actuators. The user can override this and specify their own function using the *INFLUENCE_FUNCTION_FILE* optional parameter to **PROP_DM** that gives the name of a FITS file containing the 2D influence function map. There are two required header keywords: P2PD_M specifies the influence function sampling in meters, and C2CD_M specifies the actuator-to-actuator-centers spacing in meters (these values are assumed the same in X and Y). The oversampling factor, C2CD_M / P2PD_M, must be an integer value. The function must have the same dimensions in X and Y, and they must be an odd number of pixels, with the function centered in the middle of the array. The function should be normalized to have a peak value of 1.0. It should be smooth (the routine is not intended to reproduce actuators with significant surface errors, like those seen on Boston Micromachines MEMS DMs).

Inclination and Rotation of the Deformable Mirror

Deformable mirrors are typically not used at normal incidence but are usually inclined with respect to the incoming beam. If the beam is collimated and circular then it will have an elliptical footprint on the DM surface, with more

actuators across the beam along the major axis than in the perpendicular one. When projected onto the wavefront that is sampled equally in X and Y, the DM actuators will appear elongated. **PROP_DM** allows for the three-dimensional rotation of the DM surface about the wavefront central point and projects the surface orthographically (i.e., no perspective effect) onto the wavefront array. The optional *XTILT, YTILT*, and *ZTILT* keyword parameters allow the user to specify the rotations in degrees. The coordinate system is left handed with the unrotated DM surface in the X-Y plane with the first pixel in the lower left and the Z axis towards the observer. By default the rotations are done in X, Y, then Z; the *ZYX* switch can be used to reverse the order. Examples of the rotated surfaces are shown in Figure 4.

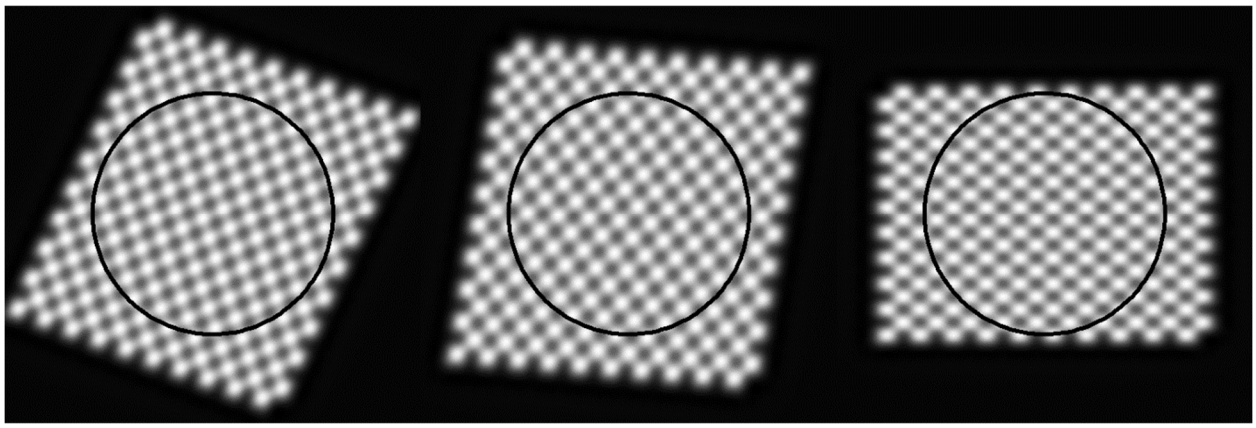


Figure 4. Surface maps (returned using the MAP keyword to PROP_DM) for different rotations of a 20 × 20 actuator DM with alternating actuator pistons. A circular beam is shown superposed for reference. Note that the central case was done using the reverse rotation order (using the ZYX switch).

Phase and Amplitude Error Maps Defined by Power Spectral Densities

Power Spectral Density

When modeling the propagation of light through an optical system, it is not enough to simply know that some given surface introduces an RMS wavefront error of *x*. If all of the errors are at low spatial frequencies, like spherical aberration or coma or astigmatism, the diffraction effects will be different than if they are all at higher ones (like surface roughness or pitting). Low spatial frequency errors (those of less than a few cycles across the surface, such as figuring errors) are often described by Zernike polynomials. Higher spatial frequency aberrations, which scatter light to larger angles, require many higher-order Zernike terms, often too many to be practically specified. If an aberration map is available for a given optic, say from an interferometer measurement, then it can be used to represent the errors up to the resolution limit of the map. In most instances, however, such maps are not available or they have insufficient resolution to represent the optic at higher spatial frequencies. A statistical description of the errors relative to the spatial scales of concern must instead be established. Using this, a two-dimensional map with similar statistics as the measured or expected surface can be generated. In PROPER, this can be done using the routine **PROP_PSD_ERRORMAP**.

A commonly used statistical descriptor, the power spectral density (PSD), specifies the power density (square of the amplitude per spatial frequency) of the error that exists at each spatial frequency. In essence, it says that a *y* amount of error with a spatial frequency of *x* cycles per diameter exists in the surface. If you take the Fourier transform of a one-dimensional profile measurement of a surface and then take the modulus square of the result, what you get is the one-dimensional PSD of that profile (ignoring things like windowing effects, etc.). Because the phase and the sign of the amplitude are lost when taking the modulus of the transform, a PSD can only represent the spatial frequency content of the errors and not their distribution across the surface.

One-dimensional PSDs represent the errors along a one-dimensional surface profile. However, most surfaces are considered two-dimensional structures, so there are two-dimensional PSDs as well. The 2D PSD can be obtained by taking the modulus square of the Fourier transform of a 2D surface map (again, excluding windowing effects). If the surface errors have spatially-random distributions (i.e. isotropic structure) or are circular and concentric, then the power distribution in the 2D PSD will be fully symmetric about the origin. In this case, a one-dimensional profile can be used to describe the mean radial profile of the 2D PSD (this is often the case). You can usually tell from the units whether a PSD profile is 1D or 2D. 1D PSDs are typically specified in units of height^2 × [distance/cycle] (e.g. nm^3) and 2D PSDs in height^2 × [distance/cycle]^2 (e.g. nm^4). From this point on, a "2D PSD" will refer to a one-dimensional profile representing the mean 2D PSD radial distribution. If a surface is isotropic such that a 1D surface height profile along any direction has essentially the same 1D PSD, then the mean 1D PSD can be converted to a 2D PSD (Church, Takacs, & Leonard in SPIE Proceedings, v. 1165, 136 (1989); Elson & Bennett in Applied Optics, v. 34, 201 (1995)). Real-world optics are often described by PSDs derived by combining interferometric, profile-stylus, and scatterometer measurements, spanning a wide range of spatial scales.

The majority of the errors in most optics are in the low spatial frequency range due to limitations of the figuring process. Beyond a certain spatial frequency (often related to the size of the figuring & polishing tools), the surface errors rapidly decrease in power. As an example, let us look at the surface errors in a real mirror (Figure 5). In this case, 1D PSDs for many 1D radial surface profiles were computed to create the mean 1D PSD profile shown (this surface is, for the most part, radially isotropic but not generally isotropic, so this is not strictly accurate to some degree, but is easier to understand). The profile shows that the largest errors are at low spatial frequencies, and the error level decreases toward higher ones. At some frequency, the aberration level begins to drop more rapidly as figuring errors give way to polishing errors.

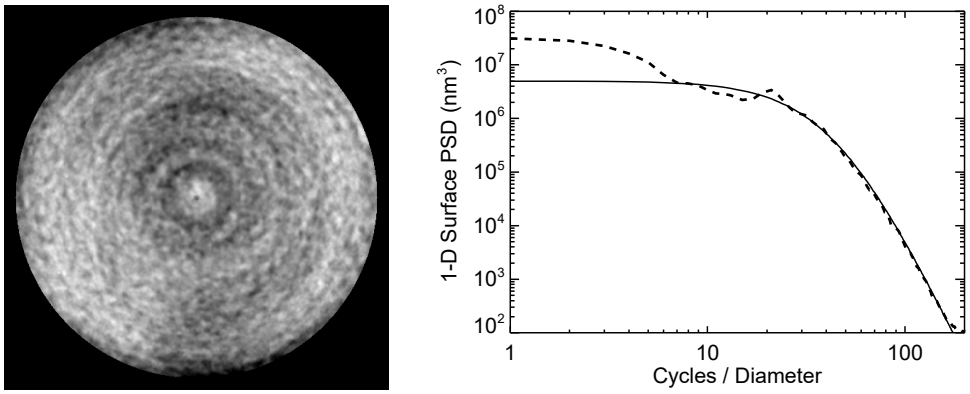


Figure 5. (Left) The surface error map for a 0.2 m, ultra-high-precision mirror; (Right) The mean 1-D PSD profile derived from line profiles across the mirror, shown as a dashed line. A fitted PSD curve is overplotted as a solid line. The fitted PSD matches well at high spatial frequencies, but it under-represents the amount of low-order aberrations.

A PSD profile is often characterized by fractal-like spectrum parameters, which is how PSDs are specified in **PROP_PSD_ERRORMAP**. By default, the parameterized PSD is assumed to have the K-correlation form described by Church, Takacs, & Leonard in SPIE Proceedings, v. 1165, 136 (1989):

$$PSD_{2D}(k) = \frac{a}{\left[1 + \left(k/b\right)^2\right]^{\frac{c+1}{2}}}$$

where k is the spatial frequency (cycles/meter) and a is the power at low spatial frequencies (m^4). The value b is the correlation length; $b/(2\pi)$ is where the lines describing the flat spectrum at the low spatial frequencies and the power law falloff at higher ones intersect (in cycles/meter) – it essentially indicates where the PSD curve bends. The power law falloff is defined by c. An alternative form used by the Terrestrial Planet Finder project is assumed when the */TPF* switch is set in **PROP_PSD_ERRORMAP**:

$$PSD_{2D}(k) = \frac{a}{\left[1 + \left(k/b\right)^c\right]}$$

Note that for the TPF form, the bend in the PSD occurs at b rather than $b/(2\pi)$. As shown in the example in Figure 5, the parameterized PSD profile matches this particular measured PSD at higher spatial frequencies but not at lower ones. This is a constraint of the flat low-frequency distribution assumed by the parameterized PSD. If we create an error map using these parameters, its low-order aberration content will be below that of the actual surface.

Let us now try to force the real surface error map to have a PSD more like those assumed by the above equations. This is done only to demonstrate certain aspects and limitations of the PSD specification that the user needs to take into account when modeling a system (and it demonstrates another PROPER routine along the way). Because our map is a bit heavy in low-order errors, we can try to get rid of some them by fitting and subtracting Zernike polynomial aberrations. **PROP_FIT_ZERNIKE**S can be used to fit Zernike polynomials to an error map, and it returns a map containing just those polynomials, as shown in Figure 6. Subtracting the Zernike map from the surface clearly gets rid of a large fraction of the low-order errors, though it leaves behind those that do not match the Zernike polynomial patterns.

Figure 6. (Left) The measured surface error map; (Middle) The sum of the Zernike polynomials fit to the original map using PROP_FIT_ZERNIKES; (Right) The original map after subtraction of a limited set of Zernike aberration polynomials.

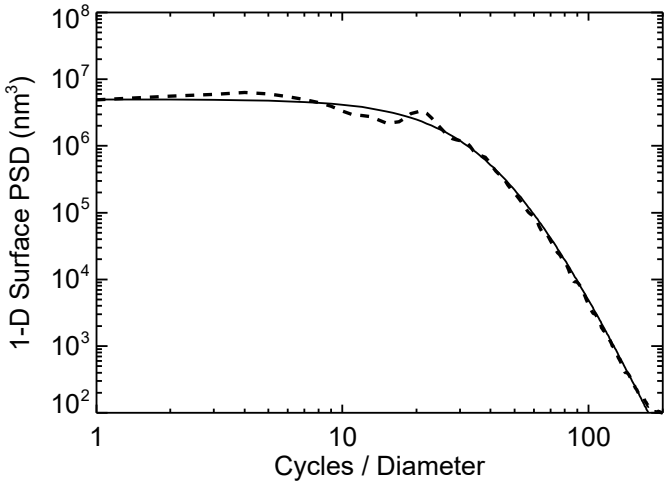


Figure 7. PSD profile derived from the surface map after subtraction of 22 Zernike polynomials.

The PSD of this new, Zernike-subtracted map is shown in Figure 7. This more closely matches the profile shape assumed by the parameterized PSD. We can now take the fitted PSD parameters and plug them into **PROP_PSD_ERRORMAP** to create a realization of the PSD (a map having the same spatial frequency error content as the Zernike-subtracted real map). By running models using the Zernike-subtracted and PSD-realization maps, we can compare how well the two maps agree in terms of diffraction effects.

An Example Using PSD-Defined Error Maps

A good system to use for this example is a stellar coronagraph, which is discussed in a later section. A coronagraph suppresses the light from a point source (a star) that is diffracted by edges in a system (entrance aperture, support vanes, etc.), but it does not reduce the light scattered by mid-and-high spatial frequency surface errors like those that

are characterized by a PSD. Because the bulk of the diffraction structures (Airy rings) are removed, it is easier to see the effects of these surface errors in the simulated image. We use here the simple Lyot coronagraph detailed in the Examples section, applying either the Zernike-subtracted measured map or the PSD-realized map to the telescope lens (note that errormap.fits is not distributed with PROPER).

IDL:

```
pro psdtest, wavefront, wavelength, gridsize, sampling, PASSVALUE=usepsdmap

lens_diam = 0.212d              ;-- 0.212 meter lens diameter
lens_fl = 24.0 * lens_diam      ;-- focal length (f/24 focal ratio)
beam_width_ratio = 0.5

prop_begin, wavefront, lens_diam, wavelength, gridsize, beam_width_ratio

;-- create circular entrance aperture

prop_circular_aperture, wavefront, lens_diam/2
prop_define_entrance, wavefront

;-- If the variable usepsdmap is not defined (via optional passed value),
;-- read in and use the map which represents the wavefront error (in this case,
;-- it's in nanometers, hence we need to multiply it by 1e-9 to convert it to meters)
;-- and has a sampling of 0.4 mm/pixel.  If usepsdmap is defined, then generate
;-- and use a PSD-defined map.  The maps have an RMS of about 1.0 nm.

if ( n_elements(usepsdmap) eq 0 ) then begin
  prop_errormap, wavefront, 'errormap.fits', SAMPLING=0.0004, MULTIPLY=1e-9, /WAVEFRONT
endif else begin
  a = 3.29d-23   ;-- low-freq power in m^4
  b = 212.26     ;-- correlation length (cycles/m)
  c = 7.8        ;-- high-frequency falloff (r^-c)
  prop_psd_errormap, wavefront, a, b, c
endelse

prop_lens, wavefront, lens_fl, 'telescope lens'
prop_propagate, wavefront, lens_fl, 'intermediate focus'

;-- multiply field by occulting mask with 4*lam/D HWHM transmission

prop_8th_order_mask, wavefront, 4, /CIRCULAR

prop_propagate, wavefront, lens_fl, 'pupil imaging lens'
  prop_lens, wavefront, lens_fl, 'pupil imaging lens'
prop_propagate, wavefront, lens_fl, 'lyot stop'
prop_circular_aperture, wavefront, 0.53, /NORM     ;-- Lyot stop

prop_propagate, wavefront, lens_fl, 'reimaging lens'
  prop_lens, wavefront, lens_fl, 'reimaging lens'
prop_propagate, wavefront, prop_get_distancetofocus(wavefront), 'final focus'

prop_end, wavefront, sampling

return
end
```

Python:

```
import proper

def psdtest(wavelength, gridsize, PASSVALUE = {'usepsdmap': False}):

    lens_diam = 0.212            # 0.212 meter lean diameter
    lens_fl = 24. * lens_diam    # focal length (f/24 focal ratio)
    beam_width_ratio = 0.5

    wfo = proper.prop_begin(lens_diam, wavelength, gridsize, beam_width_ratio)

    # Create circular entrance aperture
    proper.prop_circular_aperture(wfo, lens_diam/2)
    proper.prop_define_entrance(wfo)

    #-- If the variable usepsdmap is not defined (via optional passed value),
    #-- read in and use the map which represents the wavefront error (in this case,
    #-- it's in nanometers, hence we need to multiply it by 1e9 to convert it to meters)
    #-- and has a sampling of 0.4 mm/pixel.  If usepsdmap is defined, then generate
    #-- and use a PSD-defined map.  The maps have an RMS of about 1.0 nm.

    if PASSVALUE['usepsdmap']:
        a = 3.29e-23     # low-freq power in m^4
        b = 212.26       # correlation length (cycles/m)
        c = 7.8          # high-freq falloff (r^-c)

        proper.prop_psd_errormap(wfo, a, b, c)
    else:
        proper.prop_errormap(wfo, 'errormap.fits', SAMPLING = 0.0004,
                MULTIPLY = 1e-9, WAVEFRONT = True)

    proper.prop_lens(wfo, lens_fl, 'telescope lens')
    proper.prop_propagate(wfo, proper.prop_get_distancetofocus(wfo), 'intermediate focus')

    # multiply field by occulting mask with 4*lam/D HWHM transmission
    mask = proper.prop_8th_order_mask(wfo, 4, CIRCULAR = True, MASK = True)

    proper.prop_propagate(wfo, lens_fl, 'pupil imaging lens')
    proper.prop_lens(wfo, lens_fl, 'pupil imaging lens')
    proper.prop_propagate(wfo, lens_fl, 'lyot stop')
    proper.prop_circular_aperture(wfo, 0.53, NORM = True)  # Lyot stop

    proper.prop_propagate(wfo, lens_fl, 'reimaging lens')
    proper.prop_lens(wfo, lens_fl, 'reimaging lens')
    proper.prop_propagate(wfo, proper.prop_get_distancetofocus(wfo), 'final focus')

    (wfo, sampling) = proper.prop_end(wfo)

    return (wfo, sampling)
```

Matlab:

```
function [wavefront, sampling] = psdtest(wavelength, gridsize, usepsdmap)

  if nargin < 3
    usepsdmap = 0;
  end

  lens_diam = 0.212;                % telescope diameter (m)
  lens_fl =   24.0 * lens_diam;     % focal length (m) (f/24 focal ratio)
  beam_width_ratio = 0.5;           % beam diameter fraction

  wavefront = prop_begin(lens_diam, wavelength, gridsize, beam_width_ratio);
```

```
 % Create circular entrance aperture

  wavefront = prop_circular_aperture(wavefront, lens_diam / 2.0);
  wavefront = prop_define_entrance(wavefront);

  % If the variable usepsmap is not set, read in and use the map which
  % represents the wavefront error (in this case, it's in nanometers,
  % hence we need to multiply it by 1e-9 to convert it to meters) and has
  % a sampling of 0.4 mm/pixel.  If usepsdmap is set, then generate and
  % use a PSD-defined map.  The maps have an RMS of about 1.0 nm.

  if usepsdmap == 0
    wavefront = prop_errormap(wavefront, 'errormap.fits', 'wavefront', ...
      'sampling', 0.0004, 'multiply', 1.0d-9)
  else
    a = 3.290d-23;     % low spatial frequency power (m^4)
    b = 212.26;        % correlation length parameter (cycles / meter)
    c = 7.8;           % high frequency falloff power law exponent
    wavefront = prop_psd_errormap(wavefront, a, b, c);
  end

  wavefront = prop_lens(wavefront, lens_fl, 'telescope lens');
  dif = prop_get_distancetofocus(wavefront);    % distance to focus (m)
  wavefront = prop_propagate(wavefront, dif, 'snm', 'intermediate focus');

  % Multiply field by occulting mask with 4*wavelength/lens_diam HWHM transmission

  [wavefront, mask] = prop_8th_order_mask(wavefront, 4.0, 'circular');

  wavefront = prop_propagate(wavefront, lens_fl, 'snm', 'pupil imaging lens');
  wavefront = prop_lens(wavefront, lens_fl, 'pupil imaging lens');
  wavefront = prop_propagate(wavefront, lens_fl, 'snm', 'lyot stop');
  wavefront = prop_circular_aperture(wavefront, 0.53d0, 'norm');  % Lyot stop

  wavefront = prop_propagate(wavefront, lens_fl, 'snm', 'reimaging lens');
  wavefront = prop_lens(wavefront, lens_fl, 'reimaging lens');
  dff = prop_get_distancetofocus(wavefront);    % distance to focus (m)
  wavefront = prop_propagate(wavefront, dff, 'snm', 'final focus');

  [wavefront, sampling] = prop_end(wavefront);
end
```

Figure 8 shows the "real" map and the simulated one generated by **PROP_PSD_ERRORMAP** using the PSD of the "real" one. The overall scales of the structures in simulated map matches those in the real map fairly well at moderate spatial frequencies. However, the small ripples on the real surface are arranged in concentric patterns about the center, a result of the figuring and polishing processes, while they are randomly distributed on the simulated surface. This is a result of the lack of phase information being carried over into the PSD. There are also some lower-order, annular zones in the real map that are not reproduced in the simulation.

Figure 9 shows the results of using each map. The speckles are created by the surface errors (the image would be almost completely dark without these aberrations). Their sizes and intensities relative to their radius from the center are similar in both, indicating that the simulated map is reproducing the spatial frequency content well. This is also evident in the radial profiles shown in Figure 10. However, near the center, where lower-spatial-frequency aberrations are dominant, the circular rings appear in the "real" image that are not apparent in the "fake" one. This is due to the low-to-mid-spatial-frequency errors that were not subtracted out. This mismatch at lower frequencies is also seen in the radial intensity profiles.

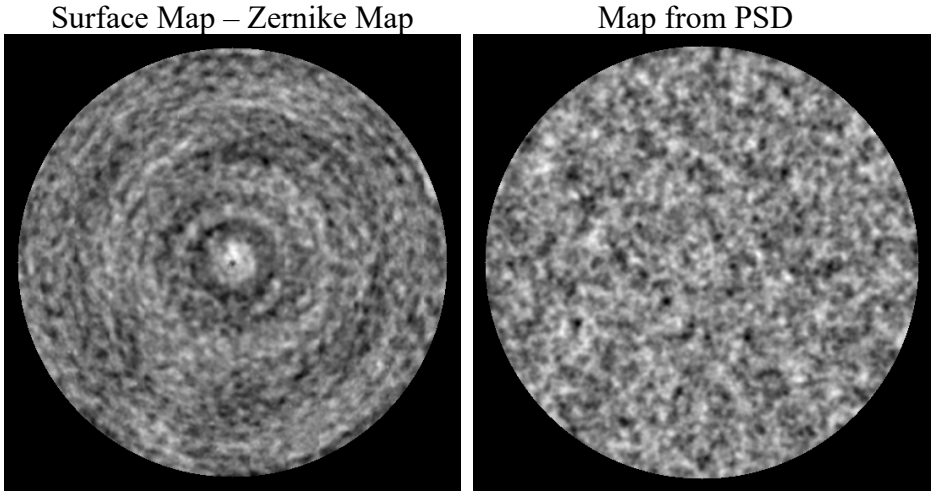


Figure 8. (Left) Surface error map after fit and subtraction of Zernike polynomials; (Right) Simulated surface error map (multiplied by the aperture shape) generated using the PSD of the map on the left.

Defining Amplitude Errors Using PSDs

At some level an optic will have non-uniform transmittance or reflectance over its area due to unavoidable manufacturing errors, such as irregular coating deposition. These alter the wavefront's amplitude spatial distribution and can create effects similar to phase errors, such as speckles (though amplitude-induced speckles have different wavelength dependencies than phase-induced speckles). In the large majority of optical systems such amplitude errors are not important, as phase errors are usually orders of magnitude larger in effect. However, for some systems in which the phase errors are corrected to a high degree with deformable mirrors, such as space-based planet-finding stellar coronagraphs, amplitude errors may be the primary residual wavefront aberration.

The spatial distributions and ranges of coating amplitude errors do not appear to be well known, especially at sub-percent levels. Presumably, amplitude errors can be described using PSDs like phase errors (though it is not known to the author whether this is used by industry or not). **PROP_PSD_ERRORMAP** can generate an amplitude error map when the *AMPLITUDE* keyword is defined using the same PSD profile specifications used for phase errors. The value of the *AMPLITUDE* keyword sets the maximum amplitude level. Because intensity is the square of amplitude, *AMPLITUDE=0.9* would result in a maximum transmission of 0.81. The *amp* parameter in **PROP_PSD_ERRORMAP** specifies the RMS amplitude variation of the entire map about the mean amplitude (thus the *RMS* switch is ignored when *AMPLITUDE* is specified). The parameters *b* and *c* define the PSD profile shape in the same way as they do for phase errors. The wavefront is multiplied by the amplitude map.

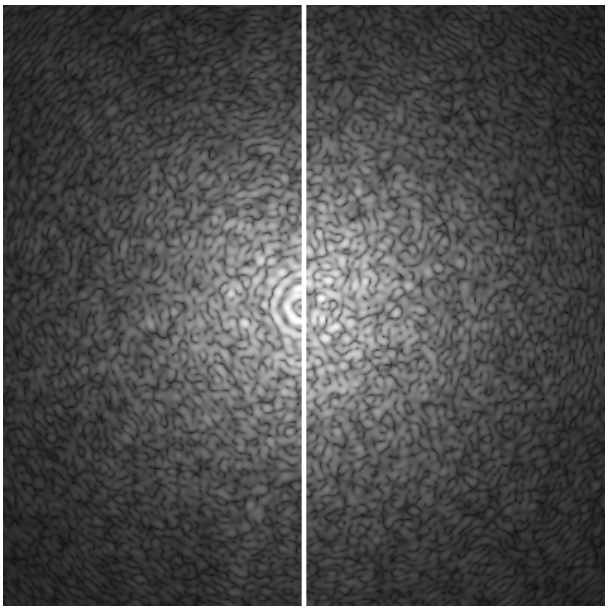


Figure 9. The simulated image (monochromatic light) of a point source observed using a coronagraph with a circular, 8th-order occulting mask with a 4λ/D half-width-half-max transmission profile. The left side shows the image computed using the measured, Zernike-subtracted surface map to represent the errors in the telescope lens. The right shows the result of the same propagation sequence but using the map generated from the PSD of the "real" map. Both are displayed with identical, nearly-logarithmic intensity scaling. The image is 200 *λ/D* on each side. An aberration with a spatial scale of one cycle over the pupil diameter (*D*) would create a speckle at *λ/D*.

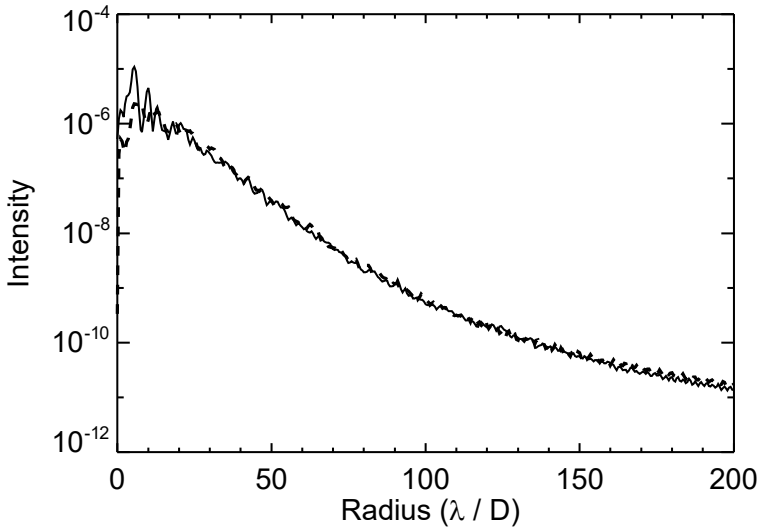


Figure 10. Mean radial intensity profiles of the coronagraphic images shown in Figure 9. The solid line is from the image based on the measured map and the dashed is from that based the PSD-derived map.

PSD-Defined Maps for Inclined Surfaces

When modeling a surface that has errors defined by a PSD, the inclination of the surface relative to the direction of the incoming beam must be taken into account. The projection of the beam onto the surface results in unequal beam diameters along the orthogonal reference axes. For example, consider the elliptical mirror shown in the left side of Figure 11 that has circular polishing errors. This mirror is then inclined 45º relative to an incident, circular, collimated beam, bending the beam 90º. The projection of the circular beam onto the mirror leads to a difference in the spatial frequency distribution of the errors in terms of cycles/diameter for each axis, as shown in the right side of the figure. As a result, the diffraction pattern at the image plane will have light scattered to greater angles along the direction corresponding to the major axis of the elliptical mirror than along the other.

The *INCLINATION* keyword in **PROP_PSD_ERRORMAP** can be set to the inclination in degrees of the surface relative to the direction of the beam (e.g. at *INCLINATION=0*, the beam direction is perpendicular to the surface). The inclination angle is measured in the Y-Z plane relative to the Y axis. The sign of the inclination is not significant. The *ROTATION* keyword can be set to an angle to rotate the map after projection onto the inclined surface.

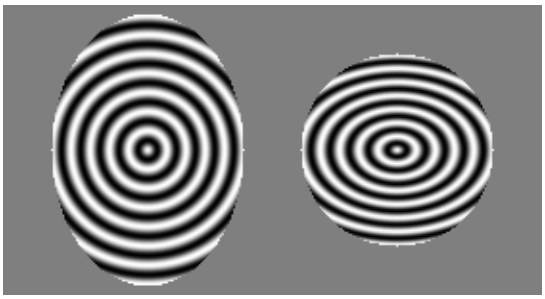


Figure 11. (Left) Looking perpendicularly to an elliptical surface containing errors of a certain spatial frequency; (Right) The projection of the surface errors in a collimated beam with a 45º incidence angle to the elliptical mirror. In the inclined case, the spatial frequency of the errors is higher relative to the beam diameter in the Y direction.

Limitations of PSD-Defined Error Maps

An error map generated by **PROP_PSD_ERRORMAP** provides a useful method for simulating the aberrations in an optical component over a broad range of spatial scales. However, it cannot fully encompass the entire range of errors that may be present. As previously shown, a PSD cannot specify the spatial distribution of errors (e.g. whether they occur in rings), which can lead to a lack of spatially-correlated structure in the diffraction pattern.

The error map is generated by taking the Fourier transform of the 2D PSD (mapped onto a 2D array with the same dimensions as the wavefront grid) with random phases. Because the PSD array has discrete sampling, not all spatial frequencies can be represented. The lowest spatial frequency is approximately ½ cycle over the beam diameter (piston is not included by **PROP_PSD_ERRORMAP**). This means that wavefront tilt, for instance, would not appear in these maps. More advanced methods of error map generation may be implemented in the future that would include lower-frequency terms. Lower-order terms can be added using **PROP_ZERNIKES**.

The aberration map is not guaranteed to have the expected RMS error over the illuminated portion of the surface due to the method used to generate it. If the PSD is grossly dominated by low-spatial-frequency errors, then there can be localized regions in the error map with significantly different mean deviations, and so the RMS error within the illuminated portion can depend strongly on the location of the beam with respect to these regions. The PSD is best used to define aberrations of a few cycles or higher.

Notes Regarding PROP_PSD_ERRORMAP

To avoid recomputing a PSD-defined error map every time the same prescription is run, the generated map can be written to a file by specifying a filename with the *FILE* keyword. The next time the prescription is run, the routine will look first to see if the map file exists, in which case it will read it in rather than generate a new map.

If a PSD-defined map file already exists for one grid size and a propagation is performed at a larger size, an error message indicating this will be displayed when **PROP_PSD_ERRORMAP** tries to read in the file, and execution will stop. If you wish to use the same map regardless of the grid dimensions, do an initial propagation using the largest grid size that might be used in the future. You can then perform the propagation again with a smaller size and **PROP_PSD_ERRORMAP** will resample the larger map to match the smaller grid. NOTE: This method is definitely not optimal, and it is strongly suggested that a map is used only for the grid size at which it was generated.

The map generated by **PROP_PSD_ERRORMAP** can be returned using the *MAP* keyword.

If want to generate an error map but do not want to apply it to the wavefront, then use the *NO_APPLY* switch.

Multiple PSD-defined error maps can be applied to the same surface by successively calling the routine.

Because of the way the maps are generated, aberrations with spatial frequencies of less than ~½ cycle/D will not be included. Such aberrations may be included by adding in Zernike polynomials.

When generating an amplitude error map, the map is divided by the maximum value in the entire array (not just the illuminated portion) and then multiplied by the value specified by the *AMPLITUDE* keyword.

Examples

A Simple Telescope

A very simple telescope consists of an objective lens and an eyepiece. The objective can be a convex lens (for a refractive telescope) or a concave mirror (a reflecting telescope). The eyepiece is a simple lens that, in combination with the eye, magnifies the image produced at the focus of the objective. The telescope here has a 60 mm diameter objective with a focal ratio of *f*/15, and the eyepiece has a 21 mm focal length. The system is focused at infinity for astronomical observations. The eye is represented by a lens with a 22 mm focal length that focuses onto the retina (the final focus of the system). The lens of the eye is located at the exit pupil of the system. The exit pupil is where the eyepiece forms an image of the objective, and its location is easily derived in this case using the lens law. This prescription will return the point spread function of the telescope (i.e. the image of a star) as seen by the eye. Because the beam starts off with half of the width of the computational grid (`beam_ratio=0.5`), the final PSF is Nyquist-sampled. Note that the optional passed value is not used here, but it must still be included in the procedure's parameter declarations.

This PSF is shown in the left image of Figure 12. The expected Airy pattern should be perfectly circularly symmetrical, but the simulated PSF shows obvious computational artifacts. The cross-shaped pattern and streaks along the diagonals and the image axes are caused by wrap-around aliasing errors that are a consequence of using Fourier methods with finite grids. Increasing the grid size reduces many of these artifacts (right image in Figure 12), at the expense of increased computation. The level of these errors is very small (they are enhanced in the images by using a logarithmic intensity stretch), and are probably not significant when modeling most systems, especially if surface errors are included that scatter light into the wings.

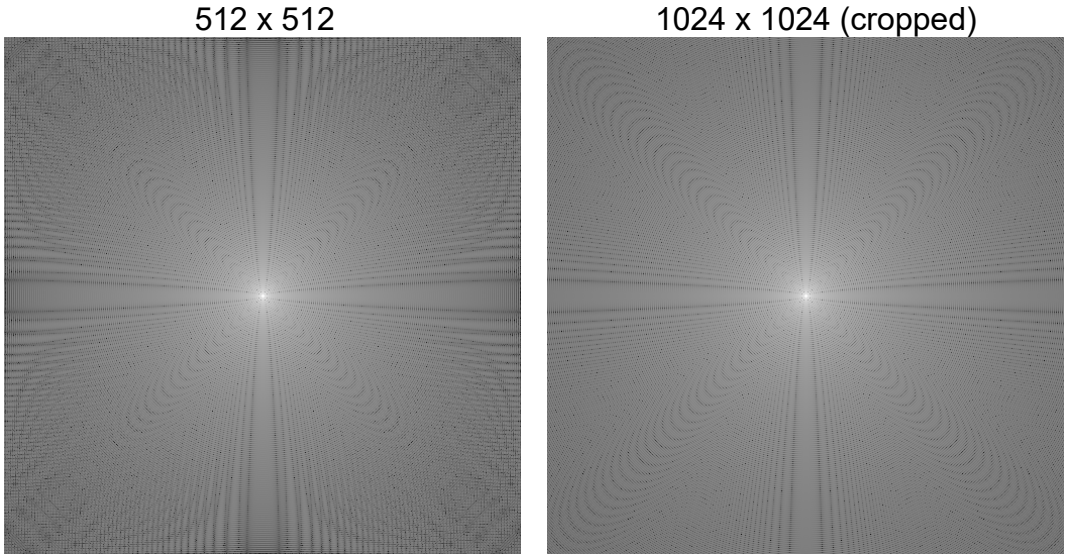


Figure 12. Nyquist-sampled point spread functions for a circular aperture computed using 512^2 and 1024^2 grids (the PSF from the larger grid has been cropped). The images are shown with logarithmic intensity stretches to emphasize the low-level features.

The example above produced a Nyquist-sampled PSF. Theoretically, the PSF at any finer resolution can be derived from this PSF using sinc interpolation (using **PROP_MAGNIFY**, for instance). In reality, computational artifacts introduce noise that create interpolation errors. It is sometimes better to create an image sampled slightly finer than Nyquist. To create a PSF that is sampled 1.2 times finer than Nyquist in the example, *beam_ratio* can be decreased 20% (`beam_ratio=0.4`).

Using **PROP_MAGNIFY()** to resample both the Nyquist and Nyquist/1.2 sampled PSFs to 2x finer than Nyquist, it is evident in Figure 13 that the result of using the more finely sampled source PSF has fewer interpolation artifacts (black pixels that indicated erroneously negative values).

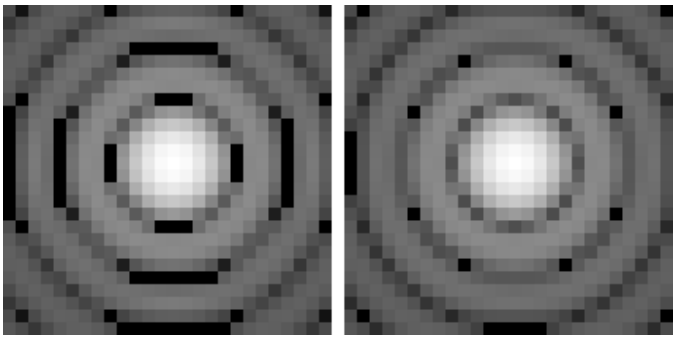


Figure 13. Central portions of PSFs computed at Nyquist (left) and 1.2x better than Nyquist sampling (right), resampled to 2x better than Nyquist using PROP_MAGNIFY().

IDL:

```
pro simple_telescope, wavefront, wavelength, gridsize, sampling, PASSVALUE=optval

d_objective = 0.060d                ;-- objective diameter in meters
fl_objective = 15.0 * d_objective   ;-- objective focal length in meters
fl_eyepiece = 0.021d                ;-- eyepiece focal length
fl_eye = 0.022d                     ;-- human eye focal length
beam_ratio = 0.5                    ;-- initial beam width/grid width

prop_begin, wavefront, d_objective, wavelength, gridsize, beam_ratio

prop_circular_aperture, wavefront, d_objective/2.0
prop_define_entrance, wavefront

prop_lens, wavefront, fl_objective, 'objective'

prop_propagate, wavefront, fl_objective+fl_eyepiece, 'eyepiece'
prop_lens, wavefront, fl_eyepiece, 'eyepiece'

exit_pupil_distance = fl_eyepiece / (1.0 - fl_eyepiece/(fl_objective+fl_eyepiece))
prop_propagate, wavefront, exit_pupil_distance, 'exit pupil at eye lens'
prop_lens, wavefront, fl_eye, 'eye'

prop_propagate, wavefront, fl_eye, 'retina'

prop_end, wavefront, sampling

return
end
```

To execute this prescription at a wavelength of 0.5 μm using a grid size of 512 by 512 elements, issue the command:

```
prop_run, 'simple_telescope', psf, 0.5, 512
```

Python:

```
import proper

def simple_telescope(wavelength, gridsize):

    d_objective = 0.060                         # objective diameter in meters
    fl_objective = 15.0 * d_objective           # objective focal length in meters
    fl_eyepiece = 0.021                         # eyepiece focal length
    fl_eye = 0.022                              # human eye focal length
    beam_ratio = 0.5                            # initial beam width/grid width

    wfo = proper.prop_begin(d_objective, wavelength, gridsize, beam_ratio)

    proper.prop_circular_aperture(wfo, d_objective/2)
    proper.prop_define_entrance(wfo)

    proper.prop_lens(wfo, fl_objective, "objective")

    proper.prop_propagate(wfo, fl_objective+fl_eyepiece, "eyepiece")
    proper.prop_lens(wfo, fl_eyepiece, "eyepiece")

    exit_pupil_distance = fl_eyepiece / (1 - fl_eyepiece/(fl_objective+fl_eyepiece))
    proper.prop_propagate(wfo, exit_pupil_distance, "exit pupil at eye lens")
    proper.prop_lens(wfo, fl_eye, "eye")

    proper.prop_propagate(wfo, fl_eye, "retina")

    (wfo, sampling) = proper.prop_end(wfo)

    return (wfo, sampling)
```

To execute this prescription at a wavelength of 0.5 μm using a grid size of 512 by 512 elements, issue the command:

```
(psf, sampling) = proper.prop_run( 'simple_telescope', 0.5, 512 )
```

Matlab:

```
function [wavefront, sampling] = simple_telescope(wavelength, gridsize)

  d_objective  = 0.06;               % objective diameter (m)
  fl_objective = 15.0 * d_objective; % focal length objective (m)
  fl_eyepiece = 0.021;               % focal length eyepiece (m)
  fl_eye = 0.022;                    % focal length human eye (m)
  beam_ratio = 0.5;                  % initial beam width/grid width

  wavefront   = prop_begin(d_objective, wavelength, gridsize, beam_ratio);
  wavefront   = prop_circular_aperture(wavefront, d_objective / 2.0);
  wavefront   = prop_define_entrance(wavefront);
  wavefront   = prop_lens(wavefront, fl_objective, 'objective');

  wavefront   = prop_propagate(wavefront, fl_objective + fl_eyepiece, 'surface_name', …
                'eyepiece');
  wavefront   = prop_lens(wavefront, fl_eyepiece, 'eyepiece');

  exit_pupil_distance  = fl_eyepiece / (1.0d0 - fl_eyepiece / (fl_objective + fl_eyepiece));
  wavefront   = prop_propagate(wavefront, exit_pupil_distance, 'surface_name', …
                'exit pupil at eye lens');
  wavefront   = prop_lens(wavefront, fl_eye, 'eye');

  wavefront   = prop_propagate(wavefront, fl_eye, 'surface_name', 'retina');

  [wavefront, sampling] = prop_end(wavefront);

end
```

To execute this prescription at a wavelength of 0.5 μm using a grid size of 512 by 512 elements, issue the command:

```
[psf, sampling] = prop_run( 'simple_telescope', 0.5, 512 )
```

The Hubble Space Telescope

The Hubble Space Telescope has a 2.4 m diameter primary mirror and smaller secondary mirror in a Ritchey-Chretien configuration (a variant of the Cassegrain design). As just about everyone knows, the primary was figured to the wrong shape due to errors in the measurement setup, causing significant spherical aberration. This problem was not discovered until images of stars were taken on-orbit. Optics installed by astronauts during later servicing missions compensated for the error and allowed Hubble to operate as it was intended.

The Hubble telescope prescription below includes the primary and secondary mirrors, represented by simple lenses (not conic mirrors, like in the real telescope). In this example, the system does not have the spherical aberration problem. The aperture function includes the shadows of the secondary mirror and support vanes and the primary mirror pads (see Figure 2).

Hubble is focused by moving the secondary mirror away or towards the primary, keeping the distance between the back of the primary mirror and the focal plane fixed. This method is reproduced here using the variable `delta_sec`. Moving the secondary increases the path length between it and the primary and also between it and the focal plane. `delta_sec` is defined by the optional *PASSVALUE* keyword, defaulting to zero if it is not specified in the call to **PROP_RUN**.

Note that because HST in reality uses conic optics, the effects of defocus, etc., will actually be different than presented here (which are merely for demonstration purposes).

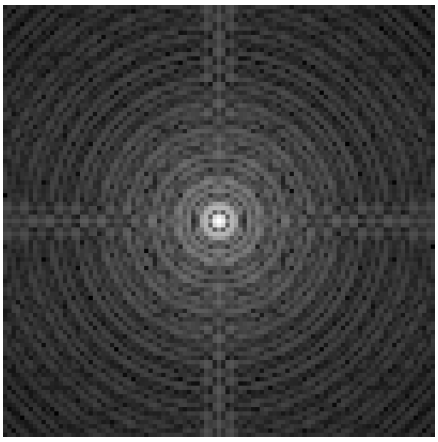


Figure 14. Central 100 x 100 pixel region of a simulated in-focus PSF generated using the simple Hubble Space Telescope prescription, displayed with a logarithmic intensity stretch.

IDL:

```
pro hubble_simple, wavefront, wavelength, grid_n, sampling, PASSVALUE=delta_sec

diam = 2.4d                   ;-- telescope diameter in meters

fl_pri = 5.52085d             ;-- HST primary focal length (m)
d_pri_sec = 4.907028205d      ;-- primary to secondary separation (m)
fl_sec = -0.6790325d          ;-- HST secondary focal length (m)
d_sec_to_focus = 6.3919974d   ;-- nominal distance from secondary to focus

beam_ratio = 0.5

;-- delta_sec = additional primary-to-secondary separation offset (m)

if ( n_elements(delta_sec) eq 0 ) then delta_sec = 0.0

prop_begin, wavefront, diam, wavelength, grid_n, beam_ratio

;-- create entrance aperture pattern

prop_circular_aperture, wavefront, diam/2                    ;-- primary mirror
prop_circular_obscuration, wavefront, 0.396                  ;-- secondary obscuration
prop_rectangular_obscuration, wavefront, 0.0264, 2.5         ;-- secondary vane (vert)
prop_rectangular_obscuration, wavefront, 2.5, 0.0264         ;-- secondary vane (horiz)
prop_circular_obscuration, wavefront, 0.078, -0.9066, -0.5538 ;-- primary mirror pad 1
prop_circular_obscuration, wavefront, 0.078, 0., 1.0705      ;-- primary mirror pad 2
prop_circular_obscuration, wavefront, 0.078, 0.9127, -0.5477  ;-- primary mirror pad 3

prop_define_entrance, wavefront

prop_lens, wavefront, fl_pri, 'primary'                      ;-- primary mirror

prop_propagate, wavefront, d_pri_sec+delta_sec, 'secondary'
prop_lens, wavefront, fl_sec, 'secondary'                    ;-- secondary mirror

prop_propagate, wavefront, d_sec_to_focus+delta_sec, 'HST focus'

prop_end, wavefront, sampling

return
end
```

To execute this prescription at λ=0.5 μm using a 512 x 512 grid, the call to **PROP_RUN** looks like this:

```
prop_run, 'hubble_simple', psf, 0.5, 512, sampling
```

Python:

```
import proper


def hubble_simple(wavelength, gridsize, PASSVALUE = {'delta_sec': 0.}):
    diam = 2.4                                  # telescope diameter in meters
    fl_pri = 5.52085                            # HST primary focal length (m)
    d_pri_sec = 4.907028205                     # primary to secondary separation (m)
    fl_sec = -0.6790325                         # HST secondary focal length (m)
    d_sec_to_focus = 6.3919974                  # nominal distance from secondary to focus
    beam_ratio = 0.5                            # initial beam width/grid width

    # delta_sec = additional primary-to-secondary separation offset (m)

    delta_sec = PASSVALUE['delta_sec']
```

```
    wfo = proper.prop_begin(diam, wavelength, gridsize, beam_ratio)

    proper.prop_circular_aperture(wfo, diam/2)                          # primary mirror
    proper.prop_circular_obscuration(wfo, 0.396)                        # secondary obscuration
    proper.prop_rectangular_obscuration(wfo, 0.0264, 2.5)               # secondary vane (vert)
    proper.prop_rectangular_obscuration(wfo, 2.5, 0.0264)               # secondary vane (horiz)
    proper.prop_circular_obscuration(wfo, 0.078, -0.9066, -0.5538)      # primary mirror pad 1
    proper.prop_circular_obscuration(wfo, 0.078, 0., 1.0705)            # primary mirror pad 2
    proper.prop_circular_obscuration(wfo, 0.078, 0.9127, -0.5477)       # primary mirror pad 3

    proper.prop_define_entrance(wfo)

    proper.prop_lens(wfo, fl_pri, "primary")                            # primary mirror

    proper.prop_propagate(wfo, d_pri_sec+delta_sec, "secondary")
    proper.prop_lens(wfo, fl_sec, "secondary")

    proper.prop_propagate(wfo, d_sec_to_focus+delta_sec, "HST focus", TO_PLANE = False)

    (wfo, sampling) = proper.prop_end(wfo)

    return (wfo, sampling)
```

To execute this prescription at λ=0.5 μm using a 512 x 512 grid, the call to **PROP_RUN** looks like this:

```
(psf, sampling) = proper.prop_run( 'hubble_simple', 0.5, 512 )
```

Matlab:

```
function [wfa, dx] = hubble_simple(lambda, gridsize, delta_sec)

  if nargin < 3
    delta_sec = 0.0;                % delta primary to secondary spacing (m)
  end

  diam = 2.4;                       % diameter of telescope (m)
  fl_pri = 5.52085;                 % focal length mirror 1 (m)
  d_pri_sec = 4.907028205;          % mirror 1 to mirror 2  (m)
  fl_sec = -0.6790325;              % focal length mirror 2 (m)
  d_sec_focus = 6.3919974;          % mirror 2 to focus     (m)
  beam_ratio = 0.5;

  wavefront = prop_begin(diam, lambda, gridsize, beam_ratio);

  wavefront = prop_circular_aperture(wavefront, diam / 2.0);    % primary edge
  wavefront = prop_circular_obscuration(wavefront, 0.396);      % secondary obscuration
  wavefront = prop_rectangular_obscuration(wavefront, 0.0264, 2.5);  % vertical vanes
  wavefront = prop_rectangular_obscuration(wavefront, 2.5, 0.0264);  % horizontal vanes
  wavefront = prop_circular_obscuration(wavefront, 0.078, 'xc', -0.9066, 'yc', -0.5538);
  wavefront = prop_circular_obscuration(wavefront, 0.078, 'xc', 0.0, 'yc', 1.0705);
  wavefront = prop_circular_obscuration(wavefront, 0.078, 'xc', 0.9127, 'yc', -0.5477);

  wavefront = prop_define_entrance(wavefront);

  wavefront = prop_lens(wavefront, fl_pri, 'primary');

  wavefront = prop_propagate(wavefront, d_pri_sec + delta_sec, 'surface_name', 'secondary');

  wavefront = prop_lens(wavefront, fl_sec, 'secondary');

  wavefront = prop_propagate(wavefront, d_sec_focus + delta_sec, 'surface_name', 'HST focus');

  [wfa, dx] = prop_end(wavefront);

end
```

To run this example, type the following on the Matlab command line:

```
[psf, sampling] = prop_run( 'hubble_simple', 0.5, 512 )
```

The *PASSVALUE* keyword is not specified in this call, so the normal separation between the primary and secondary mirrors is used, producing an in-focus PSF at the focal plane. Note that the initial beam width/grid width ratio is explicitly set to 0.5 (which is the default assumed by **PROP_BEGIN** if the ratio is not specified). This sizing results in a Nyquist-sampled PSF at the focal plane (Figure 14). Finer sampling of the PSF can be obtained by reducing the initial beam diameter/grid width (*beam_ratio*), at the expense of reducing the extent of the PSF and the initial sampling of the beam.

Changing the Focus (and a detour discussion on errors and sampling)

The focus of this simplified Hubble can be changed by providing the secondary mirror offset from its nominal position by means of the *PASSVALUE* keyword. Let us now create PSFs with the secondary moved 40 μm closer and 40 μm further from the primary:

IDL:

```
prop_run, 'hubble_simple', psfa, 0.5, 512, samplinga, PASSVALUE=-40e-6
prop_run, 'hubble_simple', psfb, 0.5, 512, samplingb, PASSVALUE=40e-6
```

Python:

```
(psfa, samplinga) = proper.prop_run( 'hubble_simple', 0.5, 512,
                        PASSVALUE={'delta_sec':-40e-6} )
(psfb, samplingb) = proper.prop_run( 'hubble_simple', 0.5, 512,
                        PASSVALUE={'delta_sec':40e-6} )
```

Matlab:

```
[psfa, samplinga] = prop_run( 'hubble_simple', 0.5, 512, 'PASSVALUE', -40e-6 );
[psfb, samplingb] = prop_run( 'hubble_simple', 0.5, 512, 'PASSVALUE', 40e-6 );
```

The PSFs for the defocused system are shown in Figure 15. There are some things to notice here. First, the samplings of both are significantly different than for the in-focus PSF ($\Delta_{\text{in-focus}}$=6.0 μm, Δ_{-40}=6.6 μm, Δ_{+40}=7.7 μm). The PSFs are sufficiently defocused so that they are in the far-field region, thus their sampling is proportional to their respective distances from the beam waist (focus). This ensures that the beam, as it expands from or contracts towards focus, occupies approximately the same region of the wavefront array.

Rectangular grid patterns can also be seen in the PSFs. These are computational artifacts caused by wrap-around errors due to the Fourier transforms. As can be seen in the log-stretched images, the wings of the PSFs have sufficient signal that they wrap-around the sides of the array, creating interference patterns. Working to suppress such artifacts is one of the primary headaches of using near-field/far-field propagators.

Let us first try a larger grid size, 1024 x 1024 instead of 512 x 512. As shown in Figure 16, increasing the grid size while maintaining the same beam/grid width ratio does not really help to suppress the artifacts.

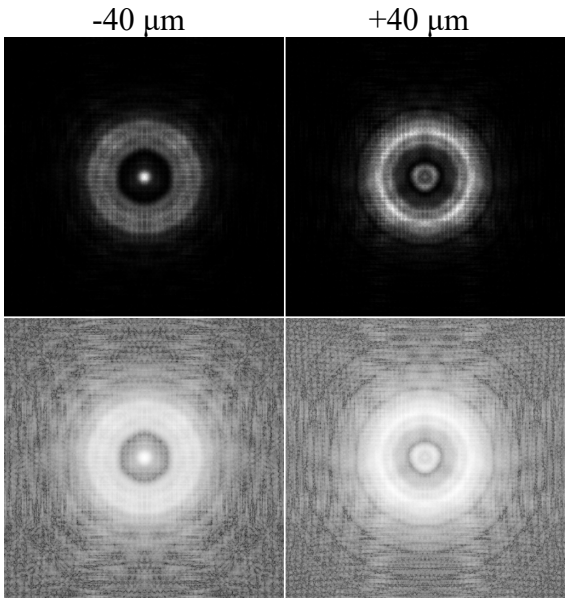


Figure 15. Simulated PSFs from the simplified Hubble prescription with the secondary mirror moved 40 μm toward (-) and away (+) from the primary mirror relative to its nominal position. The top panels have linear intensity stretches and the bottom are logarithmic. Wrap-around errors from the FFTs cause the grid-like patterns seen in the images. The entire 512 x 512 grids are shown.

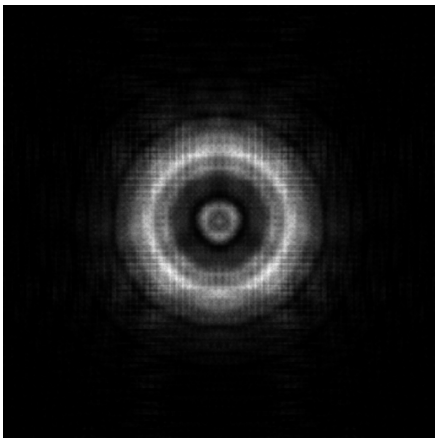


Figure 16. Simulated PSF for the simple Hubble with the secondary moved +40 μm from the primary. The grid size was 1024 x 1024 (shown rebinned to 512 x 512) and the initial beam/grid size ratio was 0.5. Increasing the grid size relative to that used in Figure 16 does not reduce the grid-pattern artifacts.

Next, we can try the combination of increasing the grid size and reducing the initial beam/grid width ratio. Choosing 1024 x 1024 with a beam ratio of 0.25 will create an initial beam with 256 samples across it, just the same as in the 512 x 512 case with a beam ratio of 0.5. By doing this, additional zero padding is created that allows the wings to extend further before wrapping. As shown in Figure 17, this greatly reduces the errors, though some are still visible. Increasing the grid size further while proportionally decreasing the beam ratio to maintain constant beam sampling will suppress the errors even more.

A side effect of this technique is that while the sampling remains constant in the far field, in the near field it will vary with the beam ratio. A smaller beam ratio results in finer sampling in the near field (the PSF looks bigger). However, and conversely to the far-field case, this also reduces the area into which the wings can expand, increasing the chances of wrap-around. This is just one of those things that has to be dealt with when using near field/far field propagators.

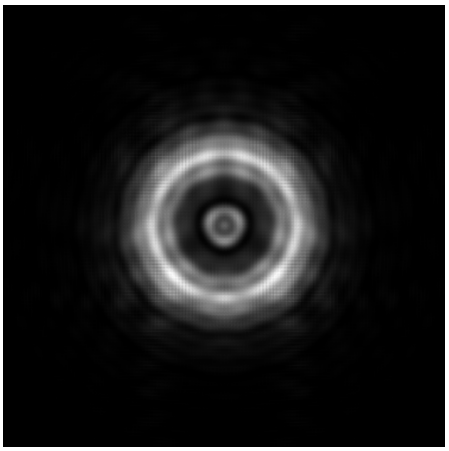


Figure 17. Like Figure 16 except that the initial beam/grid width ratio was 0.25, creating more room for the PSF wings to extend before wrapping. The central 512 x 512 region of the 1024 x 1024 array is shown.

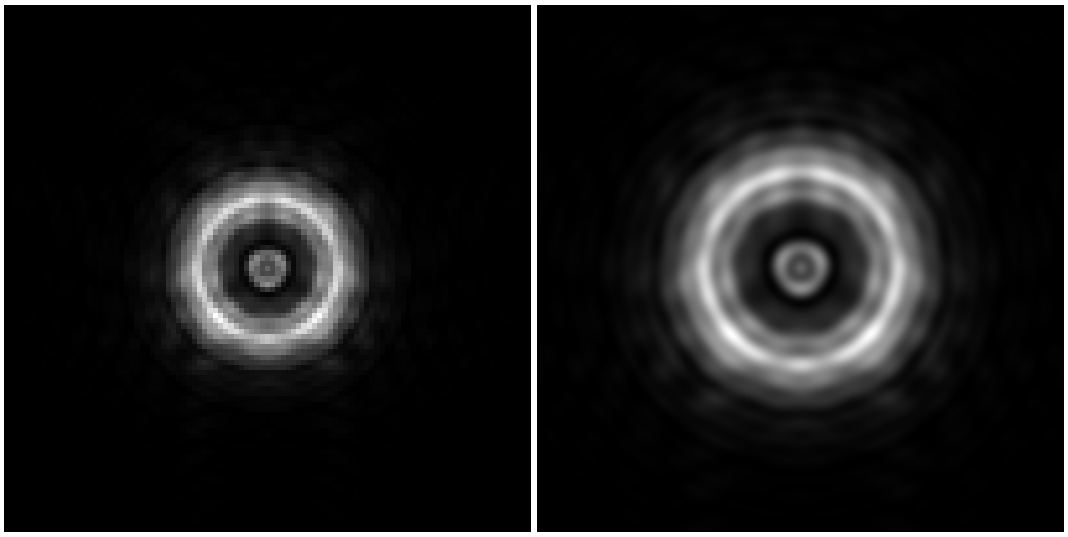


Figure 18. (Left) Simulated PSF generated with a 512 x 512 grid and beam ratio of 0.4 with forced propagation using *TO_PLANE*. The central 100 x 100 pixels are shown. (Right) PSF on left interpolated to 7x finer sampling using PROP_MAGNIFY. The entire 512 x 512 interpolated region is shown.

Now let's try another trick. The amount of defocus we have applied is sufficiently small that we can try forcing the PROPER routines to use the near-field propagator further away from focus than they normally would. This can be done using the *TO_PLANE* switch in the call to **PROP_PROPAGATE** that propagates the beam the final step. Recall when traveling from one surface to another, the PROPER routines propagate to the focus of the current beam and then to the next surface. By specifying *TO_PLANE*, the angular spectrum propagator is used to go from focus to the desired position, and the result will have the same sampling as it would at focus. For the simplified Hubble, we can try this by modifying the last **PROP_PROPAGATE** line:

IDL: `prop_propagate, wavefront, d_sec_to_focus+delta_sec, /TO_PLANE`

Python: `proper.prop_propagate( wavefront, d_sec_to_focus+delta_sec, TO_PLANE=True )`

Matlab: `prop_propagate( wavefront, d_sec_to_focus+delta_sec, 'TO_PLANE' );`

Using a 512 x 512 grid with a beam ratio of 0.4 and *TO_PLANE* produces a PSF (Figure 18) that is more coarsely sampled than those generated above. Using **PROP_MAGNIFY** to interpolate this to finer sampling produces a result equivalent to that created using a 4096 x 4096 grid with 0.0625 beam ratio and default propagation, but in only 2% of the time.

The Talbot Effect

The *Talbot effect* is the name for the curious phenomenon that occurs when a wavefront has a spatially-periodic amplitude pattern of period *P*. As it propagates through space, the amplitude pattern evolves into a phase pattern with a period of *P* and a reduced amplitude pattern with a period of *P/2*. It then turns into an amplitude pattern of period *P* but with reversed contrast, and then back into a phase pattern. Finally, it ends up as the same amplitude pattern it began as. The Talbot length, L_T, the distance over which the amplitude pattern reconstitutes itself, is $L_T=2P^2/\lambda$, where λ is the wavelength. A discussion of the Talbot effect is provided by Goodman in *Introduction to Fourier Optics*.

The PROPER routines can be used to simulate the Talbot effect. The prescription below, `talbot`, creates a 128 x 128 wavefront array with a cosine amplitude pattern (*i.e.* an amplitude grating) varying along the X axis. The caller passes to the prescription the spatial period of the pattern in meters and the distance over which to propagate. The *TO_PLANE* switch on **PROP_PROPAGATE** forces it to propagate to a plane to make interpreting the phase values easier (it would do this anyway for the examples that follow, but this just emphasizes the point). The *NOABS* switch in the call to **PROP_END** tells it to return the complex-amplitude wavefront array rather than the modulus-square of the wavefront, so that the amplitude and phase can be examined.

IDL:

```
pro talbot, wavefront, wavelength, gridsize, sampling, PASSVALUE=optval

talbot_length = 2.0d * optval.period^2 / wavelength

beam_ratio = 0.5
prop_begin, wavefront, optval.diam, wavelength, gridsize, beam_ratio

;-- create 1-D grating pattern

m = 0.2 ;-- pattern amplitude
x = (findgen(gridsize) - gridsize/2) * prop_get_sampling(wavefront)
grating = 0.5 * (1 + m * cos(2 * !pi * x / optval.period))

;-- create 2-D amplitude grating pattern

grating = grating # replicate(1.0,gridsize)

prop_multiply, wavefront, grating
prop_define_entrance, wavefront

prop_propagate, wavefront, optval.dist, /TO_PLANE

prop_end, wavefront, sampling, /NOABS

return
end
```

Python:

```
import proper
import numpy as np

def talbot(wavelength, gridsize, PASSVALUE = {'period': 0., 'diam': 0., 'dist': 0.}):
    talbot_length = 2. * PASSVALUE['period']**2 / wavelength

    beam_ratio = 0.5
    wfo = proper.prop_begin(PASSVALUE['diam'], wavelength, gridsize, beam_ratio)

    # create 1-D grating pattern

    m = 0.2
    x = (np.arange(gridsize, dtype = np.float64) - gridsize/2)  \
                * proper.prop_get_sampling(wfo)
    grating = 0.5 * (1 + m * np.cos(2*np.pi*x/PASSVALUE['period']))

    # create 2-D amplitude grating pattern

    grating = np.dot(grating.reshape(gridsize,1), np.ones([1,gridsize], dtype = np.float64))

    proper.prop_multiply(wfo, grating)
    proper.prop_define_entrance(wfo)

    proper.prop_propagate(wfo, PASSVALUE['dist'], TO_PLANE = True)

    (wfo, sampling) = proper.prop_end(wfo, NOABS = True)

    return (wfo, sampling)
```

Matlab:

```
function [wavefront, sampling] = talbot(wavelength, gridsize, optval)

  talbot_length = 2.0 * optval.period^2 / wavelength;

  wavefront = prop_begin(optval.diam, wavelength, gridsize);

  % Create 1-D grating pattern

  m =  0.2;    % pattern amplitude
  x = ((1:gridsize)-fix(gridsize/2)-1)*prop_get_sampling(wavefront);

  % Create 2-D amplitude grating pattern

  [gx, gy] = meshgrid(x, x);
  grating = 0.5 * (1.0 + m * cos(2.0*pi*gx/optval.period));

  wavefront = prop_multiply(wavefront, grating);
  wavefront = prop_define_entrance(wavefront);

  wavefront = prop_propagate(wavefront, optval.dist, 'to_plane');

  [wavefront, sampling] = prop_end(wavefront, 'noabs');

end
```

The demo program listed below, `talbot_demo`, will propagate the wavefront every 1/8th of the Talbot length for one length and plot the amplitude and phase at each interval.

IDL:

```
pro talbot_demo

diam = 0.1                  ;-- beam diameter in meters
period = 0.04               ;-- period of cosine pattern (meters)
wavelength_microns = 0.5d
wavelength_m = wavelength_microns * 1.0e-6
n = 128
nseg = 9
talbot_length = 2 * period^2 / wavelength_m
delta_length = talbot_length / (nseg - 1.0)

window, xsize=500, ysize=900
!p.multi = [0, 2, nseg]     ;-- setup a 2 by "nseg" panel of plots

z = 0.0d
for i = 1, nseg do begin
      prop_run, 'talbot', wavefront, wavelength_microns, n, $
              PASSVALUE={diam:diam, period:period, dist:z}

      ;-- extract a horizontal cross-section of array
      wavefront = wavefront(*,n/2)

      amp = abs(wavefront)
      amp = amp - mean(amp)
      phase = atan(wavefront, /phase)
      phase = phase - mean(phase)

      plot, amp, yran=[-0.0013,0.0013], xstyle=1, ystyle=1
      plot, phase, yran=[-0.25,0.25], xstyle=1, ystyle=1

      z = z + delta_length
endfor

!p.multi = 0

end
```

Python:

```
import matplotlib.pyplot as plt

def talbot_demo():
    diam = 0.1                  # beam diameter in meters
    period = 0.04               # period of cosine pattern (meters)
    wavelength_microns = 0.5
    wavelength_m = wavelength_microns * 1.e-6
    n = 128

    nseg = 9
    talbot_length = 2 * period**2 / wavelength_m
    delta_length = talbot_length / (nseg - 1.)

    z = 0.

    plt.close('all')
    f = plt.figure(figsize = (8, 18))

    for i in range(nseg):
        (wavefront, sampling) = proper.prop_run('talbot',
             wavelength_microns, n,
             PASSVALUE = {'diam': diam, 'period': period, 'dist': z})

        # Extract central cross-section of array
        wavefront = wavefront[:,n/2]
```

```
        amp = np.abs(wavefront)
        amp -= np.mean(amp)
        phase = np.arctan2(wavefront.imag, wavefront.real)
        phase -= np.mean(phase)

        ax1 = f.add_subplot(nseg,2,2*i+1)
        ax1.set_ylim(-0.0015, 0.0015)
        ax1.plot(amp)
        ax2 = f.add_subplot(nseg,2,2*i+2)
        ax2.set_ylim(-0.25, 0.25)
        ax2.plot(phase)

        z += delta_length

    f.tight_layout()
    plt.show()

    return

if __name__ == '__main__':
    talbot_demo()
```

Matlab:

```
  diam = 0.1;          % beam diameter (m)
  n = 128;             % number of pixels
  nseg = 9;            % number of segments
  period = 0.04;       % period of cosine pattern (m)
  z = 0.0;             % propagation distance (m)
  wavelength_microns = 0.5;
  wavelength_m = wavelength_microns * 1.0d-6;

  talbot_length = 2.0 * period^2 / wavelength_m;
  delta_length = talbot_length / (nseg - 1);

  figarray(nseg, 2);

  ifig = 0;
  for i = 1 : nseg
    ov = struct('diam', diam, 'dist', z, 'period', period);
    wavefront = prop_run('talbot', wavelength_microns, n, 'PASSVALUE', ov);

    % Extract a horizontal cross-section of array
    wavefront = wavefront(fix(n / 2) + 1, :);

    amp = abs(wavefront);
    amp = amp - mean(amp);
    pha = phase(wavefront);
    pha = pha - mean(pha);

    ifig = ifig + 1;
    figplace(ifig);
    clf;
    axes('FontSize', 16);
    plot(amp);
    axis([1, n, -0.0013, 0.0013]);
    set(get(gcf, 'CurrentAxes'), 'FontSize', 16);

    ifig = ifig + 1;
    figplace(ifig);
    clf;
    axes('FontSize', 16);
    plot(pha);
    axis([1, n, -0.25, 0.25]);
    set(get(gcf, 'CurrentAxes'), 'FontSize', 16);

    z    = z + delta_length;
  end
```

The results are plotted to the screen. A touched-up versions of the plots are shown in the left panel of Figure 19. As the plots show, the amplitude pattern becomes a phase pattern at $\frac{1}{4}L_T$ with a reduced amplitude pattern with half the spatial period. At $1L_T$ (and multiples thereof) it returns to its initial state and then repeats this process over again.

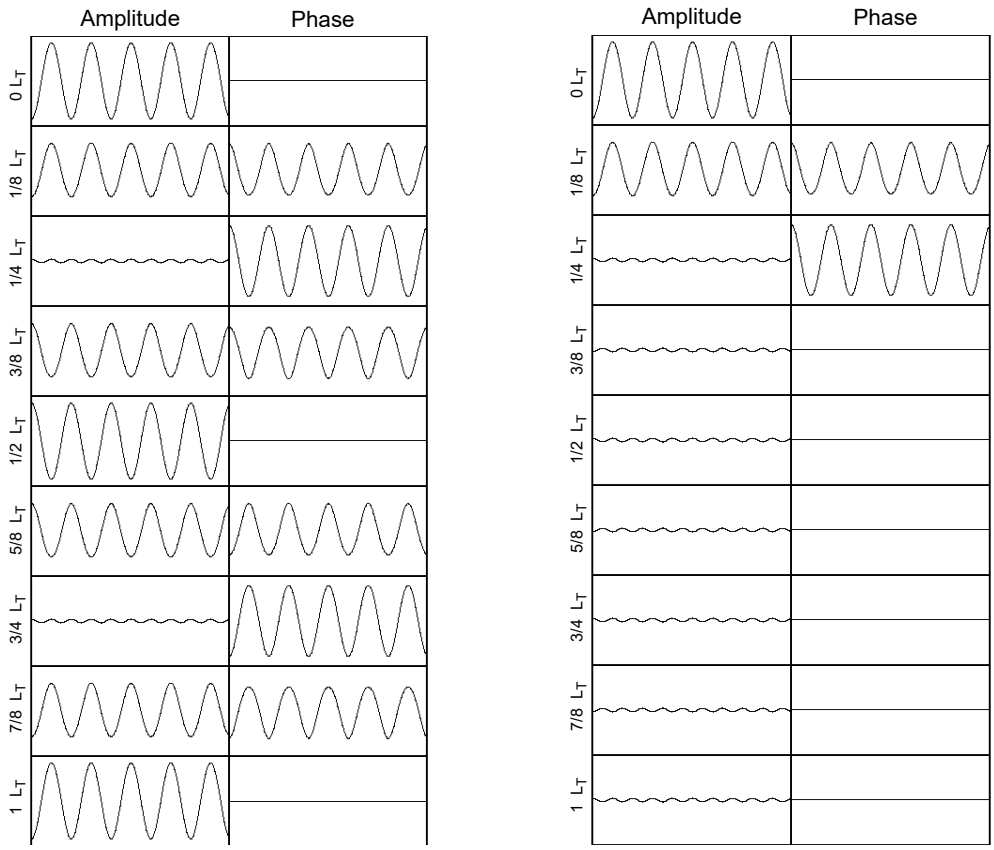


Figure 19. Cross-sectional plots of the amplitude and phase of a two-dimensional (*x*,*y*) wavefront that initially has a amplitude pattern of spatial period *P* along the *x* direction. The wavefront is propagated over various distances *z*, which are shown as fractions of the Talbot length ($L_T=2P^2/\lambda$) from the initial position. (Left) The evolution of the wavefront over the specified propagation distance. (Right) The same as the left panel, except that the phase is negated at $\frac{1}{4}L_T$ (shown before negation at that position).

That a spatially-periodic amplitude pattern turns into a phase pattern at $\frac{1}{4}L_T$ introduces the possibility of correcting amplitude wavefront errors with phase-modifying devices, such as deformable mirrors. A modification to `talbot` can demonstrate this (creating `talbot_correct`). The call to **PROP_PROPAGATE** is replaced with the following (in IDL, with something similar in Python or Matlab):

```
if ( optval.dist le 0.25*talbot_length ) then begin
       prop_propagate, wavefront, optval.dist, /TO_PLANE
endif else begin
       prop_propagate, wavefront, 0.25*talbot_length, /TO_PLANE
       phase = prop_get_phase( wavefront )   ;-- in radians
       phase = phase * wavelength / (2 * !pi)  ;-- in meters
       prop_add_phase, wavefront, -phase
       prop_propagate, wavefront, optval.dist-0.25*talbot_length, /TO_PLANE
endelse
```

`talbot_demo` is modified to call `talbot_correct`, creating `talbot_correct_demo`.

If the wavefront is being propagated past $¼L_T$, it is first propagated to there. The phase is extracted using **PROP_GET_PHASE()**, converted from radians to meters, and the opposite phase added to the wavefront with **PROP_ADD_PHASE**, effectively zeroing-out the phase pattern created by the evolution of the initial amplitude pattern. The wavefront is then propagated the remaining distance. As shown in the right panel of Figure 19, compensating for the phase at $¼L_T$ negates the bulk of the amplitude pattern, leaving only the $P/2$ component, which is at a much lower level than the initial pattern. Note that L_T in this case is on the order of 6×10^{15} meters!

A Simple Microscope (and Objects at Finite Distances)

The example telescopes described previously assumed that the emitting source was at infinity or practically so (e.g. a star), so that the light entering the optical systems was collimated. However, when an emitting point source is at a finite distance from the system, it will create a spherically expanding wavefront, and the incoming light is no longer collimated. The *PROPER* routines cannot explicitly simulate propagation beginning from a point source at a finite distance because the first surface must be the entrance aperture of the system. There is a trick that will accomplish the same thing. An expanding spherical wavefront emitted from a source located at some distance d prior to the entrance aperture can be simulated by inserting a negative-power (concave) lens with a focal length of $-d$ at the aperture, creating the same phase variation.

An optical system in which the object is almost always at some close distance is a microscope. The simplest compound (i.e. two or more lenses) microscope (Figure 20) has an objective lens that creates an intermediate, magnified image of the object and an eyepiece (ocular) lens that further magnifies that image. The object is placed at distance d_{source} prior to the objective lens of focal length f_{obj}. The objective forms the intermediate image at a distance of $f_{obj}+L$, where L (known as the tube length) has been standardized to 16 cm by microscope makers. This image is located at the focus of the ocular lens having a focal length of f_{oc} that ideally forms a nearly-collimated beam, with the exit pupil located f_{oc} past the ocular. This is where the lens of the user's eye ($f_{eye} \approx 22$ mm) is positioned, with the eye focused at infinity to form the final image on the retina. In this simple system, the image is inverted.

A prescription for a simple compound microscope is given below. The objective (which is also the entrance aperture) has a diameter of 5 mm and focal length of 10 mm. An eyepiece lens of focal length 20 mm is used. The distance the object must be from the objective to form an in-focus image is computed using the Gaussian lens equation. The focus may be changed by specifying an offset from this distance using the *PASSVALUE* keyword to **PROP_RUN**. The exit pupil distance is determined from the lens equation as the location where the ocular forms an image of the objective. This program produces the PSF of the point source object as seen with the eye (it ignores the effects of changes in the indices of refraction).

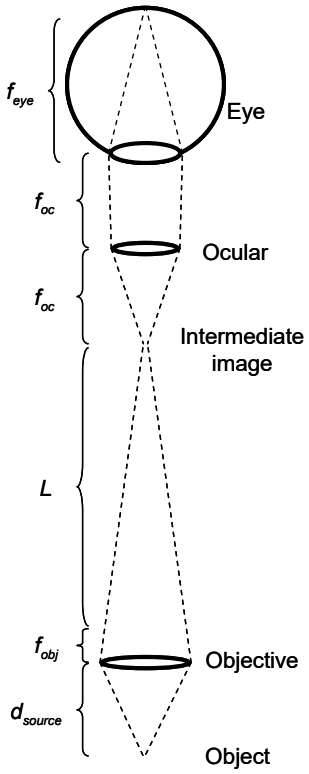


Figure 20. Schematic layout of a simple compound microscope.

IDL:

```
pro microscope, wavefront, wavelength, gridsize, sampling, PASSVALUE=focus_offset

d_objective = 0.005d      ;-- objective diameter in meters
fl_objective = 0.010d     ;-- objective focal length in meters
fl_eyepiece = 0.020d      ;-- eyepiece focal length
fl_eye = 0.022d           ;-- human eye focal length

beam_ratio = 0.4

prop_begin, wavefront, d_objective, wavelength, gridsize, beam_ratio

d1 = 0.160d               ;-- standard tube length
d_intermediate_image = fl_objective + d1

;-- compute in-focus distance of object from objective

d_object = 1 / (1/fl_objective - 1/d_intermediate_image)

prop_circular_aperture, wavefront, d_objective/2
prop_define_entrance, wavefront

;-- simulate the diverging wavefront emitted from a point source placed
;-- "d_object" in front of the objective by using a negative lens (focal
;-- length = -d_object) placed at the location of the objective

if ( n_elements(focus_offset) eq 0 ) then focus_offset = 0.0

prop_lens, wavefront, -(d_object + focus_offset)
prop_lens, wavefront, fl_objective, 'objective'

prop_propagate, wavefront, d_intermediate_image, 'intermediate image'
prop_propagate, wavefront, fl_eyepiece, 'eyepiece'
prop_lens, wavefront, fl_eyepiece, 'eyepiece'
exit_pupil_distance = fl_eyepiece / (1 - fl_eyepiece/(d_intermediate_image+fl_eyepiece))
```

```
    prop_propagate, wavefront, exit_pupil_distance, 'exit pupil/eye'

    prop_lens, wavefront, fl_eye, 'eye'
    prop_propagate, wavefront, fl_eye, 'retina'

    prop_end, wavefront, sampling

    return
    end
```

Python:

```
import proper

def microscope(wavelength, gridsize, PASSVALUE = {'focus_offset': 0.}):

    # Define entrance aperture diameter and other quantities
    d_objective = 0.005                     # objective diameter in meters
    fl_objective = 0.010                    # objective focal length in meters
    fl_eyepiece = 0.020                     # eyepiece focal length
    fl_eye = 0.022                  # human eye focal length

    beam_ratio = 0.4

    # Define the wavefront
    wfo = proper.prop_begin(d_objective, wavelength, gridsize, beam_ratio)

    d1 = 0.160                          # standard tube length
    d_intermediate_image = fl_objective + d1

    # Compute in-focus distance of object from objective
    d_object = 1 /(1/fl_objective - 1/d_intermediate_image)

    # Define a circular aperture
    proper.prop_circular_aperture(wfo, d_objective/2.)

    # Define entrance
    proper.prop_define_entrance(wfo)

    # simulate the diverging wavefront emitted from a point source placed
    # "d_object" in front of the objective by using a negative lens (focal
    # length = -d_object) placed at the location of the objective

    focus_offset = PASSVALUE['focus_offset']

    # Define a lens
    proper.prop_lens(wfo, -(d_object + focus_offset))
    proper.prop_lens(wfo, fl_objective, "objective")

    # Propagate the wavefront
    proper.prop_propagate(wfo, d_intermediate_image, "intermediate image")
    proper.prop_propagate(wfo, fl_eyepiece, "eyepiece")
    proper.prop_lens(wfo, fl_eyepiece, "eyepiece")
    exit_pupil_distance = fl_eyepiece / (1 - fl_eyepiece/(d_intermediate_image+fl_eyepiece))
    proper.prop_propagate(wfo, exit_pupil_distance, "exit pupil/eye")

    proper.prop_lens(wfo, fl_eye, "eye")
    proper.prop_propagate(wfo, fl_eye, "retina")

    # End
    (wfo, sampling) = proper.prop_end(wfo)

    return (wfo, sampling)
```

Matlab:

```
function [wavefront, sampling] = microscope(wavelength, gridsize, focus_offset)
  d_objective = 0.005;        % objective diameter (m)
  fl_objective = 0.01;        % focal length objective (m)
  fl_eyepiece = 0.02;         % focal length eyepiece (m)
  fl_eye = 0.022;             % focal length human eye (m)
  beam_ratio = 0.4;           % initial beam width/grid width
  d1 = 0.16;                  % standard tube length (m)
  d_intermediate_image = fl_objective + d1;    % intermediate image distance (m)

  % Compute in-focus distance of object from objective
  d_object = 1.0 / (1.0 / fl_objective - 1.0 / d_intermediate_image);

  wavefront = prop_begin(d_objective, wavelength, gridsize, beam_ratio);
  wavefront = prop_circular_aperture(wavefront, d_objective / 2.0);
  wavefront = prop_define_entrance(wavefront);

  % Simulate the diverging wavefront emitted from a point source placed
  % "d_object" in front of the objective by using a negative lens (focal
  % length = -d_object) placed at the location of the objective.

  if (nargin < 3)
    focus_offset = 0.0;       % focus offset (m)
  end

  wavefront = prop_lens(wavefront, -(d_object + focus_offset));
  wavefront = prop_lens(wavefront, fl_objective, 'objective');

  wavefront = prop_propagate(wavefront, d_intermediate_image, …
               'surface_name', 'intermediate image');
  wavefront = prop_propagate(wavefront, fl_eyepiece, ...
               'surface_name', 'eyepiece');
  wavefront = prop_lens(wavefront, fl_eyepiece, 'eyepiece');

  exit_pupil_distance = fl_eyepiece / ...
              (1.0 - fl_eyepiece / (d_intermediate_image + fl_eyepiece));
  wavefront = prop_propagate(wavefront, exit_pupil_distance, ...
               'surface_name', 'exit pupil/eye');

  wavefront = prop_lens(wavefront, fl_eye, 'eye');
  wavefront = prop_propagate(wavefront, fl_eye, 'surface_name', 'retina', 'to_plane');

  [wavefront, sampling] = prop_end(wavefront);

end
```

A Stellar Coronagraph

When a telescope looks at a star, the star's light is diffracted by the edges of the telescope aperture and any obscurations that may be in the system (e.g. a secondary mirror). This creates the point spread function (PSF) that, if the optics are good, has a sharp, narrow core and faint wings. If an astronomer is using the telescope to look for something faint near that star, such as a planet, the light in the wings of the stellar PSF will overwhelm that from the planet. For example, if one were to travel to the nearest star (4.3 light years away), and look back at our own solar system with a telescope, Jupiter would be about 10^9 times fainter than the Sun and appear no further than 3.7 arcseconds from it. Earth would be 10^{10} times fainter than the Sun and would be no further than 0.7 arcsec away. With a 2.4 meter diameter telescope (circular, no obscurations, perfect optics) and looking at visible wavelengths (λ = 0.5 μm), the signal from Jupiter would be about 150 times fainter than the wings of the PSF at the same location. With real optics that have figuring and polishing errors that scatter light, Jupiter would be about 2000 times fainter than the PSF wings. A method is needed to get rid of the light from the star, allowing the planet to be seen.

Simply blocking the light from the star with some sort of occulter will not solve the problem – the PSF wings will remain unsuppressed (it does help reduce scatter from optical surfaces after the occulter, though). In 1939, Bernard Lyot devised a solution that makes use of the wave-like nature of light to suppress the wings (his goal was to observe the corona of the Sun rather than stars outside of the Solar System). The image of the star is first focused onto an occulter of some sort (e.g. a small disk), sized so that it will not block the image of the surrounding object of interest (e.g. a planet or circumstellar dust disk). The light from the PSF wings then passes to a lens or mirror that creates a collimated beam, forming an image of the entrance pupil in the Lyot plane. Because of the wave-like nature of light, the occulter acts as a spatial filter and concentrates the residual flux into the regions of the pupil corresponding to the diffracting edges in the entrance pupil. These portions of the pupil are blocked with a mask (the Lyot stop), removing most of the remaining light from the star. Sources beyond the radius of the occulter are not filtered and pass through the system essentially unaffected (except for attenuation by the Lyot stop). Another lens or mirror recoverges the light, creating an image in the final focal plane. A simple schematic of such a coronagraph is shown in Figure 21.

The effect of the coronagraph can be understood if one views the pupil and focal planes as Fourier transforms of each other, so that the amplitude distribution in the focal plane (the PSF) represents the frequency spectrum of the pupil. Blocking the central portion of the PSF with an occulter is thus equivalent to removing the low-spatial-frequency components of the pupil, leaving only the high ones (the edges). This explains the image at the Lyot stop. Note that as the size of the occulter increases, higher spatial frequencies in the pupil are suppressed, causing the regions around the edges in the Lyot plane to become narrower. This allows a more "open" or "less aggressive" Lyot stop to be used, increasing throughput for field sources (e.g. planets).

The narrowness of the residual light regions in the Lyot plane is also dependent on the shape of the occulter. A hard-edge spot will not concentrate the light around the edges very well (it essentially diffracts light in the image plane into the pupil). Occulters with graded transmissions, such as a Gaussian spot, work much better. Recent studies (Kuchner and Traub, *Astrophysical Journal*, 570, 900 (2002)) have invented "band-limited" occulters that theoretically concentrate all of the remaining light around the edges, providing full suppression of the stellar light (one of these occulters is implemented with **PROP_8TH_ORDER_MASK**).

A coronagraph only suppresses the diffraction pattern created by the edges in a system – it does nothing to the scattered light created by surface errors, which must be corrected using wavefront control techniques.

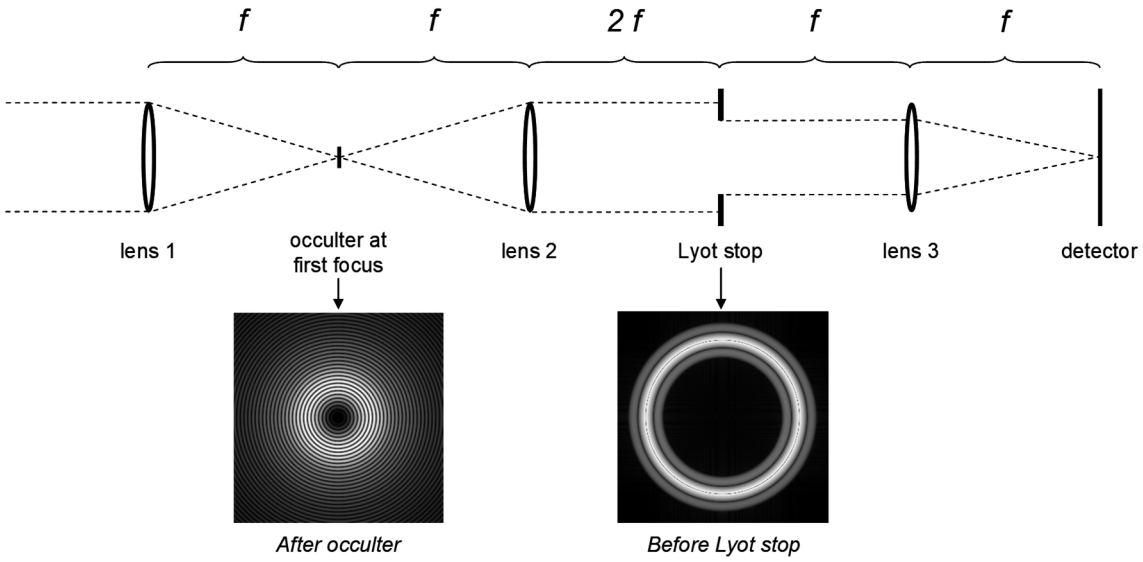


Figure 21. Schematic layout of a Lyot stellar coronagraph. For simplicity, all lenses have the same focal length, *f*. Collimated light (e.g. from a star) enters from the left. The image at the first focus is shown (square-root intensity stretch) multiplied by a circular, 8th-order occulter (shown here with 50% intensity transmission at $r=8\lambda/D$). The corresponding image at the Lyot plane, prior to masking by the Lyot stop, is also shown ($I^{0.2}$ intensity stretch). The Lyot stop would only pass the central dark region. The distance from the pupil reimaging lens (lens 2) to the Lyot stop is sometimes set to *f* rather than 2*f*. The difference is not usually significant, though 2*f* produces a true image of the entrance pupil.

A Simple Coronagraph with Selectable Occulters

Coronagraphs can be easily modeled using PROPER routines (in fact that is why the PROPER library was created). As an example, we can start off with the simple coronagraph shown in Figure 21. Because the coronagraph will be a part of a more complete optical system in a later example, it is defined as a separate function that will be called from a prescription routine. In `coronagraph.pro` listed below, different occulters can be used: a solid spot, an apodized Gaussian spot, and an 8th-order band-limited apodized occulter. Each has a matching Lyot stop that is appropriately sized and shaped. The occulter type is selected by assigning the parameter *occulter_type* to a string ('SOLID', 'GAUSSIAN', or '8TH_ORDER'). The occulter 50% intensity transmission radius (or solid spot radius) is set here to be 4λ/D radians (D = entrance aperture diameter). The routine will display the wavefront amplitude immediately after the occulter and then just before the Lyot stop.

IDL:

```
pro coronagraph, wavefront, fl_lens, occulter_type, diam

prop_lens, wavefront, fl_lens, 'coronagraph imaging lens'
prop_propagate, wavefront, fl_lens, 'occulter'

;-- occulter sizes are specified here in units of lambda/diameter;
;-- convert lambda/diam to radians then to meters

lambda = prop_get_wavelength( wavefront )
occrad = 4.0                                    ;-- occulter radius in lam/D
occrad_rad = occrad * lambda / diam     ;-- occulter radius in radians
dx_m = prop_get_sampling(wavefront)
dx_rad = prop_get_sampling_radians(wavefront)
occrad_m = occrad_rad * dx_m / dx_rad     ;-- occulter radius in meters
```

```
    case occulter_type of
      '8TH_ORDER' : prop_8th_order_mask, wavefront, occrad, /CIRCULAR
       'GAUSSIAN' : begin
                      r = prop_radius( wavefront )
                      h = sqrt(-0.5 * occrad_m^2 / alog(1 - sqrt(0.5)))
                      gauss_spot = 1 - exp(-0.5*(r/h)^2)
                      prop_multiply, wavefront, gauss_spot
                    end
          'SOLID' : prop_circular_obscuration, wavefront, occrad_m
    endcase

    tvscl, sqrt(prop_get_amplitude(wavefront))
    xyouts, 256, 10, 'After Occulter', CHARSIZE=2, ALIGN=0.5, /DEVICE

    prop_propagate, wavefront, fl_lens, 'pupil reimaging lens'
    prop_lens, wavefront, fl_lens, 'pupil reimaging lens'

    prop_propagate, wavefront, 2*fl_lens, 'lyot stop'

    tvscl, prop_get_amplitude(wavefront)^0.2, 512, 0
    xyouts, 768, 10, 'Before Lyot Stop', CHARSIZE=2, ALIGN=0.5, /DEVICE

    case occulter_type of
      '8TH_ORDER' : prop_circular_aperture, wavefront, 0.50, /NORM
       'GAUSSIAN' : prop_circular_aperture, wavefront, 0.25, /NORM
          'SOLID' : prop_circular_aperture, wavefront, 0.84, /NORM
    endcase

    prop_propagate, wavefront, fl_lens, 'reimaging lens'
    prop_lens, wavefront, fl_lens, 'reimaging lens'

    prop_propagate, wavefront, fl_lens, 'final focus'

    return
    end
```

The `coronagraph` procedure is called from the prescription `run_occulter`:

```
    pro run_occulter, wavefront, wavelength, grid_size, sampling, PASSVALUE=optval

    diam = 0.1d                             ;-- telescope diameter in meters
    fl_lens = 24 * diam
    beam_ratio = 0.3
    prop_begin, wavefront, diam, wavelength, grid_size, beam_ratio

    prop_circular_aperture, wavefront, diam/2
    prop_define_entrance, wavefront

    coronagraph, wavefront, fl_lens, optval.occulter_type, diam

    prop_end, wavefront, sampling

    return
    end
```

Python:

```
import proper
import numpy as np
import matplotlib.pylab as plt

def coronagraph(wfo, f_lens, occulter_type, diam):

    proper.prop_lens(wfo, f_lens, "coronagraph imaging lens")
    proper.prop_propagate(wfo, f_lens, "occulter")
```

```
    # occulter sizes are specified here in units of lambda/diameter;
    # convert lambda/diam to radians then to meters
    lamda = proper.prop_get_wavelength(wfo)
    occrad = 4.                          # occulter radius in lam/D
    occrad_rad = occrad * lamda / diam    # occulter radius in radians
    dx_m = proper.prop_get_sampling(wfo)
    dx_rad = proper.prop_get_sampling_radians(wfo)
    occrad_m = occrad_rad * dx_m / dx_rad  # occulter radius in meters

    plt.figure(figsize=(12,8))

    if occulter_type == "GAUSSIAN":
        r = proper.prop_radius(wfo)
        h = np.sqrt(-0.5 * occrad_m**2 / np.log(1 - np.sqrt(0.5)))
        gauss_spot = 1 - np.exp(-0.5 * (r/h)**2)
        proper.prop_multiply(wfo, gauss_spot)
        plt.suptitle("Gaussian spot", fontsize = 18)
    elif occulter_type == "SOLID":
        proper.prop_circular_obscuration(wfo, occrad_m)
        plt.suptitle("Solid spot", fontsize = 18)
    elif occulter_type == "8TH_ORDER":
        proper.prop_8th_order_mask(wfo, occrad, CIRCULAR = True)
        plt.suptitle("8th order band limited spot", fontsize = 18)

    # After occulter
    plt.subplot(1,2,1)
    plt.imshow(np.sqrt(proper.prop_get_amplitude(wfo)), origin = "lower", cmap = plt.cm.gray)
    plt.text(200, 10, "After Occulter", color = "w")

    proper.prop_propagate(wfo, f_lens, "pupil reimaging lens")
    proper.prop_lens(wfo, f_lens, "pupil reimaging lens")

    proper.prop_propagate(wfo, 2*f_lens, "lyot stop")

    plt.subplot(1,2,2)
    plt.imshow(proper.prop_get_amplitude(wfo)**0.2, origin = "lower", cmap = plt.cm.gray)
    plt.text(200, 10, "Before Lyot Stop", color = "w")
    plt.show()

    if occulter_type == "GAUSSIAN":
        proper.prop_circular_aperture(wfo, 0.25, NORM = True)
    elif occulter_type == "SOLID":
        proper.prop_circular_aperture(wfo, 0.84, NORM = True)
    elif occulter_type == "8TH_ORDER":
        proper.prop_circular_aperture(wfo, 0.50, NORM = True)

    proper.prop_propagate(wfo, f_lens, "reimaging lens")
    proper.prop_lens(wfo, f_lens, "reimaging lens")

    proper.prop_propagate(wfo, f_lens, "final focus")

    return
```

The `coronagraph` procedure is called from the prescription `run_occulter`:

```
    import proper
    from coronagraph import coronagraph


    def run_occulter(wavelength, grid_size, PASSVALUE = {'occulter_type': 'GAUSSIAN'}):

        diam = 0.1                 # telescope diameter in meters
        f_lens = 24 * diam
        beam_ratio = 0.3

        wfo = proper.prop_begin(diam, wavelength, grid_size, beam_ratio)

        proper.prop_circular_aperture(wfo, diam/2)
        proper.prop_define_entrance(wfo)
```

```
        coronagraph(wfo, f_lens, PASSVALUE["occulter_type"], diam)

        (wfo, sampling) = proper.prop_end(wfo)

        return (wfo, sampling)
```

Matlab:

```
function wavefront = coronagraph(wavefront, f_lens, occulter_type, diam)
  global ifig;            % index of figure

  wavefront  = prop_lens(wavefront, f_lens, 'coronagraph imaging lens');
  wavefront  = prop_propagate(wavefront, f_lens, 'snm', 'occulter');

  % Occulter sizes are specified here in units of lambda / diameter.
  % Convert lambda / diameter to radians, then to meters.

  lambda = prop_get_wavelength(wavefront);          % wavelength (m)
  occrad = 4.0;                                     % occulter radius in lambda / diam
  occrad_rad = occrad * lambda / diam;              % occulter radius (radians)
  dx_m = prop_get_sampling(wavefront);              % pixel spacing (m)
  dx_rad = prop_get_sampling_radians(wavefront);    % pixel spacing (radians)
  occrad_m = occrad_rad * dx_m / dx_rad;            % occulter radius (m)

  switch occulter_type
    case '8TH_ORDER'
      wavefront = prop_8th_order_mask(wavefront, occrad, ...
                  'tmin', 0.0d0, 'tmax', 1.0d0, 'circ');
    case 'GAUSSIAN'
      r          = prop_radius(wavefront);
      h          = sqrt(-0.5d0 * occrad_m^2 / log(1.0d0 - sqrt(0.5d0)));
      gauss_spot = 1.0d0 - exp(-0.5d0 * (r / h).^2);
      wavefront  = prop_multiply(wavefront, gauss_spot);
    case 'SOLID'
      wavefront  = prop_circular_obscuration(wavefront, occrad_m);
  end

  ifig = ifig + 1;
  figure(ifig);
  clf;
  axes('FontSize', 16);
  imagesc((prop_get_amplitude(wavefront)).^0.5);
  axis equal;                   % x-axis units = y-axis units
  axis tight;                   % set axis limits to range of data
  axis xy;                      % set y-axis to increase from bottom
  hc = colorbar('vert');
  set(hc, 'FontSize', 16);
  colormap(gray);
  set(get(gcf, 'CurrentAxes'), 'FontSize', 16);
  tit1 = sprintf('After Occulter');
  title(tit1, 'FontSize', 16);

  wavefront  = prop_propagate(wavefront, f_lens, 'snm', 'pupil reimaging lens');
  wavefront  = prop_lens(wavefront, f_lens, 'pupil reimaging lens');

  wavefront  = prop_propagate(wavefront, f_lens * 2.0, 'snm', 'lyot stop');

  ifig = ifig + 1;
  figure(ifig);
  clf;
  axes('FontSize', 16);
  imagesc((prop_get_amplitude(wavefront)).^0.2);
  axis equal;                   % x-axis units = y-axis units
  axis tight;                   % set axis limits to range of data
  axis xy;                      % set y-axis to increase from bottom
  hc = colorbar('vert');
  set(hc, 'FontSize', 16);
  colormap(gray);
  set(get(gcf, 'CurrentAxes'), 'FontSize', 16);
```

```
  tit2 = sprintf('Before Lyot Stop');
  title(tit2, 'FontSize', 16);

  switch occulter_type
    case '8TH_ORDER'
      wavefront  = prop_circular_aperture(wavefront, 0.5, 'norm');
    case 'GAUSSIAN'
      wavefront  = prop_circular_aperture(wavefront, 0.25, 'norm');
    case 'SOLID'
      wavefront  = prop_circular_aperture(wavefront, 0.84, 'norm');
  end

  wavefront  = prop_propagate(wavefront, f_lens, 'snm', 'reimaging lens');
  wavefront  = prop_lens(wavefront, f_lens, 'reimaging lens');

  wavefront  = prop_propagate(wavefront, f_lens, 'snm', 'final focus');

end
```

The `coronagraph` procedure is called from the prescription `run_occulter`:

```
function [wavefront, sampling] = run_occulter(wavelength, grid_size, optval)
  beam_ratio = 0.3;           % beam diameter fraction
  diam = 0.1;                 % telescope diameter (m)
  f_lens = 24.0 * diam;       % focal length (m)

  wavefront  = prop_begin(diam, wavelength, grid_size, beam_ratio);

  wavefront  = prop_circular_aperture(wavefront, diam / 2.0d);
  wavefront  = prop_define_entrance(wavefront);

  wavefront  = coronagraph(wavefront, f_lens, optval.occulter_type, diam);

  [wavefront, sampling] = prop_end(wavefront);

end
```

In this contrived example, the entrance aperture is 10 cm wide and all of the lenses have a focal ratio of *f*/24. The beam ratio is set to 0.3, which creates an oversampled image at the focus. This allows better sampling of the occulter. The occulter type is defined by setting the *occulter_type* member of a structure that is passed to **PROP_RUN** using the *PASSVALUE* keyword. The type is a string ('SOLID', 'GAUSSIAN', or '8TH_ORDER'). A structure is used because additional parameters will be added in later examples. An example run would be:

IDL:
```
prop_run, 'run_occulter', psf, 0.6, 512, PASSVALUE={occulter_type:'GAUSSIAN'}
```

Python:
```
(psf, sampling) = proper.prop_run( 'run_occulter', 0.6, 512,
                                   PASSVALUE={"occulter_type":"GAUSSIAN"} )
```

Matlab:
```
param.type = 'GAUSSIAN';
[psf, sampling] = prop_run( 'run_occulter', 0.6, 512, 'PASSVALUE', param );
```

which uses a Gaussian occulter and generates an image for λ=0.6 μm with a grid size of 512 x 512.

We can now use these two routines to show how different occulters modify the wavefront as seen in the Lyot plane. The routine below, `occulter_demo`, calls the coronagraph routine using each occulter type. The occulted image and the amplitude in the Lyot plane are displayed for each. This routine is executed by typing "`occulter_demo`" on the command line. The results are shown in Figure 22.

IDL:

```
pro occulter_demo

n = 512              ;-- grid size
lambda = 0.55        ;-- wavelength (microns)

window, 0, XSIZE=n*2, YSIZE=n, TITLE='Solid spot'
prop_run, 'run_occulter', solid, lambda, n, $
      PASSVALUE={occulter_type:'SOLID'}
window, 1, XSIZE=n*2, YSIZE=n, TITLE='Gaussian spot'
prop_run, 'run_occulter', gaussian, lambda, n, $
      PASSVALUE={occulter_type:'GAUSSIAN'}
window, 2, XSIZE=n*2, YSIZE=n, TITLE='8th order band limited spot'
prop_run, 'run_occulter', eighth_order, lambda, n, $
      PASSVALUE={occulter_type:'8TH_ORDER'}
end
```

Python:

```
import proper

def occulter_demo():

    n = 512           # grid size
    lamda = 0.55      # wavelength (microns)

    (solid, sampl_solid) = proper.prop_run('run_occulter', lamda, n,
        PASSVALUE = {"occulter_type": "SOLID"})
    (gaussian, sampl_gauss) = proper.prop_run('run_occulter', lamda, n,
        PASSVALUE = {"occulter_type": "GAUSSIAN"})
    (eighth_order, sampl_eighth_order) = proper.prop_run('run_occulter', lamda, n,
        PASSVALUE = {"occulter_type": "8TH_ORDER"})

    return

if __name__ == '__main__':
    occulter_demo()
```

Matlab:

```
global ifig;         % index of figure
ifig = 0;
n = 512;             % number of pixels
lambda = 0.550;      % wavelength (um)

optval.occulter_type = 'SOLID';            % solid occulter
solid = prop_run('run_occulter', lambda, n, 'passvalue', optval);

optval.occulter_type = 'GAUSSIAN';         % gaussian occulter
gaussian = prop_run('run_occulter', lambda, n, 'passvalue', optval);

optval.occulter_type = '8TH_ORDER';        % prop_8th_order_mask
eighth_order = prop_run('run_occulter', lambda, n, 'passvalue', optval);
```

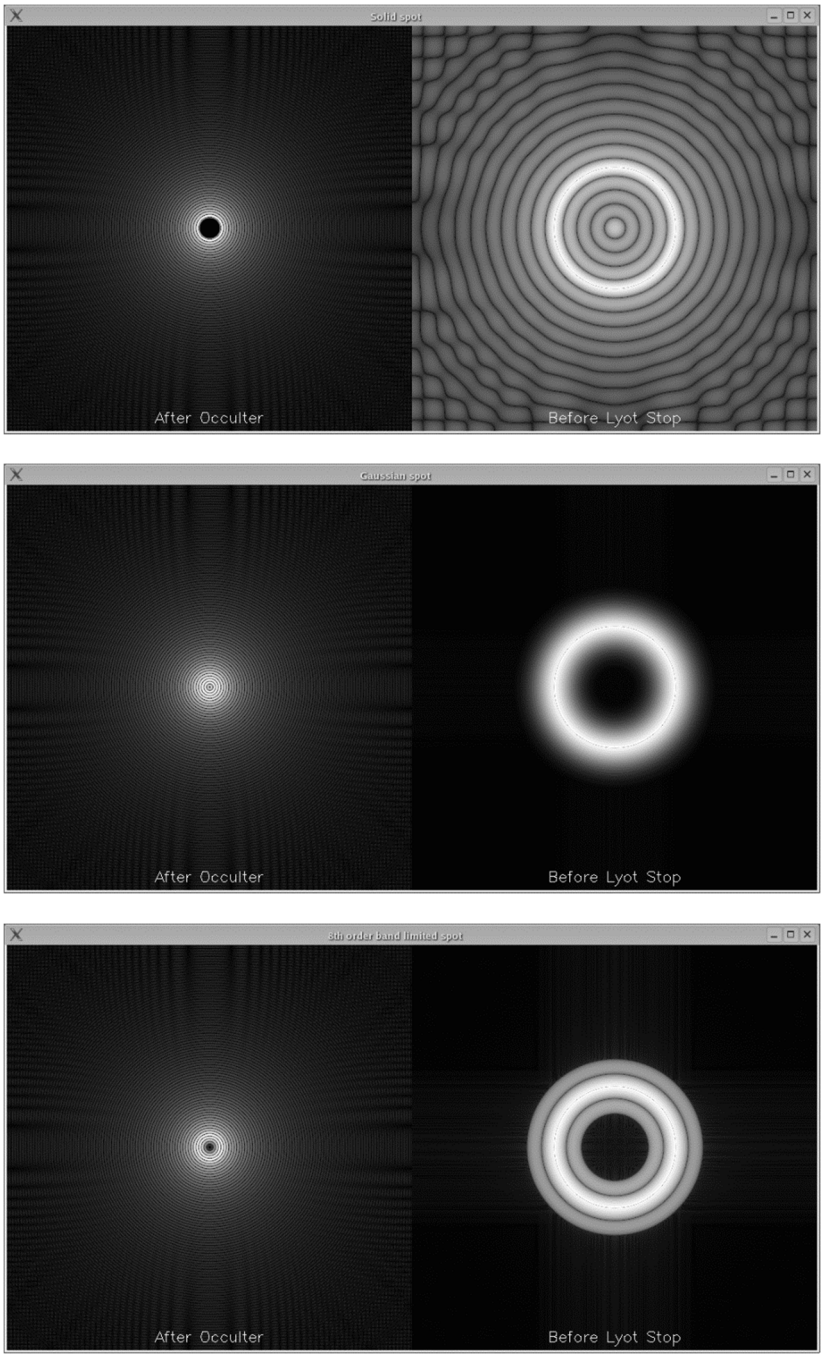


Figure 22. Output from running `occulter_demo`. The solid occulter does a relatively poor job of concentrating light around the edge of the pupil in the Lyot plane, where it could be masked by the Lyot stop. The Gaussian and 8th-order occulters do better. The inner edge of the pupil "doughnut" is quite sharp for the 8th-order, and it allows for a more open (higher throughput) Lyot stop than the Gaussian (which declines less rapidly toward the center). Both the Gaussian and 8th-order occulters provide for greater light suppression than the solid spot. The horizontal and vertical patterns are caused by the limitations of using a finite rectangular grid with Fourier-based propagators (they are at a relatively low level, and the images have been strongly stretched to show such details).

A Simple Coronagraph with a Telescope Having Optical Surface Errors

A real coronagraph would be attached to a telescope that has optical surface errors that are unavoidable during their manufacture. Usually, the bulk of these errors are in the objective lens (or primary mirror), especially mid-spatial-frequency errors that scatter light to large angles from the source. A coronagraph will not suppress this scattered light.

To investigate the effect of these errors, we expand on our example coronagraph, adding a telescope that will function as the front end of the system. Like the coronagraph, the telescope will be a separate procedure (`telescope`) called from the prescription. The telescope objective will be given some mid-spatial-frequency aberrations using **PROP_PSD_ERRORMAP** if the telescope routine's parameter *use_errors* is non-zero. The routine propagates the wavefront from the objective, through focus, and then to a pupil imaging lens, which collimates the beam. The wavefront is then propagated to where a deformable mirror will be inserted later, and then to the entrance of the coronagraph.

IDL:

```
pro telescope, wavefront, fl_lens, use_errors

if ( use_errors ) then begin
  rms_error = 10.0e-9    ;-- RMS wavefront error in meters
  c_freq = 15.0          ;-- correlation frequency (cycles/meter)
  high_power = 3.0       ;-- high frequency falloff (r^-high_power)
  prop_psd_errormap, wavefront, 10.0e-9, c_freq, high_power, $
         /RMS, MAP=obj_map, FILE='telescope_obj.fits'
endif
prop_lens, wavefront, f_lens, 'objective'

;-- propagate through focus to the pupil

prop_propagate, wavefront, fl_lens*2, 'telescope pupil imaging lens'
prop_lens, wavefront, fl_lens, 'telescope pupil imaging lens'

;-- propagate to a deformable mirror (to be inserted later)

prop_propagate, wavefront, fl_lens, 'DM'

prop_propagate, wavefront, fl_lens, 'coronagraph lens'

return
end
```

Python:

```
import proper

def telescope(wfo, f_lens, use_errors, use_dm = False):

    if use_errors:
        rms_error = 10.e-9       # RMS wavefront error in meters
        c_freq = 15.             # correlation frequency (cycles/meter)
        high_power = 3.          # high frewquency falloff (r^-high_power)

        proper.prop_psd_errormap(wfo, rms_error, c_freq, high_power, RMS = True,
            MAP = "obj_map", FILE = "telescope_obj.fits")

    proper.prop_lens(wfo, f_lens, "objective")

    # propagate through focus to pupil
    proper.prop_propagate(wfo, f_lens*2, "telescope pupil imaging lens")

    proper.prop_lens(wfo, f_lens, "telescope pupil imaging lens")

    # propagate to a deformable mirror (to be inserted later)
    proper.prop_propagate(wfo, f_lens, "DM")

    proper.prop_propagate(wfo, f_lens, "coronagraph lens")

    return
```

Matlab:

```
function wavefront = telescope(wavefront, fl_lens, use_errors)

  if use_errors == 1
    rms_error = 10.0d-09;     % RMS wavefront error
    c_freq = 15.0;            % correlation frequency (cycles / m)
    high_power = 3.0;  % high frequency falloff
    flnm = 'telescope_obj.fits';
    [wavefront, obj_map] = prop_psd_errormap(wavefront, rms_error, ...
                             c_freq, high_power, 'file', flnm, 'rms');
  end

  wavefront = prop_lens(wavefront, fl_lens, 'objective');

  % Propagate through focus to the pupil

  wavefront = prop_propagate(wavefront, fl_lens * 2.0, ...
                'snm', 'telescope pupil imaging lens');
  wavefront = prop_lens(wavefront, fl_lens, 'telescope pupil imaging lens');

  % Propagate to a deformable mirror (no actual DM here)

  wavefront = prop_propagate(wavefront, fl_lens, 'snm', 'DM');
  wavefront = prop_propagate(wavefront, fl_lens, 'snm', 'coronagraph lens');

end
```

We can integrate the telescope and the coronagraph by adding the following line to `run_occulter` just before the call to the routine `coronagraph`, naming the new routine `run_coronagraph`:

IDL:
```
telescope, wavefront, fl_lens, optval.use_errors
```

Python:
```
telescope(wfo, fl_lens, PASSVALUE['use_errors'])
```

Matlab:
```
wavefront = telescope(wavefront, fl_lens, optval.use_errors);
```

Now, when **PROP_RUN** is called, the *use_errors* member of the structure assigned to the *PASSVALUE* keyword must also be set, like so:

IDL:
```
prop_run, 'run_coronagraph', psf, 0.55, 512,
     PASSVALUE={occulter_type:'GAUSSIAN',use_errors:1}
```

Python:
```
(psf, dx) = prop_run( 'run_coronagraph', 0.55, 512,
       PASSVALUE={"occulter_type":'GAUSSIAN',"use_errors":1} )
```

Matlab:
```
optval.type = 'GAUSSIAN';
optval.use_errors = 1;
[psf, dx] = prop_run( 'run_coronagraph', 0.55, 512, 'PASSVALUE', optval );
```

Results of this run are shown in Figure 23 and Figure 24. The mid-frequency errors create speckles of scattered light. Over a broad passband, the PSF expands with wavelength and the speckles will smear into streaks.

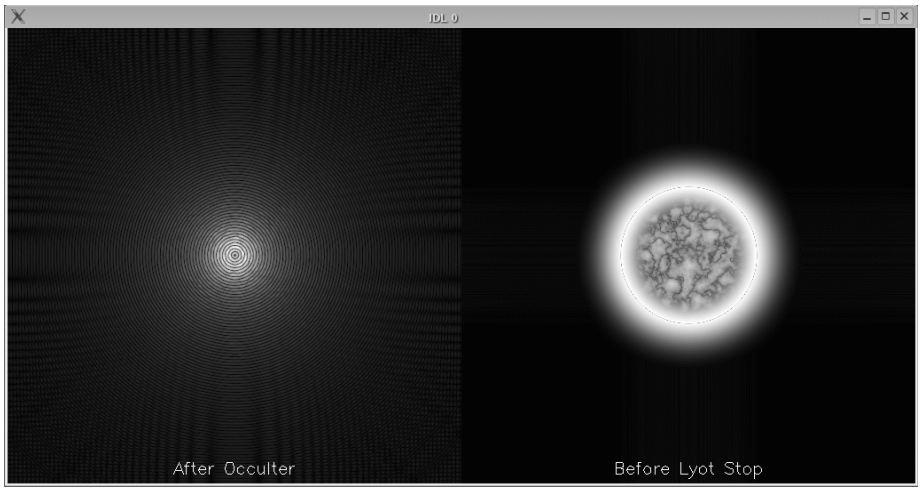


Figure 23. Results displayed by `run_coronagraph` when surface errors in the telescope objective are included. The errors scatter light that cannot be suppressed by the coronagraph. This scattered light can be seen within the center of the Lyot plane image.

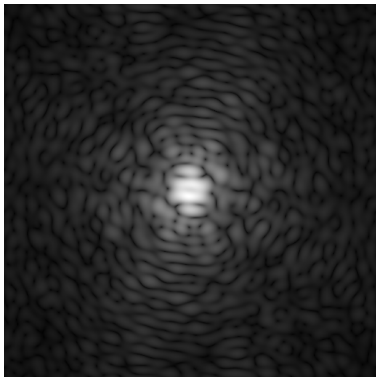


Figure 24. Occulted star image for a system with wavefront errors, corresponding to the case presented in Figure 23. At this stretch ($I^{1/4}$), nothing would be seen if the system had no errors.

A Simple Coronagraph: Wavefront Correction with a Deformable Mirror

We can see from the previous example that the scattered light background created by wavefront errors reduces the effectiveness of the coronagraph. In that example the wavefront had only phase errors introduced at the objective, but it would also be possible to have amplitude errors caused by non-uniform coatings or glass impurities that would also scatter light. Wavefront phase errors can be reduced by placing a deformable mirror (DM) at a pupil prior to the occulter. Just such a place was defined in the `telescope` routine. Amplitude errors can also be reduced by a DM to some degree, but we will not discuss that here. A DM can only correct errors over a limited range of spatial frequencies, up to $N_{act}/2$ cycles over the beam diameter where N_{act} is the number of actuators across the beam.

In the real world, the greatest difficulty with the use of a DM to correct aberrations is measuring those errors. There are a wide variety of techniques to do so, each with its pros and cons. In the following example, however, to avoid the complexity introduced by wavefront sensing, we shall simply cheat and make use of the error map created by **PROP_PSD_ERRORMAP**. That map is in meters of wavefront error. The DM surface is thus set to be half that

(because reflection doubles the path length) and with the opposite sign. The map must also be rotated 180º because the beam went through focus prior to the DM (after rotation using IDL's **ROTATE()** function, the map must be shifted 1 pixel in each direction to return its center in the same place). A 49 by 49 element DM with 47 elements spanning the beam diameter is used. The map must be projected onto this reduced number of samples, and that is done with **PROP_MAGNIFY**. The resized map is then passed to **PROP_DM** with the */FIT* switch set. This tells **PROP_DM** that the map is the requested surface height at each actuator rather than the height of the actuator. The map will be fit to include the effects of the actuator influence function in order to determine the actuator heights necessary to obtain the required surface.

The example code below is inserted in `telescope` just after the propagation to the DM. An additional parameter, *use_dm*, must be added to the end of the parameter definition of `telescope`. The modified routine is now called `telescope_dm`, and a new version of `run_coronagraph`, called `run_coronagraph_dm`, is created that calls it instead of `telescope`. When *use_dm* is not zero, the wavefront will be compensated by the deformable mirror.

IDL:

```
if ( use_dm and use_errors ) then begin
  nact = 49                       ;-- number of DM actuators along one axis
  nact_across_pupil = 47 ;-- number of DM actuators across pupil
  dm_xc = nact / 2                ;-- actuator at wavefront center
  dm_yc = nact / 2
  d_beam = 2 * prop_get_beamradius( wavefront )        ;-- beam diameter
  act_spacing = d_beam / nact_across_pupil     ;-- actuator spacing
  map_spacing = prop_get_sampling( wavefront ) ;-- map sampling

  ;-- have passed through focus, so pupil has rotated 180 deg;
  ;-- need to rotate error map (also need to shift due to the way
  ;-- the rotate() function operates to recenter map)

  obj_map = shift(rotate(obj_map,2),1,1)

  ;-- interpolate map to match number of DM actuators

  dm_map = prop_magnify( obj_map, map_spacing/act_spacing, nact )

  ;-- Need to put on opposite pattern;
  ;-- convert wavefront error to surface height

  prop_dm, wavefront, -dm_map/2, dm_xc, dm_yc, act_spacing, /FIT
endif
```

Python:

```
if use_dm:
    nact = 49                       # number of DM actuators along one axis
    nact_across_pupil = 47          # number of DM actuators across pupil
    dm_xc = nact / 2
    dm_yc = nact / 2
    d_beam = 2 * proper.prop_get_beamradius(wfo)        # beam diameter
    act_spacing = d_beam / nact_across_pupil            # actuator spacing
    map_spacing = proper.prop_get_sampling(wfo)         # map sampling

    # have passed through focus, so pupil has rotated 180 deg;
    # need to rotate error map (also need to shift due to the way
    # the rotate() function operates to recenter map)

    obj_map = np.roll(np.roll(np.rot90(obj_map, 2), 1, 0), 1, 1)

    # interpolate map to match number of DM actuators

    dm_map = proper.prop_magnify(obj_map, map_spacing/act_spacing, nact, QUICK = True)

    # Need to put on opposite pattern; convert wavefront error to surface height

    proper.prop_dm(wfo, -dm_map/2, dm_xc, dm_yc, act_spacing, FIT = True)
```

Matlab:

```
if use_dm == 1
  nact = 49;                  % number of DM actuators along one axis
  nact_across_pupil = 47;     % number of DM actuators across pupil
  dm_xc = fix(nact / 2);      % actuator X index at wavefront center
  dm_yc = fix(nact / 2);      % actuator Y index at wavefront center
  d_beam = 2.0 * prop_get_beamradius(wavefront);    % beam diameter
  act_spacing = d_beam / nact_across_pupil;         % actuator spacing
  map_spacing = prop_get_sampling(wavefront);       % map sampling

  % Have passed through focus, so pupil has rotated 180 degrees;
  % need to rotate error map (also need to shift due to the way
  % the rotate function operates to recenter map)

  obj_map = circshift(rot90(obj_map, 2), [1, 1]);

  % Interpolate map to match number of DM actuators

  dm_map = prop_magnify(obj_map, map_spacing / act_spacing, 'size_out', nact);

  % Need to put on opposite pattern and convert wavefront error to surface height

  wavefront = prop_dm(wavefront, -dm_map / 2.0, dm_xc, dm_yc, act_spacing, 'fit');
end
```

The procedure `coronagraph_demo`, listed below, demonstrates this new functionality to our example coronagraph.

IDL:

To run it, simply type “coronagraph_demo” on the command line.

```
pro coronagraph_demo

n = 512          ;-- grid size
lambda = 0.55    ;-- wavelength (microns)

window, 0, xsize=n*2, ysize=n, title='no errors'
prop_run, 'run_coronagraph_dm', no_errors, lambda, n, $
    PASSVALUE={use_errors:0,use_dm:0,occulter_type:'8TH_ORDER'}

window, 1, xsize=n*2, ysize=n, title='with errors, no DM'
prop_run, 'run_coronagraph_dm', with_errors, lambda, n, $
    PASSVALUE={use_errors:1,use_dm:0,occulter_type:'8TH_ORDER'}

window, 2, xsize=n*2, ysize=n, title='with errors, DM correction'
prop_run, 'run_coronagraph_dm', with_dm, lambda, n, $
    PASSVALUE={use_errors:1,use_dm:1,occulter_type:'8TH_ORDER'}

np = 256
psfs = fltarr(np*3,np)
psfs(0,0) = no_errors(n/2-np/2:n/2-1,n/2-np/2:n/2-1)
psfs(np,0) = with_errors(n/2-np/2:n/2-1,n/2-np/2:n/2-1)
psfs(np*2,0) = with_dm(n/2-np/2:n/2-1,n/2-np/2:n/2-1)

window, 3, xsize=np*3, ysize=np, title='PSFs'
tvscl, psfs^0.25
xyouts, 0.5*np, 10, 'No errors', /dev, align=0.5, size=2
xyouts, 1.5*np, 10, 'With errors', /dev, align=0.5, size=2
xyouts, 2.5*np, 10, 'DM corrected', /dev, align=0.5, size=2

print, 'Maximum speckle flux / stellar flux :'
print, '   No wavefront errors   = ', max(no_errors)
print, '   With wavefront errors = ', max(with_errors)
print, '   With DM correction    = ', max(with_dm)

end
```

Python:

To run it, simply type "coronagraph_demo" on the command line.

```
import proper
import numpy as np
import matplotlib.pylab as plt

def coronagraph_demo():

    n = 256             # grid size
    lamda = 0.55        # wavelength (microns)

    no_errors, no_errors_sampl = proper.prop_run("run_coronagraph_dm", lamda, n,
           PASSVALUE = {'use_errors': False, 'use_dm': False, 'occulter_type': '8TH_ORDER'},
           VERBOSE = False)

    with_errors, with_errors_sampl = proper.prop_run("run_coronagraph_dm", lamda, n,
           PASSVALUE = {'use_errors': True, 'use_dm': False, 'occulter_type': '8TH_ORDER'},
           VERBOSE = False)

    with_dm, with_dm_sampl = proper.prop_run("run_coronagraph_dm", lamda, n,
           PASSVALUE = {'use_errors': True, 'use_dm': True, 'occulter_type': '8TH_ORDER'},
           VERBOSE = False)

    nd = 256
    psfs = np.zeros([3,nd,nd], dtype = np.float64)
    psfs[0,:,:] = no_errors[n/2-nd/2:n/2+nd/2,n/2-nd/2:n/2+nd/2]
    psfs[1,:,:] = with_errors[n/2-nd/2:n/2+nd/2,n/2-nd/2:n/2+nd/2]
    psfs[2,:,:] = with_dm[n/2-nd/2:n/2+nd/2,n/2-nd/2:n/2+nd/2]

    plt.figure(figsize = (14,7))
    plt.suptitle("PSFs", fontsize = 18, fontweight = 'bold')

    plt.subplot(1,3,1)
    plt.imshow(psfs[0,:,:]**0.25, origin = "lower", cmap = plt.cm.gray)
    plt.subplot(1,3,2)
    plt.imshow(psfs[1,:,:]**0.25, origin = "lower", cmap = plt.cm.gray)
    plt.subplot(1,3,3)
    plt.imshow(psfs[2,:,:]**0.25, origin = "lower", cmap = plt.cm.gray)
    plt.show()

    print "Maximum speckle flux / stellar flux :"
    print "  No wavefront errors = ", np.max(no_errors), np.min(no_errors)
    print "  With wavefront errors = ", np.max(with_errors)
    print "  With DM correction = ", np.max(with_dm)

if __name__ == '__main__':
    coronagraph_demo()
```

Matlab:

Here is `coronagraph_demo`:

```
global ifig;                        % figure number
ifig = 0;                           % figure number

n = 512;                            % grid size
nps = 256;                          % number of pixels in psf sample
nps2 = fix(nps / 2.0);
icx = fix(n   / 2.0) + 1;           % index of center of array X
icy = fix(n   / 2.0) + 1;           % index of center of array Y
ix1 = icx - nps2;           % psf sample min index X
ix2 = ix1 + nps - 1;        % psf sample max index X
iy1 = icy - nps2;           % psf sample min index Y
iy2 = iy1 + nps - 1;        % psf sample max index Y
```

```
lambda = 0.55;                      % wavelength (um)

figarray(4, 4);                     % set up positions of figures in array

optval.use_dm = 0;                     % deformable mirror: no
optval.use_errors = 0;                 % prop_psd_errormap: no
optval.occulter_type = '8TH_ORDER';    % prop_8th_order_mask: yes
psf1 = prop_run('run_coronagraph_dm', lambda, n, 'prm', optval);
psfs = psf1(iy1:iy2, ix1:ix2);

% Plot Point Spread Function intensity for case with no errors
ifig = ifig + 1;
figplace(ifig);
clf;
axes('FontSize', 16);
imagesc(psfs.^0.25);
axis equal;                         % x-axis units = y-axis units
axis tight;                         % set axis limits to range of data
axis xy;                            % set y-axis to increase from bottom
hc = colorbar('vert');
set(hc, 'FontSize', 16);
caxis([0.0, 0.03]);
colormap(gray);
set(get(gcf, 'CurrentAxes'), 'FontSize', 16);
tit1 = sprintf('PSF: no errors');
title(tit1, 'FontSize', 16);

optval.use_dm = 0;                     % deformable mirror: no
optval.use_errors = 1;                 % prop_psd_errormap: yes
optval.occulter_type = '8TH_ORDER';    % prop_8th_order_mask: yes
psf2 = prop_run('run_coronagraph_dm', lambda, n, 'prm', optval);
psfs = psf2(iy1:iy2, ix1:ix2);

% Plot Point Spread Function intensity for case with errors, no DM
ifig = ifig + 1;
figplace(ifig);
clf;
axes('FontSize', 16);
imagesc(psfs.^0.25);
axis equal;                   % x-axis units = y-axis units
axis tight;                   % set axis limits to range of data
axis xy;                      % set y-axis to increase from bottom
hc = colorbar('vert');
set(hc, 'FontSize', 16);
caxis([0.0, 0.03]);
colormap(gray);
set(get(gcf, 'CurrentAxes'), 'FontSize', 16);
tit2 = sprintf('PSF: with errors');
title(tit2, 'FontSize', 16);

optval.use_dm = 1;                     % deformable mirror: yes
optval.use_errors = 1;                 % prop_psd_errormap: yes
optval.occulter_type = '8TH_ORDER';    % prop_8th_order_mask: yes
psf3 = prop_run('run_coronagraph_dm', lambda, n, 'prm', optval);
psfs = psf3(iy1:iy2, ix1:ix2);

% Plot Point Spread Function intensity for case with errors, DM correction
ifig = ifig + 1;
figplace(ifig);
clf;
axes('FontSize', 16);
imagesc(psfs.^0.25);
axis equal;                   % x-axis units = y-axis units
axis tight;                   % set axis limits to range of data
axis xy;                      % set y-axis to increase from bottom
hc = colorbar('vert');
set(hc, 'FontSize', 16);
caxis([0.0, 0.03]);
colormap(gray);
set(get(gcf, 'CurrentAxes'), 'FontSize', 16);
tit3 = sprintf('PSF: DM corrected');
```

```
title(tit3, 'FontSize', 16);

fprintf(1, 'Maximum speckle flux / stellar flux :\n');
fprintf(1, '   No wavefront errors   = %16.7e\n', max(max(psf1)));
fprintf(1, '   With wavefront errors = %16.7e\n', max(max(psf2)));
fprintf(1, '   With DM correction    = %16.7e\n', max(max(psf3)));
```

The first call to `run_coronagraph_dm` simulates an unaberrated system, producing results like those shown in the bottom panel of Figure 22. The second includes wavefront aberrations, and the third uses a DM to correct them. The results of the last two are shown in Figure 25 and Figure 26. The DM reduces the brightest speckles by about a factor of 300.

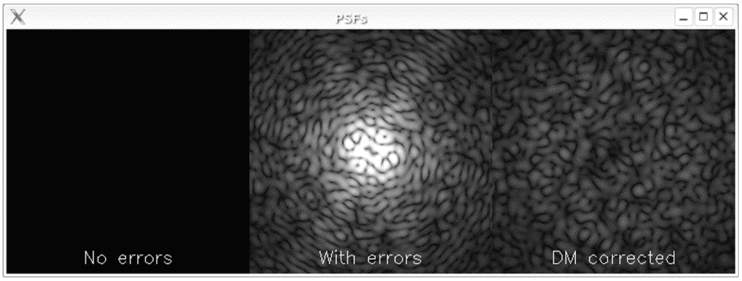


Figure 25. Displayed results from coronagraph_demo. These images show the intensity at the final focal plane of the coronagraph (identically stretched in intensity to show low-level detail). On the left is the occulted source in a system without aberrations, the middle has aberrations, and on the right is after wavefront error correction by a deformable mirror.

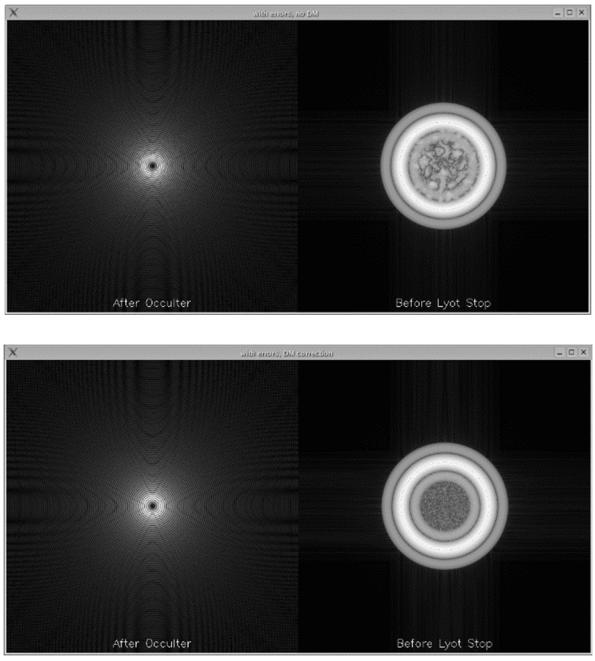


Figure 26. Results of running coronagraph_demo. In each panel, the left half shows the wavefront amplitude at the first image plane (after occulter) and on the right in the Lyot plane (before Lyot stop), stretched to show low-intensity details. The top panel shows the effects of mid-spatial-frequency aberrations, and the bottom shows the same case after correction by a deformable mirror. The residuals have higher spatial frequencies than can be corrected by the DM. An 8^{th}-order band-limited occulter with 50% intensity transmission at 4λ/D was used.

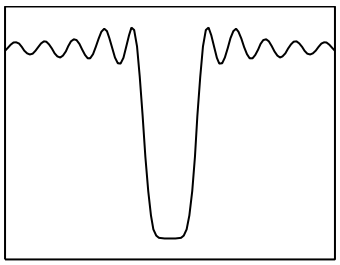
PROPER Routine Reference Manual

PROP_8TH_ORDER_MASK

Multiply the current wavefront by an 8th-order transmission profile representing the occulter in a coronagraph. These masks are described by Kuchner et al. in *The Astrophysical Journal*, 628, 466 (2005). The transmission (square of the amplitude) is defined to be

$$T(x) = a \cdot \left[\frac{l-m}{l} - \frac{\sin(\pi x \varepsilon / l)^l}{(\pi x \varepsilon / l)^l} + \frac{m}{l} \frac{\sin(\pi x \varepsilon / m)^m}{(\pi x \varepsilon / m)^m} \right]^2$$

where x is the radius from the center of the occulter, $\varepsilon = 1.788/w$ (w is half-width where the transmission is 50%), and l and m describe the form of the transmission function (l=3, m=1). The constant a is set to define the required transmission range. By default, a linear mask is created (mask transmission varies along the X axis with constant values along the Y axis, or the opposite if the */Y_AXIS* switch is set). Circular and elliptical masks can also be created.



Intensity transmission profile of an 8th-order mask.

Syntax

IDL: **prop_8th_order_mask**, *wavestruct, hwhm [, /CIRCULAR] [, ELLIPTICAL=ratio] [, MASK=maskarray] [, MAX_TRANSMISSION=maxvalue] [, /METERS] [, MIN_TRANSMISSION=minvalue] [, /Y_AXIS]*

Python: *[mask =]* **proper.prop_8th_order_mask(** *wavestruct, hwhm [, CIRCULAR=True/False] [, ELLIPTICAL=ratio] [, MAX_TRANSMISSION=maxvalue] [, METERS=True/False] [, MIN_TRANSMISSION=minvalue] [, Y_AXIS=True/False]*)

Matlab: *wavestruct_out =* - OR -
[wavestruct_out, mask] =
prop_8th_order_mask(*wavestruct_in, hwhm [, 'CIRCULAR'] [, 'ELLIPTICAL', ratio] [, 'MAX_TRANSMISSION', maxvalue] [, 'METERS'] [, 'MIN_TRANSMISSION', minvalue] [, 'Y_AXIS']*);

Returns

wavestruct_out (Matlab)
(Required) The modified wavefront structure.

mask (Python)
mask (Matlab)
(Optional) A variable that will contain the mask amplitude pattern created by this routine.

Arguments

wavestruct (IDL)
wavestruct (Python)
wavestruct_in (Matlab)
(Required) The current wavefront structure.

hwhm
(Required) The radius from the mask center at which the intensity transmission is 50% of its maximum value. By default, this is assumed to be in units of λ/*D* radians (*D* = entrance pupil diameter), but can be in meters if the *METERS* switch is given.

Keywords and Switches

/CIRCULAR (IDL)
CIRCULAR=*True* or *False* (Python)
'CIRCULAR' (Matlab)
(Optional) Switch that indicates that a circularly symmetric mask should be generated (the default is a linear mask).

ELLIPTICAL=*ratio* (IDL)
ELLIPTICAL=*ratio* (Python)
'ELLIPTICAL', *ratio* (Matlab)
(Optional) Indicates that an elliptical mask should be created with an aspect ratio of *ratio* = *x_width* / *y_width*. The ellipse axes are aligned along the wavefront X and Y axes.

MASK=*maskarray* (IDL)
(Optional) In IDL, a variable that will contain the mask amplitude pattern created by this routine.

MAX_TRANSMISSION=*maxvalue* (IDL)
MAX_TRANSMISSION=*maxvalue* (Python)
'MAX_TRANSMISSION', *maxvalue* (Matlab)
(Optional) Specifies the maximum value of the mask transmission. The default is 1.0.

/METERS (IDL)
METERS=*True* or *False* (Python)
'METERS' (Matlab)
(Optional) Switch that indicates that the value of *hwhm* is in meters rather than units of λ/*D*.

MIN_TRANSMISSION=*minvalue* (IDL)
MIN_TRANSMISSION=*minvalue* (Python)
'MIN_TRANSMISSION', *minvalue* (Matlab)
(Optional) Specifies the minimum value of the mask transmission. The default is 0.0.

/Y_AXIS (IDL)
Y_AXIS=*True* or *False* (Python)
'Y_AXIS' (Matlab)
(Optional) Switch that specifies that the transmission of a linear occulter should vary along the Y axis.

Examples

Multiply the current wavefront by a mask with a 4 λ/D HWHM (*D* = diameter of entrance aperture):

IDL:
```
prop_8th_order_mask, wavefront, 4.0
```

Python:
```
proper.prop_8th_order_mask( wavefront, 4.0 )
```

Matlab:
```
wavefront = prop_8th_order_mask( wavefront, 4.0 );
```

Multiply the current wavefront by a circular mask with a 3λ/D HWHM and return the mask amplitude in the variable *maskarray*:

IDL:
```
prop_8th_order_mask, wavefront, 3.0, MASK=maskarray, /CIRCULAR
```

Python:
```
maskarray = proper.prop_8th_order_mask( wavefront, 3.0, CIRCULAR=True )
```

Matlab:
```
[wavefront, maskarray] = prop_8th_order_mask( wavefront, 3.0, 'CIRCULAR' );
```

PROP_ADD_PHASE

Add an error map or value to the current wavefront phase component. If an array, it is assumed to be at the same sampling as the wavefront. Note that this is wavefront, not surface, error.

Syntax

IDL: **prop_add_phase**, *wavestruct, phase_error*

Python: **proper.prop_add_phase**(*wavestruct, phase_error*)

Matlab: *wavestruct_out* = **prop_add_phase(** *wavestruct_in, phase_error*);

Returns

wavestruct_out (Matlab)
(Required) The modified wavefront structure.

Arguments

wavestruct (IDL)
wavestruct (Python)
wavestruct_in (Matlab)
(Required) The current wavefront structure.

phase_error
(Required) Two dimensional array containing the wavefront error in meters. The spacing must be the same as that in the current wavefront, which can be obtained using **PROP_GET_SAMPLING**.

Examples

Add 0.5 μm RMS of defocus to the current wavefront, given that the Zernike polynomial for defocus for an unobscured circular aperture is $\sqrt{3}(2r^2 - 1)$ where r = 1.0 at the pupil radius (equivalent to using **PROP_ZERNIKES**):

IDL:
```
rho = prop_radius( wavefront )
focus = 0.5e-6 * sqrt(3) * (2*rho^2 - 1)
prop_add_phase, wavefront, focus
```

Python:
```
rho = proper.prop_radius( wavefront )
focus = 0.5e-6 * np.sqrt(3) * (2*rho**2 - 1)
proper.prop_add_phase( wavefront, focus )
```

Matlab:
```
rho = prop_radius( wavefront );
focus = 0.5e-6 .* sqrt(3) .* (2 .* rho.^2 - 1);
wavefront = prop_add_phase( wavefront, focus );
```

See Also

PROP_ERRORMAP, PROP_PSD_ERRORMAP, PROP_ZERNIKES

PROP_BEGIN

Initialize a PROPER prescription. This routine must be called before calling other PROPER routines.

Syntax

IDL: **prop_begin**, *wavestruct, beam_diam, wavelength, grid_n, [, beam_diam_fraction]*

Python: *wavestruct* = **proper.prop_begin**(*beam_diam, wavelength, grid_n [, beam_diam_fraction]*)

Matlab: *wavestruct* = **prop_begin**(*beam_diam, wavelength, grid_n [, beam_diam_fraction]);*

Returns

wavestruct (Python)
wavestruct (Matlab)
(Required) The new, initialized wavefront structure .

Arguments

wavestruct (IDL)
(Required) A variable to contain the new, initialized wavefront structure.

beam_diam
(Required) The initial diameter of the beam in meters.

wavelength
(Required) The wavelength of the simulation in meters.

grid_n
(Required) The dimensions of the simulation grid (grid_n by grid_n pixels). For maximum efficiency, this must be a power of two. Using too small of a grid may lead to significant Fourier transform artifacts (wrap-around, etc.) caused by undersampling and aliasing.

beam_diam_fraction
(Optional) Specifies the ratio of the beam diameter to the grid diameter (default is 0.5). This affects the sampling of the propagation. For instance, in a telescope prescription, a ratio of 0.5 (the beam occupies half of the grid width) at the entrance aperture results in a Nyquist-sampled image at the focus. A smaller ratio results in finer sampling but with the possibility of undersampling features in the pupil and FFT wrap-around errors.

Examples

Initialize a simulation for a 1.0 meter diameter beam, a wavelength of 0.5 microns, a computation grid size of 1024 by 1024 elements, and a beam diameter ratio of 0.4 (beam diameter = 0.4 × 1024 = 409.6 pixels):

IDL: `prop_begin, wavefront, 1.0, 0.5e-6, 1024, 0.4`

Python: `wavefront = proper.prop_begin( 1.0, 0.5e-6, 1024, 0.4 )`

Matlab: `wavefront = prop_begin(1.0, 0.5e-6, 1024, 0.4);`

See Also

PROP_END

PROP_CIRCULAR_APERTURE

Multiply the current wavefront by a circular aperture (clear inside, dark outside). The edge of the circle is antialiased (the value of an edge pixel varies between 0.0 and 1.0 in proportion to the amount of a pixel covered by the circle; the accuracy of antialiasing is set using **PROP_SET_ANTIALIASING**). To create an array of circular apertures (a Shack-Hartmann mask, for instance), the user should create a single mask image using multiple calls to **PROP_ELLIPSE** and then use **PROP_MULTIPLY** to multiply the wavefront by that mask.

Syntax

IDL: **prop_circular_aperture**, *wavestruct, radius* [, *xoff, yoff*] [, */NORM*]

Python: **proper.prop_circular_aperture**(*wavestruct, radius* [, *xoff, yoff*] [, *NORM=True/False*])

Matlab: *wavestruct_out* = **prop_circular_aperture**(*wavestruct_in, radius [, 'XC', xoff] [, 'YC', yoff] [, 'NORM']*);

Returns

wavestruct_out (Matlab)
(Required) The modified wavefront structure.

Arguments

wavestruct (IDL)
wavestruct (Python)
wavestruct_in (Matlab)
(Required) The current wavefront structure.

radius
(Required) Radius of aperture in meters, or if the *NORM* switch is set, the radius in terms of the fraction of the beam radius at the current surface.

xoff, yoff (IDL)
xoff, yoff (Python)
(Optional) X and Y axis offsets of the circle center from the center of the wavefront grid. These are specified in meters unless the *NORM* switch is set, in which case they are in fractions of the current beam radius. By default, the circle is centered at the center of the wavefront grid.

Keywords and Switches

/NORM (IDL)
NORM=*True* or *False* (Python)
'NORM' (Matlab)
(Optional) Switch that indicates that radius and, if provided, *xoff* and *yoff*, are in fractions of the current beam radius (for an unaberrated beam) rather than in meters.

'XC', xoff
'YC', yoff (Matlab)

(Optional) X and Y axis offsets of the circle center from the center of the wavefront grid. These are specified in meters unless the *NORM* switch is set, in which case they are in fractions of the current beam radius. By default, the circle is centered at the center of the wavefront grid.

Examples

Multiply the wavefront by a circular entrance aperture with a central obscuration that is 33% of the diameter of the beam radius:

IDL:
```
prop_circular_aperture, wavefront, 1.0, /NORM
prop_circular_obscuration, wavefront, 0.33, /NORM
```

Python:
```
proper.prop_circular_aperture( wavefront, 1.0, NORM=True )
proper.prop_circular_obscuration( wavefront, 0.33, NORM=True )
```

Matlab:
```
wavefront = prop_circular_aperture( wavefront, 1.0, 'NORM' );
wavefront = prop_circular_obscuration( wavefront, 0.33, 'NORM' );
```

Multiply the wavefront by a filled circle of 10.0 mm radius and centered +1 mm from the wavefront center along the X axis:

IDL:
```
prop_circular_aperture, wavefront, 0.010, 0.001, 0.0
```

Python:
```
proper.prop_circular_aperture( wavefront, 0.010, 0.001, 0.0 )
```

Matlab:
```
wavefront = prop_circular_aperture( wavefront, 0.010, 'XC', 0.001, ...
                    'YC', 0.0 );
```

See Also

PROP_CIRCULAR_OBSCURATION, PROP_ELLIPSE, PROP_ELLIPTICAL_APERTURE, PROP_ELLIPTICAL_OBSCURATION, PROP_RECTANGLE, PROP_RECTANGULAR_APERTURE, PROP_RECTANGULAR_OBSCURATION, PROP_SET_ANTIALIASING

PROP_CIRCULAR_OBSCURATION

Multiply the current wavefront by a circular obscuration (dark inside, clear outside). The edge of the circle is antialiased (the value of an edge pixel varies between 0.0 and 1.0 in proportion to the amount of a pixel covered by the circle; the accuracy of antialiasing is set using **PROP_SET_ANTIALIASING**).

Syntax

IDL: **prop_circular_obscuration**, *wavestruct, radius* [, *xoff, yoff*] [, */NORM*]

Python: **proper.prop_circular_obscuration**(*wavestruct, radius* [, *xoff, yoff*] [, *NORM=True/False]*)

Matlab: *wavestruct_out* = **prop_circular_obscuration**(*wavestruct_in, radius [, 'XC', xoff] [, 'YC', yoff] [, 'NORM']*);

Returns

wavestruct_out (Matlab)
(Required) The modified wavefront structure.

Arguments

wavestruct (IDL)
wavestruct (Python)
wavestruct_in (Matlab)
(Required) The current wavefront structure.

radius
(Required) Radius of aperture in meters, or if the *NORM* switch is set, the radius in terms of the fraction of the beam radius at the current surface.

xoff, yoff (IDL)
xoff, yoff (Python)
(Optional) X and Y axis offsets of the circle center from the center of the wavefront grid. These are specified in meters unless the *NORM* switch is set, in which case they are in fractions of the current beam radius. By default, the circle is centered at the center of the wavefront grid.

Keywords and Switches

/NORM (IDL)
NORM=*True* or *False* (Python)
'NORM' (Matlab)
(Optional) Switch that indicates that radius and, if provided, *xoff* and *yoff*, are in fractions of the current beam radius (for an unaberrated beam) rather than in meters.

'XC', xoff
'YC', yoff (Matlab)

(Optional) X and Y axis offsets of the circle center from the center of the wavefront grid. These are specified in meters unless the *NORM* switch is set, in which case they are in fractions of the current beam radius. By default, the circle is centered at the center of the wavefront grid.

Examples

Multiply the wavefront by an circular entrance aperture with a 33% central circular obscuration:

IDL:
```
prop_circular_aperture, wavefront, 1.0, /NORM
prop_circular_obscuration, wavefront, 0.33, /NORM
```

Python:
```
proper.prop_circular_aperture( wavefront, 1.0, NORM=True )
proper.prop_circular_obscuration( wavefront, 0.33, NORM=True )
```

Matlab:
```
wavefront = prop_circular_aperture( wavefront, 1.0, 'NORM' );
wavefront = prop_circular_obscuration( wavefront, 0.33, 'NORM' );
```

Multiply the wavefront by a filled dark circle of 1.0 mm radius and centered +10 mm from the wavefront center along the X axis:

IDL:
```
prop_circular_obscuration, wavefront, 0.001, 0.01, 0.0
```

Python:
```
proper.prop_circular_obscuration( wavefront, 0.001, 0.01, 0.0 )
```

Matlab:
```
wavefront = prop_circular_obscuration( wavefront, 0.001, 'XC',0.01, ...
                    'YC', 0.0 );
```

See Also

PROP_CIRCULAR_APERTURE, PROP_ELLIPTICAL_APERTURE, PROP_ELLIPTICAL_OBSCURATION, PROP_RECTANGLE, PROP_RECTANGULAR_APERTURE, PROP_RECTANGULAR_OBSCURATION, PROP_SET_ANTIALIASING

PROP_COMPILE_FFTI (IDL, Python)

Compile the C routines that provide the interface to the Intel Math Kernel Library FFT routine.

Syntax

prop_compile_ffti

Arguments

This routine has no arguments

Notes

After the interface has been successfully compiled, PROPER needs to be set to use them with the **PROP_USE_FFTI** routine.

PROP_COMPILE_FFTW (IDL, Python)

Compile the C routines that provide the interface to the FFTW library for improved FFT speed.

Syntax

prop_compile_fftw

Arguments

This routine has no arguments

Notes

After the interface has been successfully compiled, PROPER needs to be set to use them with the **PROP_USE_FFTW** routine.

PROP_DEFINE_ENTRANCE

This routine normalizes the current wavefront array to have a total intensity of 1.0. This routine should be called after the entrance aperture is drawn.

Syntax

IDL: **prop_define_entrance**, *wavestruct*

Python: **proper.prop_define_entrance**(*wavestruct*)

Matlab: *wavestruct_out* = **prop_define_entrance**(*wavestruct_in*);

Returns

wavestruct_out (Matlab)
(Required) The modified wavefront structure.

Arguments

wavestruct (IDL)
wavestruct (Python)
wavestruct_in (Matlab)
(Required) The current wavefront structure.

PROP_DIVIDE

Divide the current wavefront by a user-specified value or 2D array

Syntax

IDL: **prop_divide**, *wavestruct, value*

Python: **proper.prop_divide**(*wavestruct, value*)

Matlab: *wavestruct_out* = **prop_divide**(*wavestruct_in, value*);

Returns

wavestruct_out (Matlab)
(Required) The modified wavefront structure.

Arguments

wavestruct (IDL)
wavestruct (Python)
wavestruct_in (Matlab)
(Required) The current wavefront structure.

value
(Required) The scalar value or two-dimensional array containing the values by which the current wavefront will be divided. The map in the array is assumed to be centered within the array, and it must have the same dimensions as the current wavefront gridsize (see PROP_GET_GRIDSIZE).

Examples

Reduce the wavefront amplitude by a factor of 2:

```
IDL:    prop_divide, wavefront, 2
Python: proper.prop_divide( wavefront, 2 )
Matlab: wavefront = prop_divide( wavefront, 2 );
```

Replace the current wavefront amplitude with a different one:

```
IDL:    prop_divide, wavefront, prop_get_amp(wavefront)
        prop_multiply, wavefront, new_amp

Python: proper.prop_divide( wavefront, proper.prop_get_amp(wavefront) )
        proper.prop_multiply( wavefront, new_amp )

Matlab: wavefront = prop_divide( wavefront, prop_get_amp(wavefront) );
        wavefront = prop_multiply( wavefront, new_amp );
```

See Also

PROP_ADD_PHASE, PROP_MULTIPLY

PROP_DM

NOTE: For historical reasons, a negative piston provided to PROP_DM results in a positive displacement (away from the DM) of the facesheet for a real DM.

Modify the phase of the current wavefront using a deformable mirror (DM). The DM considered here is an array of actuators uniformly distributed on a rectangular grid and covered with a thin-plate mirror. The user specifies either the height of each actuator or the height of the surface at each actuator (in which case the required actuator height is solved for accounting for the influence function). The deflection of the mirror face sheet on sub-actuator scales includes an influence function.

Syntax

IDL: **prop_dm**, *wavestruct, dm_height, dm_xc, dm_yc, spacing*
[, /FIT] [, INFLUENCE_FUNCTION_FILE=filename] [, MAP=map]
[, /NO_APPLY] [, N_ACT_ACROSS_PUPIL=nact],
[, XTILT=xtilt] [, YTILT=ytilt] [, ZTILT=ztilt][, /XYZ] [, /ZYX]

Python: *[map =]* **proper.prop_dm(** *wavestruct, dm_height, dm_xc, dm_yc, spacing*
[, FIT=True/False] [, INFLUENCE_FUNCTION_FILE=filename]
[, NO_APPLY=True/False] [, N_ACT_ACROSS_PUPIL=nact]
[, XTILT=xtilt] [, YTILT=ytilt] [, ZTILT=ztilt][, XYZ=True/False] [, ZYX=True/False])

Matlab: *wavestruct_out =* - OR -
[wavestruct_out, map] =
prop_dm(*wavestruct_in, dm_height, dm_xc, dm_yc, spacing*
[, 'FIT'] [, 'INFLUENCE_FUNCTION_FILE', filename]
[, 'NO_APPLY'] [, 'N_ACT_ACROSS_PUPIL', nact]
[, 'XTILT', xtilt] [, 'YTILT', ytilt] [, 'ZTILT', ztilt] [, 'XYZ'] [,' ZYX']);

Returns

wavestruct_out (Matlab)
(Required) The modified wavefront structure.

map (Python)
map (Matlab)
(Optional) The *surface* height (*not wavefront*) map corresponding to the input DM strokes. The map is in meters and is at the sampling of the current wavefront.

Arguments

wavestruct (IDL)
wavestruct (Python)
wavestruct_in (Matlab)
(Required) The current wavefront structure.

dm_height

NOTE: For historical reasons, a negative piston provided to PROP_DM results in a positive displacement (away from the DM) of the facesheet for a real DM.

(Required) Either of the following:

1) A two-dimensional array (nx,ny) in which each element represents the height in meters of the corresponding actuator or required surface height at that actuator.
2) The filename of a FITS two-dimensional image file containing the heights.

The first element in the array is assumed to be in the lower left corner in the wavefront coordinate system. If the *FIT* switch is set, then the heights in the array represent the desired height of the DM surface at that position including effects of the actuator influence functions.

dm_xc, dm_yc

(Required) X, Y actuator coordinates of the optical axis. The center of the first actuator is (0.0,0.0) and of the last is (nx-1,ny-1).

spacing

(Required if *N_ACT_ACROSS_PUPIL* not defined) The distance in meters between DM actuators. This parameter is (IDL: omitted, Python: omitted, Matlab: ignored) if the *N_ACT_ACROSS_PUPIL* parameter is defined.

Keywords and Switches

/FIT (IDL)
FIT=*True* or *False* (Python)
'FIT' (Matlab)

(Optional) Switch indicating that the values in *dm_height* are the desired surface heights, rather than commanded heights. The heights will be fit by an iterative algorithm that solves for actuator heights required to produce the desired surface heights, including the effects of the actuator influence function.

INFLUENCE_FUNCTION_FILE = *filename* (IDL)
INFLUENCE_FUNCTION_FILE = *filename* (Python)
'INFLUENCE_FUNCTION_FILE', *filename* (Matlab)

(Optional) Sting giving the name of a FITS file containing a 2D map of the actuator influence function. The function's form and file header keywords are specified in *Defining the Influence Function* in the *Deformable Mirror* section earlier in this manual. When not specified, a Xinetics measured function is used.

MAP=*maparray* (IDL)

(Optional) A variable to containt the surface height (not wavefront) map corresponding to the input DM strokes. The map is in meters and is at the sampling of the current wavefront.

/NO_APPLY (IDL)
NO_APPLY=*True* or *False* (Python)
'NO_APPLY' (Matlab)

(Optional) If set, a DM surface map will be generated and returned in *map/MAP* but not applied to the wavefront.

N_ACT_ACROSS_PUPIL = *nact* (IDL)
N_ACT_ACROSS_PUPIL = *nact* (Python)
'N_ACT_ACROSS_PUPIL', *nact* (Matlab)

(Optional) The number of actuators that span the beam diameter (which is measured along the X axis). This will determine the actuator spacing, so any value specified by the *spacing* parameter will be ignored.

XTILT = *xtilt*, YTILT = *ytilt*, ZTILT = *ztilt* (IDL)
XTILT = *xtilt*, YTILT = *ytilt*, ZTILT = *ztilt* (Python)
'XTILT', *xtilt*
'YTILT', *ytilt*
'ZTILT', *ztilt* (Matlab)

(Optional) Define the rotation of the DM about the X, Y, and Z axes. By default the DM is coplanar with the wavefront array and perpendicular to the optical axis. If one or more of these is set, the DM surface pattern is rotated in three-dimensional space and orthographically projected onto the wavefront. When viewed as an image, the wavefront coordinate system assumes the wavefront and initial DM surface are in the X-Y plane with the first pixel in the lower left. The system is left-handed with the +Z axis towards the observer. By default the rotations occur in X, Y, then Z order unless the *ZYX* switch is set.

/XYZ *-or-* /ZYX (IDL)
XYZ=*True* or *False* *-or-* ZYX=*True* or *False* (Python)
'XYZ' *-or-* 'ZYX' (Matlab)

(Optional) Switches that specify whether the rotation of the DM surface occurs in X, Y, then Z order (the default) or Z, Y, then X. The default is X, Y, then Z (*XYZ*).

Examples

Provide a 32 x 32 map of DM actuator (not surface) heights and set a DM to them. The actuators are spaced by 5 mm, and the center of the beam should fall on the center of the middle actuator:

IDL:

```
…
dm_xc = 50.0d
dm_yc = 50.0d
dm_spacing = 0.005d
prop_dm, wavefront, dm_heights, dm_xc, dm_yc, dm_spacing
```

Python:

```
…
dm_xc = 50.0
dm_yc = 50.0
dm_spacing = 0.005
proper.prop_dm( wavefront, dm_heights, dm_xc, dm_yc, dm_spacing )
```

Matlab:

```
…
dm_xc = 50.0;
dm_yc = 50.0;
dm_spacing = 0.005;
wavefront = prop_dm( wavefront, dm_heights, dm_xc, dm_yc, dm_spacing );
```

PROP_ELLIPSE

Return a 2D image containing a filled ellipse with the major axis aligned along either the X or Y axis. The edge of the ellipse is antialiased (the value of an edge pixel varies between 0.0 and 1.0 in proportion to the amount of a pixel covered by the circle; the accuracy of antialiasing is set using **PROP_SET_ANTIALIASING**). Note that this routine does not modify the wavefront structure. **PROP_ELLIPSE** may be required when creating multi-aperture masks, otherwise users should use **PROP_ELLIPTICAL_APERTURE** and **PROP_ELLIPTICAL_OBSCURATION** when possible.

Syntax

IDL: *image* = **prop_ellipse**(*wavestruct, x_radius, y_radius [, xoff, yoff] [, ROTATION=angle] [, /DARK] [, /NORM]*)

Python: *image* = **proper.prop_ellipse**(*wavestruct, x_radius, y_radius [, xoff, yoff] [, ROTATION=angle] [, DARK=True/False] [, NORM=True/False]*)

Matlab: *image* = **prop_ellipse**(*wavestruct, x_radius, y_radius [, 'XC', xoff] [, 'YC', yoff] [, 'ROTATION', angle] [, 'DARK'] [, 'NORM', 1 or 0]*);

Returns

image
: (Required) A two-dimensional image containing a filled, antialiased ellipse scaled between 0.0 to 1.0.

Arguments

wavestruct
: (Required) The current wavefront structure (used to obtain sampling information).

x_radius, y_radius
: (Required) Radii of the ellipse along the X and Y image axes (in meters, unless *NORM* switch is set).

xoff, yoff (IDL)
xoff, yoff (Python)
: (Optional) X and Y axis offsets of the ellipse center relative to center of wavefront grid (in meters, unless *NORM* switch is set). If not specified, the ellipse is centered at the center of the wavefront grid.

Keywords and Switches

/DARK (IDL)
DARK=*True* or *False* (Python)
'DARK' (Matlab)
: (Optional) Switch indicating that the interior of the ellipse should be set to 0.0 and the exterior to 1.0. By default, it is 1.0 interior and 0.0 exterior.

ROTATION=*angle* (IDL)
ROTATION=*angle* (Python)
'ROTATION', *angle* (Matlab)

(Optional) Rotation of the ellipse in degrees about its center.

/NORM (IDL)
NORM=*True* or *False* (Python)
'NORM', 1 or 0 (Matlab)

(Optional) Switch that indicates that *x_radius* and *y_radius* and, if provided, *xoff* and *yoff*, are in fractions of the current beam radius (for an unaberrated beam) rather than in meters.

'XC', xoff
'YC', yoff (Matlab)

(Optional) X and Y axis offsets of the ellipse center relative to center of wavefront grid (in meters, unless *NORM* switch is set). If not specified, the ellipse is centered at the center of the wavefront grid.

Examples

Create an image with a 1.2 m semi-major axis and 0.7 semi-minor axis ellipse in it:

IDL:
```
ellipse = prop_ellipse( wavefront, 1.2, 0.7 )
```

Python:
```
ellipse = proper.prop_ellipse( wavefront, 1.2, 0.7 )
```

Matlab:
```
ellipse = prop_ellipse( wavefront, 1.2, 0.7 );
```

Create a mask with two 12 mm -radius circular holes separated by 10 mm:

IDL:
```
mask = prop_ellipse( wavefront, 0.012, 0.012, -0.005, 0.0 ) + $
          prop_ellipse( wavefront, 0.012, 0.012, 0.005, 0.0 )
```

Python:
```
mask = proper.prop_ellipse( wavefront, 0.012, 0.012, -0.005, 0.0 ) +
          proper.prop_ellipse( wavefront, 0.012, 0.012, 0.005, 0.0 )
```

Matlab:
```
mask = prop_ellipse( wavefront, 0.012, 0.012, 'XC', -0.005, 'YC', 0.0 ) + ...
          prop_ellipse( wavefront, 0.012, 0.012, 'XC', 0.005, 'YC', 0.0 );
```

See Also

PROP_CIRCULAR_APERTURE, PROP_CIRCULAR_OBSCURATION, PROP_ELLIPTICAL_APERTURE, PROP_ELLIPTICAL_OBSCURATION, PROP_RECTANGLE, PROP_RECTANGULAR_APERTURE, PROP_RECTANGULAR_OBSCURATION, PROP_SET_ANTIALIASING

PROP_ELLIPTICAL_APERTURE

Multiply the current wavefront by an elliptical aperture (clear inside, dark outside). The ellipse is antialiased (the value of an edge pixel varies between 0.0 and 1.0 in proportion to the amount of a pixel covered by the ellipse; the accuracy of antialiasing is set using **PROP_SET_ANTIALIASING**). To create a multi-aperture mask, see PROP_ELLIPSE.

Syntax

IDL: **prop_elliptical_aperture**, *wavestruct, x_ radius, y_radius [, xoff, yoff] [, ROTATION=angle] [, /NORM]*

Python: **proper.prop_elliptical_aperture**(*wavestruct, x_ radius, y_radius [, xoff, yoff] [, ROTATION=angle] [, NORM=True/False])*

Matlab: *wavestruct_out* = **prop_elliptical_aperture**(*wavestruct_in, x_ radius, y_radius [, 'XC', xoff] [, 'YC', yoff] [, 'ROTATION', angle] [, 'NORM']*);

Returns

wavestruct_out (Matlab)
(Required) The modified wavefront structure.

Arguments

wavestruct (IDL)
wavestruct (Python)
wavestruct_in (Matlab)
(Required) The current wavefront structure.

x_radius, y_radius
(Required) Radii of the aperture along X and Y image axes in meters, or if the *NORM* switch is set, the radii in terms of the fraction of the beam radius at the current surface.

xoff, yoff (IDL)
xoff, yoff (Python)
(Optional) X and Y axis offsets of the ellipse center from the center of the wavefront grid. These are specified in meters unless the *NORM* switch is set, in which case they are in fractions of the current beam radius. By default, the ellipse is centered at the center of the wavefront grid.

Keywords and Switches

ROTATION=*angle* (IDL)
ROTATION=*angle* (Python)
'ROTATION', *angle* (Matlab)
(Optional) Rotation of the ellipse in degrees about its center.

/NORM (IDL)
NORM=*True* or *False* (Python)
'NORM' (Matlab)

(Optional) Switch that indicates that *x_radius* and *y_radius* and, if provided, *xoff* and *yoff*, are in fractions of the current beam radius (for an unaberrated beam) rather than in meters.

'XC', xoff
'YC', yoff (Matlab)

(Optional) X and Y axis offsets of the ellipse center relative to center of wavefront grid (in meters, unless *NORM* switch is set). If not specified, the ellipse is centered at the center of the wavefront grid.

Examples

Multiply the wavefront by an elliptical entrance aperture with axis radii of 0.6 and 0.3 of the beam radius (the beam radius is measured along the X axis):

IDL:
```
prop_elliptical_aperture, wavefront, 0.6, 0.35, /NORM
```

Python:
```
proper.prop_elliptical_aperture( wavefront, 0.6, 0.35, NORM=True )
```

Matlab:
```
wavefront = prop_elliptical_aperture( wavefront, 0.6, 0.35, 'NORM' );
```

Multiply the wavefront by an ellipse of 10.0 mm by 5 mm radii and centered +1 mm from the wavefront center along the X axis:

IDL:
```
prop_circular_aperture, wavefront, 0.010, 0.005, 0.001, 0.0
```

Python:
```
proper.prop_circular_aperture( wavefront, 0.010, 0.005, 0.001, 0.0 )
```

Matlab:
```
wavefront = prop_circular_aperture( wavefront, 0.01, 0.005, ...
                'XC', 0.001, 'YC', 0.0 );
```

See Also

PROP_CIRCULAR_APERTURE, PROP_CIRCULAR_OBSCURATION, PROP_ELLIPTICAL_OBSCURATION, PROP_RECTANGLE, PROP_RECTANGULAR_APERTURE, PROP_RECTANGULAR_OBSCURATION, PROP_SET_ANTIALIASING

PROP_ELLIPTICAL_OBSCURATION

Multiply the current wavefront by an elliptical obscuration (dark inside, clear outside). The ellipse is antialiased (the value of an edge pixel varies between 0.0 and 1.0 in proportion to the amount of a pixel covered by the ellipse; the accuracy of antialiasing is set using **PROP_SET_ANTIALIASING**).

Syntax

IDL: **prop_elliptical_obscuration**, *wavestruct, x_ radius, y_radius [, xoff, yoff] [, ROTATION=angle] [, /NORM]*

Python: **proper.prop_elliptical_obscuration**(*wavestruct, x_ radius, y_radius [, xoff, yoff] [, ROTATION=angle] [, NORM=True/False])*

Matlab: *wavestruct_out =* **prop_elliptical_obscuration**(*wavestruct_in, x_ radius, y_radius [, 'XC', xoff] [, 'YC', yoff] [, 'ROTATION', angle] [, 'NORM']*);

Returns

wavestruct_out (Matlab)
(Required) The modified wavefront structure.

Arguments

wavestruct (IDL, Python)
wavestruct_in (Matlab)
(Required) The current wavefront structure.

x_radius, y_radius
(Required) Radii of aperture along X and Y image axes in meters, or if the *NORM* switch is set, the radii in terms of the fraction of the beam radius at the current surface.

xoff, yoff (IDL)
xoff, yoff (Python)
(Optional) X and Y axis offsets of the ellipse center from the center of the wavefront grid. These are specified in meters unless the *NORM* switch is set, in which case they are in fractions of the current beam radius. By default, the ellipse is centered at the center of the wavefront grid.

Keywords and Switches

ROTATION=*angle* (IDL)
ROTATION=*angle* (Python)
'ROTATION', *angle* (Matlab)
(Optional) Rotation of the ellipse in degrees about its center.

/NORM (IDL)
NORM=*True* or *False* (Python)
'NORM' (Matlab)

(Optional) Switch that indicates that *x_radius* and *y_radius* and, if provided, *xoff* and *yoff*, are in fractions of the current beam radius (for an unaberrated beam) rather than in meters.

'XC', xoff
'YC', yoff (Matlab)

(Optional) X and Y axis offsets of the ellipse center relative to center of wavefront grid (in meters, unless *NORM* switch is set). If not specified, the ellipse is centered at the center of the wavefront grid.

Examples

Multiply the wavefront by an elliptical obscuration aperture 10 mm by 5 mm in size:

IDL:
```
prop_elliptical_obscuration, wavefront, 0.005, 0.0025, /NORM
```

Python:
```
proper.prop_elliptical_obscuration( wavefront, 0.005, 0.0025, NORM=True )
```

Matlab:
```
wavefront = prop_elliptical_obscuration( wavefront, 0.005, 0.0025, 'NORM' );
```

See Also

PROP_CIRCULAR_APERTURE, PROP_CIRCULAR_OBSCURATION, PROP_ELLIPSE, PROP_ELLIPTICAL_APERTURE, PROP_RECTANGLE, PROP_RECTANGULAR_APERTURE, PROP_RECTANGULAR_OBSCURATION, PROP_SET_ANTIALIASING

PROP_END

Conclude the propagation through an optical system. **PROP_END** must be called as the last PROP routine in a prescription. By default, **PROP_END** will convert the *wavestruct* structure to a real-valued array containing the modulus squared of wavefront, resulting in the intensity pattern. Setting the *NOABS* switch will bypasses the calculation, and the structure will simply be converted to the wavefront array. **PROP_END** can also return only the central specified portion of the wavefront array when the *EXTRACT* keyword is specified.

Syntax

IDL: **prop_end**, *wavestruct [, sampling] [, EXTRACT=n] [, /NOABS]*

Python: (*wavefront, sampling*) = **proper.prop_end**(*wavestruct [, EXTRACT=n] [, NOABS=True/False]*)

Matlab: *wavefront = - OR -*
[*wavefront, sampling*] =
prop_end(*wavestruct [, 'EXTRACT', n] [, 'NOABS']*);

Returns

wavefront (Python)
wavefront (Matlab)
(Required) The two dimensional wavefront intensity (the modulus-squared of the wavefront, unless the *NOABS* switch is set).

sampling (Python)
sampling (Matlab)
(Optional) Variable in which the sampling of the wavefront in meters is returned.

Arguments

wavestruct
(Required) Upon call, the current wavefront structure. In IDL, **PROP_END** will replace the structure with the wavefront. By default the intensity (modulus squared) of the wavefront will be returned, unless the *NOABS* switch is set.

sampling (IDL)
(Optional) Variable in which the sampling of the wavefront in meters is returned.

Keywords and Switches

EXTRACT=*n* (IDL)
EXTRACT=*n* (Python)
'EXTRACT', *n* (Matlab)
(Optional) Specifies the size of a subarray (*n* by *n* pixels) centered on the wavefront to extract from the wavefront array.

/NOABS (IDL)
NOABS=*True* or *False* (Python)
'NOABS' (Matlab)

(Optional) Switch that causes the wavefront array to be returned without taking the modulus squared, the default. The wavefront array will have complex values.

See Also

PROP_BEGIN

PROP_END_SAVESTATE

Terminate the current state saving that was initiated with PROP_INIT_SAVESTATE. This deletes the temporary files created by PROP_STATE.

Syntax

IDL: **prop_end_savestate**

Python: **proper.prop_end_savestate**

Matlab: **prop_end_savestate**

Examples

A savestate sequence goes something like this in IDL:

```
prop_init_savestate
      …
if ( prop_is_statesaved(wavefront) ) then goto, skipstuff
      …
      prop_state, wavefront
skipstuff:
      prop_state, wavefront
      …
prop_end_savestate
```

See Also

PROP_INIT_SAVESTATE, PROP_IS_STATESAVED, PROP_STATE

PROP_ERRORMAP

Read in a wavefront, mirror surface, or amplitude error map from a FITS image file and add it to the current wavefront (wavefront and surface errors are added to the phase, while the wavefront is multiplied by the amplitude error). One (and only one) of *MIRROR_SURFACE*, *WAVEFRONT*, or *AMPLITUDE* switch must be set to properly apply the error map to the wavefront. For surface or wavefront maps, the error is assumed to be in meters, unless either of the *NM* or *MICRON* switches is set to specify the units. The amplitude map is assumed to range from 0.0 to 1.0. The map can be multiplied by a constant specified using the optional *MULTIPLY* parameter. The sampling and wavefront center coordinates of the map can be specified with the *SAMPLING*, *XC_MAP*, and *YC_MAP* keywords. The map will be interpolated to match the current wavefront sampling. The map may be rotated about the wavefront center by specifying the *ROTATEMAP* keyword.

Syntax

IDL: **prop_errormap**, *wavestruct, filename [, xshift, yshift] [, /AMPLITUDE]*
[, MAGNIFY=constant] [, MAP=array] [, /MICRONS] [, /MIRROR_SURFACE]
[, MULTIPLY=constant] [, /NM] [, ROTATEMAP=angle] [, SAMPLING=sampling]
[, /WAVEFRONT] [, XC_MAP=xc, YC_MAP=yc]

Python: *[map =]* **proper.prop_errormap(** *wavestruct, filename [, xshift, yshift]*
[, AMPLITUDE=True/False] [, MAGNIFY=constant] [, MICRONS=True/False]
[, MIRROR_SURFACE=True/False] [, MULTIPLY=constant] [, NM=True/False]
[, ROTATEMAP=angle] [, SAMPLING=sampling] [, WAVEFRONT=True/False]
[, XC_MAP=xc, YC_MAP=yc])

Matlab: *wavestruct_out =* *- OR -*
[wavestruct_out, map] =
prop_errormap(*wavestruct_in, filename [, 'XSHIFT', xshift] [, 'YSHIFT', yshift]*
[, 'AMPLITUDE'] [, 'MAGNIFY', constant] [, 'MICRONS']
[, 'MIRROR_SURFACE'] [, 'MULTIPLY', constant] [, 'NM']
[, 'ROTATEMAP', angle] [, 'SAMPLING', sampling] [, 'WAVEFRONT']
[, 'XC_MAP', xc] [, 'YC_MAP', yc]);

Returns

wavestruct_out (Matlab)

(Required) The modified wavefront structure.

map (Python)
map (Matlab)

(Optional) Variable in which the error map will be returned for use by the user. The map will be in meters of wavefront or mirror surface error, unless it is an amplitude error map. All rotations and shifts will have been applied, and it will have the current wavefront sampling.

Arguments

wavestruct (IDL)
wavestruct (Python)
wavestruct_in (Matlab)
(Required) The current wavefront structure.

filename
(Required) The name of the FITS image file containing the map.

xshift, yshift (IDL)
xshift, yshift (Python)
(Optional) Parameters specifying the amount in meters to shift the map in the X and Y wavefront directions. Shifts occur before rotation, if any.

Keywords and Switches

/AMPLITUDE *-or-* /MIRROR_SURFACE *-or-* /WAVEFRONT (IDL)
AMPLITUDE=*True* or *False* *-or-* MIRROR_SURFACE=*True* or *False* *-or-* WAVEFRONT=*True* or *False* (Python)
'AMPLITUDE' *-or-* 'MIRROR_SURFACE' *-or-* 'WAVEFRONT' (Matlab)
(Optional) Indicates the type of the error map in the file (amplitude, wavefront, or mirror surface). The default type is wavefront. If the map is a mirror surface, it is multiplied by -2 to account for reflection to convert to wavefront error (e.g. a pit in the surface will induce an additional amount of phase lag).

MAP=*array* (IDL)
(Optional) Variable in which the error map will be returned for use by the user. The map will be in meters of wavefront or mirror surface error, unless it is an amplitude error map. All rotations and shifts will have been applied, and it will have the current wavefront sampling.

/MICRONS *-or-* /NM (IDL)
MICRON=*True* or *False* *-or-* NM=*True* or *False* (Python)
'MICRONS' *-or-* 'NM' (Matlab)
(Optional) Specifies that the wavefront or mirror surface errors in the map file are in microns or nanometers (meters is the default).

MULTIPLY=*constant* (IDL)
MULTIPLY=*constant* (Python)
'MULTIPLY', *constant* (Matlab)
(Optional) Specifies a constant by which the error map should be multiplied.

ROTATEMAP=*angle* (IDL)
ROTATEMAP=*angle* (Python)
'ROTATEMAP', *angle* (Matlab)
(Optional) Specifies the angle in degrees counterclockwise that the map will be rotated around the center of the wavefront after the shifts (if any) have been applied.

SAMPLING=*sampling* (IDL)
SAMPLING=*sampling* (Python)
'SAMPLING', *sampling* (Matlab)

(Optional) Specifies the spacing in meters between samples in the image map file. If this keyword is not specified, then the image file header must contain either the PIXSIZE keyword (sampling per pixel in meters) or the RADPIX keyword (the beam diameter in pixels in the map). Note that RADPIX header keyword value will override either the *SAMPLING* keyword or the PIXSIZE header keyword. Using one of these values, the image file map will be interpolated to match the current sampling of the wavefront.

XC_MAP=*xcenter*
YC_MAP=*ycenter* (IDL)
XC_MAP=*xcenter*
YC_MAP=*ycenter* (Python)
'XC_MAP', *xcenter*
'YC_MAP', *ycenter* (Matlab)

(Optional) Specifies the pixel coordinates of the center of the wavefront in the image file map. The center of the first pixel in the map is (0.0, 0.0). By default, the center is assumed to be (nx/2, ny/2) in IDL or Python and (nx/2+1,ny/2+1) in Matlab, where nx and ny are the pixel dimensions of the map.

'XSHIFT', *xshift*
'YSHIFT', *yshift* (Matlab)

(Optional) Parameters specifying the amount in meters to shift the map in the X and Y wavefront directions. Shifts occur before rotation, if any.

Examples

Read in the wavefront error map from the file map.fits, which is in nanometers and sampled by 1.0 mm, and add it to the current wavefront array phase.

IDL:
```
prop_errormap, wavefront, 'map.fits', /NM, /WAVEFRONT, SAMPLING=0.001
```

Python:
```
proper.prop_errormap( wavefront, 'map.fits', NM=True, WAVEFRONT=True,
          SAMPLING=0.001 )
```

Matlab:
```
wavefront = prop_errormap( wavefront, 'map.fits', 'WAVEFRONT', 'NM', ...
                'SAMPLING', 0.001 );
```

See Also

PROP_PSD_ERRORMAP, PROP_READMAP, PROP_ZERNIKES

PROP_FFTW_WISDOM (IDL, Python)

Generate an FFTW wisdom file containing the setup information necessary to compute an optimized Fast Fourier Transform for a given array size. If this is not done, the default (and likely less optimal) FFTW configuration will be done. (IDL only) This routine produces a wisdom file with a rootname specified by the PROPER_FFTW_WISDOM_FILE environment variable and a suffix specifying the grid size. (Python only) The wisdom file is stored in the user's home directory as a hidden file (e.g., ".prop_1024pix1threads_wisdomfile"). The FFTW library must be installed and the PROPER FFTW interface compiled using **PROP_COMPILE_FFTW** before using this routine. The wisdom file is appropriate for a given system and array size; a change in system or installation of a new FFTW library (or perhaps even other system libraries) may require re-running this again to generate an appropriate wisdom file. The wisdom file is used only by PROPER routines. NOTE: it may take a few minutes grid sizes.

Syntax

IDL: **prop_fftw_wisdom**, *gridsize*

Python: **proper.prop_fftw_wisdom**(*gridsize*)

Arguments

gridsize
: The dimension of one side of the two-dimensional, square array for which wisdom will be attained.

See Also

PROP_COMPILE_FFTW, PROP_USE_FFTW

PROP_FIT_ZERNIKES

Fit circular Zernike polynomials to a two-dimensional map, returning the RMS amount of each aberration. The region of the map to be fit is defined by a two-dimensional mask. The Zernike polynomials are circular and orthonormal for clear apertures (up to an arbitrary number of polynomials) and centrally-obscured apertures (up to the first 22 polynomials). The Noll ordering scheme is used (use **PROP_NOLL_ZERNIKES** to generate a table of polynomials). Note that this routine ***does not*** perform phase unwrapping prior to fitting the map.

Syntax

IDL: **prop_fit_zernikes**, *input_map, mask, radius, num_z, z_coeff [, fitted_map]*
[, OBSCURATION_RATIO=value] [, XC=xcenter, YC=ycenter]

Python: *z_coeff =* *- OR -*
(z_coeff, fitted_map) =
proper.prop_fit_zernikes(*input_map, mask, radius, num_z [, FIT=True]*
[, OBSCURATION_RATIO=value] [, XC=xcenter, YC=ycenter])

Matlab: *z_coeff =* *- OR -*
[z_coeff, fitted_map] =
prop_fit_zernikes(*input_map, mask, radius, num_z*
[, 'OBSCURATION_RATIO', value] [, 'XC', xcenter] [, 'YC', ycenter]);

Returns

z_coeff (Python)
z_coeff (Matlab)

(Required) A variable to contain the fitted Zernike polynomial coefficients (RMS error) fitted by this routine, assuming the Noll ordering scheme. The first element, *z_coeff[0]*, corresponds to Z1 (piston). These coefficients will have the same units as the map being fitted. If the *FIT* switch is set, then this is part of the *(z_coeff, fitted_map)* (Python) or [*z_coeff, fitted_map*] (Matlab) tuple.

fitted_map (Python)
fitted_map (Matlab)

(Optional) Used only if the *FIT* switch is set, this is a variable in which the fitted Zernike polynomial map is returned as part of the *(z_coeff, fitted_map)* (Python) or [*z_coeff, fitted_map*] (Matlab) tuple. This map will have the same units as the map that was fit.

Arguments

input_map

(Required) The two-dimensional map to fit with circular Zernike polynomials. The returned Zernike coefficients will be in the same units used for this map.

mask

(Required) A two-dimensional mask with the same dimensions as *input_map* specifying whether the corresponding pixel in the map should be included in the fit (one or zero). This is usually defined by the pupil, aperture, or beam area on the surface.

radius

(Required) The radius in pixels of the circular region over which the Zernike polynomials are defined.

num_z

(Required) The number of Zernike polynomials to fit to the input map. The first *num_z* polynomials will be used, assuming the Noll ordering scheme. For obscured Zernikes, the maximum number is 22.

z_coeff (IDL)

A variable to contain the fitted Zernike polynomial coefficients (RMS error) fitted by this routine, assuming the Noll ordering scheme. The first element, *z_coeff[0],* corresponds to Z1 (piston). These coefficients will have the same units as the map being fitted.

FIT=*True* (Python)

(Optional) Specifies that the fitted map should be returned as part of the *(z_coeff, fitted_map*) tuple.

fitted_map (IDL)

(Optional) A variable to contain the fitted Zernike polynomial map. This map will have the same units as the map that was fit.

Keywords and Switches

OBSCURATION_RATIO=*value* (IDL)
OBSCURATION_RATIO=*value* (Python)
'OBSCURATION_RATIO', *value* (Matlab)

(Optional) Sets the ratio of the diameter of a circular central obscuration to the diameter of a circular aperture, ranging from 0.0 to 1.0. If defined, the returned Zernike coefficients are properly normalized for an obscured system (i.e. they represent the RMS aberrations over the unobscured area). Only the first 22 Zernike polynomials can be fit if this is set.

XC=*xcenter*
YC=*ycenter* (IDL)
XC=*xcenter*
YC=*ycenter* (Python)
'XC', *xcenter*
'YC', *ycenter* (Matlab)

(Optional) Specifies that the fitted map should be returned as part of the *(z_coeff, fitted_map*) (Python) or [*z_coeff, fitted_map*] (Matlab) tuple.

See Also

PROP_NOLL_ZERNIKES, PROP_ZERNIKES

PROP_FREE_THREADS (IDL, Matlab)

Destroys waiting processes created by PROP_RUN_MULTI with the KEEP_THREADS option.

Syntax

IDL: **prop_free_threads**

Matlab: **prop_free_threads**

Returns

Nothing.

Arguments

None.

See Also

PROP_RUN_MULTI

PROP_GET_AMPLITUDE

Return the amplitude distribution of the current wavefront, centered in the array.

NOTE: The wavefront array is renormalized by **PROP_DEFINE_ENTRANCE** to have a total intensity of one, which alters the amplitude. If the amplitude varies by zero to one prior to calling **PROPER_DEFINE_ENTRANCE**, afterwards it will not.

Syntax

IDL: *amp* = **prop_get_amplitude**(*wavestruct*)

Python: *amp* = **proper.prop_get_amplitude**(*wavestruct*)

Matlab: *amp* = **prop_get_amplitude**(*wavestruct*);

Returns

amp
(Required) The two-dimensional wavefront amplitude.

Arguments

wavestruct
(Required) The current wavefront structure.

See Also

PROP_GET_PHASE, PROP_GET_WAVEFRONT

PROP_GET_BEAMRADIUS

Return the current radius in meters of the pilot beam. For a unaberrated system, this provides a fairly accurate estimate of the beam radius for the current wavefront.

Syntax

IDL: *rad* = **prop_get_beamradius**(*wavestruct*)

Python: *rad* = **proper.prop_get_beamradius**(*wavestruct*)

Matlab: *rad* = **prop_get_beamradius**(*wavestruct*);

Returns

rad
(Required) The radius of the pilot beam in meters at the current location.

Arguments

wavestruct
(Required) The current wavefront structure.

See Also

PROP_FRATIO(), PROP_GET_GRIDSIZE(), PROP_GET_NYQUISTSAMPLING(), PROP_GET_SAMPLING_ARCSEC(), PROP_GET_SAMPLING()

PROP_GET_DISTANCETOFOCUS

Returns the distance in meters from the current position to the focus of the current wavefront. This is actually the distance from the current position to the beam waist of an unaberrated beam.

Syntax

IDL: *distance* = **prop_get_distancetofocus**(*wavestruct*)

Python: *distance* = **proper.prop_get_distancetofocus**(*wavestruct*)

Matlab: *distance* = **prop_get_distancetofocus**(*wavestruct*);

Returns

distance

(Required) The distance in meters of the current location from the nearest focus based on the pilot beam.

Arguments

wavestruct

(Required) The current wavefront structure.

PROP_GET_FRATIO

Return the focal ratio of the current pilot beam, which is a reasonably accurate estimate of the wavefront beam focal ratio in an unaberrated system. The focal ratio is computed by dividing the current distance to the beam waist by the diameter of the beam.

Syntax

IDL: *focal_ratio* = **prop_get_fratio**(*wavestruct*)

Python: *focal_ratio* = **proper.prop_get_fratio**(*wavestruct*)

Matlab: *focal_ratio* = **prop_get_fratio**(*wavestruct*);

Returns

focal_ratio
(Required) The current focal ratio based on the pilot beam.

Arguments

wavestruct
(Required) The current wavefront structure.

See Also

PROP_GET_WAVELENGTH

PROP_GET_GRIDSIZE

Return the dimension in pixels of one side of the wavefront array (which is square).

Syntax

IDL: *gridsize* = **prop_get_gridsize**()

Python: *gridsize* = **proper.prop_get_gridsize**(*wavestruct*)

Matlab: *gridsize* = **prop_get_gridsize**();

Returns

gridsize
(Required) The current wavefront grid diameter in pixels (integer).

Arguments

wavestruct (Python only)
(Required) The current wavefront structure.

See Also

PROP_GET_FRATIO, PROP_GET_SAMPLING, PROP_GET_WAVELENGTH

PROP_GET_NYQUISTSAMPLING

Return the sampling in meters that meets the Nyquist sampling criterion for the current wavefront, which is

$$sampling = F\lambda/2$$

where F is the current focal ratio of the pilot beam and λ is the current wavelength. The value returned really only makes sense when the current wavefront is at the focus of an unaberrated beam.

Syntax

IDL: *sampling* = **prop_get_nyquistsampling**(*wavestruct [,wavelength]*)

Python: *sampling* = **proper.prop_get_nyquistsampling**(*wavestruct [,wavelength]*)

Matlab: *sampling* = **prop_get_nyquistsampling**(*wavestruct [,wavelength]*);

Returns

sampling
(Required) The Nyquist sampling of the current wavefront in meters.

Arguments

wavestruct
(Required) The current wavefront structure.

wavelength
(Optional) The wavelength in meters that, if specified, is used instead of the current wavelength.

See Also

PROP_GET_FRATIO, PROP_GET_GRIDSIZE, PROP_GET_SAMPLING

PROP_GET_PHASE

Return the phase in radians of the current wavefront as an array. Note that this array may be phase wrapped (values are modulo 2π).

Syntax

IDL: *phase* = **prop_get_phase**(*wavestruct*)

Python: *phase* = **proper.prop_get_phase**(*wavestruct*)

Matlab: *phase* = **prop_get_phase**(*wavestruct*);

Returns

phase
(Required) The phase of the current wavefront in radians.

Arguments

wavestruct
(Required) The current wavefront structure.

See Also

PROP_GET_AMPLITUDE, PROP_GET_WAVEFRONT

PROP_GET_REFRADIUS

Return the radius (meters) of the reference surface to which the phase of the current wavefront is measured. If the surface is planar, the radius is zero. Assuming that forward propagation occurs from left to right, a negative radius indicates that the center of the reference surface is to the right of the current position (e.g. the beam is converging).

Syntax

IDL: *radius* = **prop_get_refradius**(*wavestruct*)

Python: *radius* = **proper.prop_get_refradius**(*wavestruct*)

Matlab: *radius* = **prop_get_refradius**(*wavestruct*);

Returns

radius

(Required) The reference radius of the pilot beam.

Arguments

wavestruct

(Required) The current wavefront structure.

PROP_GET_SAMPLING

Return the sampling in meters of the current wavefront.

Syntax

IDL: *sampling* = **prop_get_sampling**(*wavestruct*)

Python: *sampling* = **proper.prop_get_sampling**(*wavestruct*)

Matlab: *sampling* = **prop_get_sampling**(*wavestruct*);

Returns

sampling
(Required) The current wavefront sampling in meters.

Arguments

wavestruct
(Required) The current wavefront structure.

See Also

PROP_GET_GRIDSIZE, PROP_GET_NYQUISTSAMPLING, PROP_GET_SAMPLING_ARCSEC, PROP_GET_SAMPLING_RADIANS

PROP_GET_SAMPLING_ARCSEC

Return the sampling in arcseconds of the current wavefront. **The result is only valid at focus for beams with low aberrations.**

Syntax

IDL: *sampling* = **prop_get_sampling_arcsec**(*wavestruct*)

Python: *sampling* = **proper.prop_get_sampling_arcsec**(*wavestruct*)

Matlab: *sampling* = **prop_get_sampling_arcsec**(*wavestruct*);

Returns

sampling
(Required) The sampling of the current wavefront in arcseconds.

Arguments

wavestruct
(Required) The current wavefront structure.

See Also

PROP_GET_GRIDSIZE, PROP_GET_NYQUISTSAMPLING, PROP_GET_SAMPLING, PROP_GET_SAMPLING_RADIANS

PROP_GET_SAMPLING_RADIANS

Return the sampling in radians of the current wavefront. The result is only valid at focus for beams with low aberrations.

Syntax

IDL: *sampling* = **prop_get_sampling_radians**(*wavestruct*)

Python: *sampling* = **proper.prop_get_sampling_radians**(*wavestruct*)

Matlab: *sampling* = **prop_get_sampling_radians**(*wavestruct*);

Returns

sampling
(Required) The sampling of the current wavefront in radians.

Arguments

wavestruct
(Required) The current wavefront structure.

See Also

PROP_GET_GRIDSIZE, PROP_GET_NYQUISTSAMPLING, PROP_GET_SAMPLING, PROP_GET_SAMPLING_ARCSEC

PROP_GET_WAVEFRONT

Return the current complex-valued wavefront array with the wavefront centered in the array.

Syntax

IDL: *wavefront* = **prop_get_wavefront**(*wavestruct*)

Python: *wavefront* = **proper.prop_get_wavefront**(*wavestruct*)

Matlab: *wavefront* = **prop_get_wavefront**(*wavestruct*);

Returns

wavefront
(Required) The current two-dimensional, complex-valued wavefront array.

Arguments

wavestruct
(Required) The current wavefront structure.

See Also

PROP_GET_AMPLITUDE, PROP_GET_PHASE

PROP_GET_WAVELENGTH

Return the wavelength in meters of the current wavefront.

Syntax

IDL: *wavelength* = **prop_get_wavelength**(*wavestruct*)

Python: *wavelength* = **proper.prop_get_wavelength**(*wavestruct*)

Matlab: *wavelength* = **prop_get_wavelength**(*wavestruct*);

Returns

wavelength
(Required) The wavelength in meters of the current wavefront.

Arguments

wavestruct
(Required) The current wavefront structure.

See Also

PROP_GET_FRATIO, PROP_GET_GRIDSIZE, PROP_GET_SAMPLING

PROP_HEX_WAVEFRONT

Compute the transmission and wavefront phase errors for an hexagonally-arranged array of segmented hexagonal apertures (e.g. a segmented primary mirror telescope like Keck or JWST). The current wavefront is multiplied by a mask comprising the segments (drawn with antialiasing). Each segment may also have phase errors described by hexagonal Zernike polynomial coefficients (up to Z_{22}). The hexagonal segments are oriented with top and bottom sides parallel to the X axis (unless the *ROTATION* keyword is set). They are in the same order as the Noll-ordered circular Zernikes (see PROP_PRINT_ZERNIKES), but they are normalized for a hexagonal aperture (see Mahajan & Dai, JOSAA, 24, 2994 (2007)).

Syntax

IDL: **prop_hex_wavefront**, *wavestruct, nrings, hexrad, hexsep [, zernike_val]*
[, APERTURE=var] [, /DARKCENTER] [, /NO_APPLY] [, OMIT=omit]
[, PHASE=var] [, ROTATION=angle] [, XCENTER=xc, YCENTER=yc]

Python: *[aperture =]* - OR -
[(aperture, phase) =]
proper.prop_hex_wavefront(*wavestruct, nrings, hexrad, hexsep*
[, zernike_val] [, DARKCENTER=True/False] [, NO_APPLY=True/False]
[, OMIT=omit], [, ROTATION=angle] [, XCENTER=xc, YCENTER=yc])

Matlab: *wavestruct_out =* - OR -
[wavestruct_out, aperture] = - OR -
[wavestruct_out, aperture, phase] =
prop_hex_wavefront(*wavestruct_in, nrings, hexrad, hexsep*
[, 'DARKCENTER'] [, 'NO_APPLY'] [, 'OMIT', omit] [, 'ROTATION', angle]
[, 'XCENTER', xc] [, 'YCENTER', yc] ['ZERNIKE_VAL', zernike_val]);

Returns

wavestruct_out (Matlab)
(Required) The modified wavefront structure.

Arguments

wavestruct (IDL)
wavestruct (Python)
wavestruct_in (Matlab)
(Required) The current wavefront structure.

Returns

aperture (Python)
aperture (Matlab)
(Optional) A variable in which the aperture amplitude map is returned.

phase (Python)
phase (Matlab)

(Optional) A variable in which the phase map in meters is returned (returned only if *ZERNIKE_VAL* is defined).

Arguments

nrings

(Required) The number of rings of hexagons around the central hexagon (e.g., setting to 1 will result in a central hexagon surrounded by a ring of adjacent hexagons).

hexrad

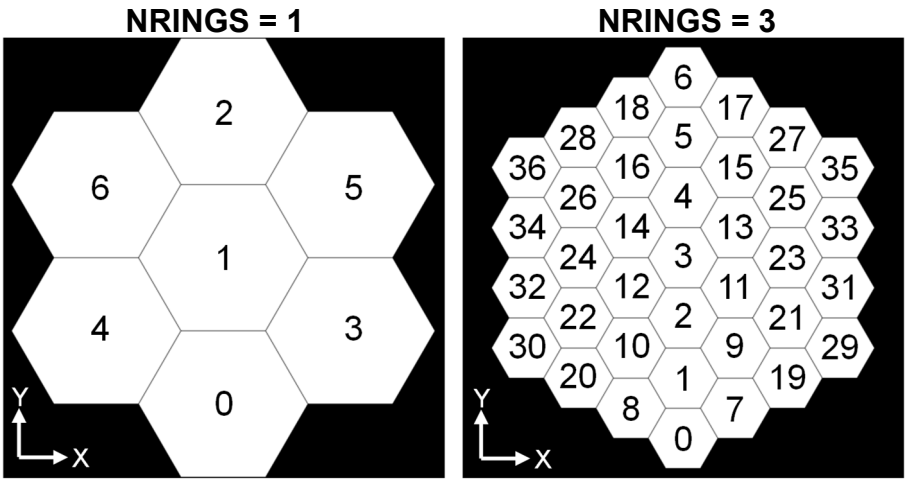
(Required) The distance in meters from the center of a hexagonal segment to a vertex.

hexsep

(Required) The distance in meters between the centers of two adjacent hexagonal segments.

zernike_val (IDL)
zernike_val (Python)
'ZERNIKE_VAL', *zernike_val* (Matlab)

(Optional) Array of dimensions (22, *num_hex*) in IDL, (22, num_hex) in Python, and (num_hex, 22) in Matlab, where *num_hex* = *nrings* × (*nrings* + 1) × 3 + 1, the number of segments (including the central one, whether darkened or not). Each row in this array specifies the hexagonal Zernike polynomial coefficients (Z_1 to Z_{22}) for a segment. The Zernikes are Noll-ordered (see prop_zernikes for a list of them). The values are in meters of RMS wavefront phase error. Even if the central segment is made dark using the *DARKCENTER* switch, there must be an entry for it.



Segment indexing scheme used for specification of Zernikes. Note that in Matlab the user should add 1 to the indices shown. The central segment is always numbered whether it is set to dark or not.

Keywords and Switches

APERTURE=*var* (IDL)
(Optional) Keyword set to a variable in which the aperture amplitude map is returned.

/DARKCENTER (IDL)
DARKCENTER=*True* or *False* (Python)
'DARKCENTER' (Matlab)
(Optional) If set, the central hexagonal segment will be dark.

/NO_APPLY (IDL)
NO_APPLY=*True* or *False* (Python)
'NO_APPLY' (Matlab)
(Optional) If set, the current wavefront is not modified. This is useful if the user wants to compute the aperture and/or phase arrays without applying them to the wavefront.

OMIT=*omit* (IDL)
OMIT=*omit* (Python)
'OMIT', *omit* (Matlab)
(Optional) Vector (or Python list) of segment IDs to omit drawing, starting with 0. If *zernike_val* is also specified, it must include entries for the omitted segments, but they are ignored. See above for segment numbering scheme.

PHASE=*var* (IDL)
(Optional) Keyword set to a variable in which the phase error map in meters is returned (use only if zernike_vals is defined).

ROTATION=*angle* (IDL)
ROTATION=*angle* (Python)
'ROTATION', *angle* (Matlab)
(Optional) The degrees counter-clockwise to rotate the aperture about its center.

XCENTER = *xc*
YCENTER = *yc* (IDL)
XCENTER = *xc*
YCENTER = *yc* (Python)
'XCENTER', *xc*
'YCENTER', *yc* (Matlab)
(Optional) Specifies the offset in meters of the aperture center from the center of the wavefront. By default, the aperture is centered on the wavefront.

Examples

Multiply the wavefront by a hexagonally segmented aperture and add aberrated segment phase errors. Each segment is given 50 nm RMS of random phase error (including piston).

IDL:

```
nrings = 2
hexrad = 0.75055
hexsep = 1.315
rms_segment = 50.0e-9
nhex = nrings * (nrings + 1) * 3 + 1
zval = fltarr(22, nhex)

;-- each segment has randomly different phase errors but the same RMS error
for i = 0, nhex-1 do begin
     val = randomu(seed,22) - 0.5
     rss_val = sqrt(total(val^2))
     zval(0,i) = val / rss_val * rms_segment
endfor

prop_hex_wavefront, wavefront, nrings, hexrad, hexsep, zval, /DARKCENTER
```

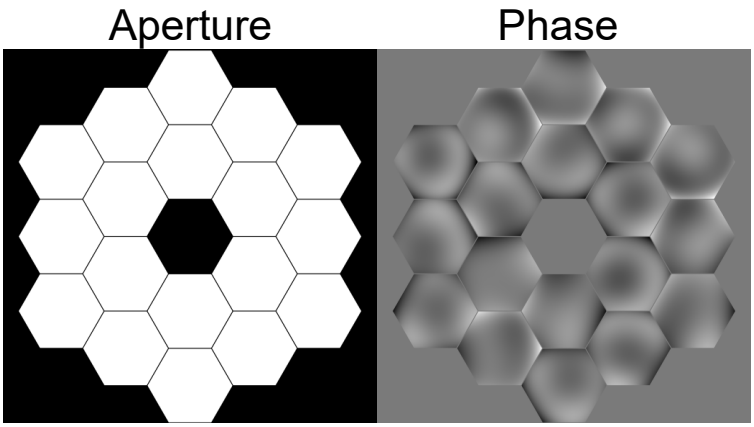
Python:

```
nrings = 2
hexrad = 0.75055
hexsep = 1.315
rms_segment = 50.0e-9
nhex = nrings * (nrings + 1) * 3 + 1
zval = np.zeros((nhex,22))

# each segment has randomly different phase errors but with the same RMS error
for i in range(0, nhex):
     val = np.random.uniform(-0.5, 0.5, 22)
     rss_val = np.sqrt(np.sum(val**2))
     zval[i,:] = val / rss_val * rms_segment

proper.prop_hex_wavefront( wavefront, nrings, hexrad, hexsep, zval,
     DARKCENTER=True )
```

Matlab:

```
nrings = 2;
hexrad = 0.75055;
hexsep = 1.315;
rms_segment = 50.0e-9;
nhex = nrings * (nrings + 1) * 3 + 1;
zval = zeros(nhex,22);

% each segment has randomly different phase errors but the same RMS error
rng('default');
for i = 1 : nhex
     val = zeros(1, 22);
     val(1:22) = rand(1, 22) - 0.5;
     rss_val = sqrt(sum(val.^2));
     zval(i, :) = val / rss_val * rms_segment;
end

wavefront = prop_hex_wavefront( wavefront, nrings, hexrad, hexsep, ...
               'ZERNIKE_VAL', zval, 'DARKCENTER' );
```

The above codes create and apply an aperture mask and phase map that look something like these:



This next example demonstrates the use of the OMIT keyword option:

IDL:

```
nrings = 4
hex_rad = 0.5407
hex_sep = 0.961

prop_hex_wavefront, wavefront, nrings, hex_rad, hex_sep, $
     OMIT=[0,8,51,52,59,60]
```

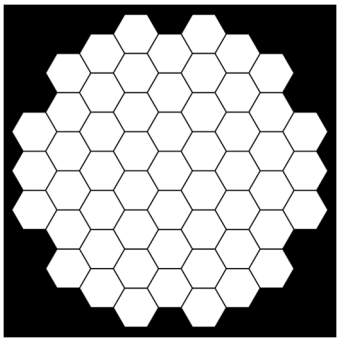
Python:

```
nrings = 4
hex_rad = 0.5407
hex_sep = 0.961

proper.prop_hex_wavefront( wavefront, nrings, hex_rad, hex_sep, \
     OMIT=[0,8,51,52,59,60] )
```

Matlab:

```
nrings = 4;
hex_rad = 0.5407;
hex_sep = 0.961;

wavefront = prop_hex_wavefront( wavefront, nrings, hexrad, hexsep, ...
               'OMIT', [0 8 51 52 59 60] );
```

These examples generate an aperture that looks like this:



Notes

If you want to generate a hexagonal array for a given number of rings (N_{rings}), aperture radius (R_{pup}), and gap width (W_{gap}), you can use the equations below to determine the radius (R_{seg}) and separation (Δ_{seg}) of the segments. Note that if the OMIT option is used, these values are for the parent aperture.

$$c = \cos(\pi/6)\,(2\,N_{rings} + 1)$$
$$b = W_{gap}\,N_{rings}$$

If R_{pup} is the radius of the circumscribed circular pupil diameter:

$$R_{seg} = 2\,(\mathrm{sqrt}[-b^2 + (2\,c\,R_{pup})^2 + R_{pup}^2] - 2\,b\,c) / (4c^2 + 1)$$

else if R_{pup} is the radius of the inscribed circular pupil diameter:

$$R_{seg} = (R_{pup} - b) / c$$

Then, for either case:

$$\Delta_{seg} = 2\,\cos(\pi/6)\,R_{seg} + W_{gap}$$

PROP_INIT_SAVESTATE

Initialize the state saving system. This routine should be called before calling PROP_RUN. See the section "Save States" for more details on this system. PROP_END_SAVESTATE must be called after all runs have been completed in order to clean up temporary storage files.

Syntax

IDL: **prop_end_savestate**

Python: **proper.prop_end_savestate**

Matlab: **prop_end_savestate**

Examples

A save state sequence goes something like this in IDL:

```
prop_init_savestate
for i = 0, 10 do begin
      prop_run, 'testprop', psf, 0.55, 1024
endfor
prop_end_savestate
```

See Also

PROP_END_SAVESTATE, PROP_IS_STATESAVED, PROP_STATE

PROP_IRREGULAR_POLYGON

Return an image containing a filled (interior value = 1) convex polygon with antialiased edges. If the DARK switch is set, then the interior = 1 and the exterior = 0. NOTE: This routine only works for convex polygons and will produce incorrect results for anything else.

Syntax

IDL: *image* = **prop_irregular_polygon(** *wavestruct, xvert, yvert [, /DARK] [, /NORM]* **)**

Python: *image* = **proper.prop_irregular_polygon(** *wavestruct, xvert, yvert [, DARK=True/False] [, NORM=True/False]* **)**

Matlab: *image* = **prop_irregular_polygon(** *wavestruct, xvert, yvert [, 'DARK'] [, 'NORM']*);

Returns

image

(Required) An image containing an antialiased filled polygon.

Arguments

wavestruct

(Required) The current wavefront structure.

xvert, yvert

(Required) One-dimensional arrays containing the X and Y vertex coordinates of the polygon. These values are offsets from the wavefront center in meters unless the NORM switch is set. The vertices must be in sequential order, either clockwise or counter-clockwise. If the last vertex does not have the same coordinates as the first, the polygon is assumed to begin and end at the first vertex.

Keywords and Switches

/DARK (IDL)
DARK=*True* or *False* (Python)
'DARK' (Matlab)

(Optional) Specifies the interior pixels of the polygon to be set to 1.0 and the exterior to 0.0. The reverse is the default.

/NORM (IDL)
NORM=*True* or *False* (Python)
'NORM' (Matlab)

(Optional) Specifies that the coordinates are normalized to the beam radius.

See Also

PROP_POLYGON

PROP_IS_STATESAVED

This function is used to determine if the state saving system has been enabled, a previous propagation run has saved a state for the current wavelength, and if it is possible to jump to a later position in the current prescription to avoid redundant computation through early parts of the prescription. It returns 1 if the state saving system is enabled, otherwise 0. For more details, see the section "Save States". Save states must not be used with **PROP_RUN_MULTI**.

Syntax

IDL: *status* = **prop_is_statesaved**(*wavestruct*)

Python: *status* = **proper.prop_is_statesaved**(*wavestruct*)

Matlab: *status* = **prop_is_statesaved**(*wavestruct*);

Arguments

wavestruct
(Required) The current wavefront structure.

Examples

PROP_IS_STATESAVED is used as shown in the follow snippet:

```
…
if ( prop_is_statesaved(wavefront) ) then goto, skipstuff
…
skipstuff:
   prop_state
   …
```

See Also

PROP_END_SAVESTATE, PROP_INIT_SAVESTATE, PROP_STATE

PROP_LENS

Alter the phase of the current wavefront as if it passed through a perfect lens or reflected from a perfect mirror.

Syntax

IDL: **prop_lens**, *wavestruct, lens_fl [, surface_name]*

Python: **proper.prop_lens**(*wavestruct, lens_fl [, surface_name]*)

Matlab: *wavestruct_out* = **prop_lens**(*wavestruct_in, lens_fl [, surface_name]*);

Returns

wavestruct_out (Matlab)
(Required) The modified wavefront structure.

Arguments

wavestruct (IDL)
wavestruct (Python)
wavestruct_in (Matlab)
(Required) The current wavefront structure.

lens_fl
(Required) The focal length of the lens in meters. It is positive for convex lenses (concave mirrors) and negative for concave lenses (convex mirrors).

surface_name
(Optional) String containing the name of the lens. The name will be printed when the lens is applied.

Examples

Apply a convex lens of 3 meter focal length:

IDL:
```
prop_lens, wavefront, 3.0
```

Python:
```
proper.prop_lens( wavefront, 3.0 )
```

Matlab:
```
wavefront = prop_lens( wavefront, 3.0 );
```

PROP_MAGNIFY

Resample an image using damped sinc interpolation. This function can be used only on real and complex-valued data. By default a Lanczos interpolation kernel is used. **PROP_MAGNIFY** calls an external C program, prop_szoom_c.c, to do the interpolation. If that executable does not exist, it calls the slower version written in the execution language. If the *QUICK* switch is specified, IDL's *interpolate* function is used with *CUBIC*=-0.5 to do the interpolation, or Python's *map_coordinates* function with order=3, or Matlab's *interp2* function with cubic interpolation. These are much faster than the sinc interpolator and typically produces similar results. Note that it is possible to have some erroneous negative values, especially near the cores of point spread functions, so the user should examine the results closely. It is often better to interpolate the complex field than the amplitude or intensity because it is often smoother.

Syntax

IDL: *new_image* = **prop_magnify**(*old_image, magnification [, n_new]*
[, /AMP_CONSERVE] [, /CONSERVE] [, /QUICK])

Python: *new_image* = **proper.prop_magnify**(*old_image, magnification [, n_new]*
[, AMP_CONSERVE=True/False] [, CONSERVE=True/False],
[, QUICK=True/False])

Matlab: *new_image* = **prop_magnify**(*old_image, magnification [, 'AMP_CONSERVE']*
[, 'CONSERVE'] [, 'QUICK'] [, 'SIZE_OUT', size_out]) ;

Returns

new_image
(Required) A 2D array containing the magnified image.

Arguments

old_image
(Required) A 2D array (real or complex) containing the image to magnify/demagnify.

magnification
(Required) The amount to magnify/demagnify. A value greater than 1.0 results in magnification.

n_new (IDL)
n_new (Python)
(Optional) The dimension of the output image (*n_new* by *n_new*) in pixels. By default, the new image size is the integral value of the old one times the magnication.

Keywords and Switches

/AMP_CONSERVE (IDL)
AMP_CONSERVE=*True* or *False* (Python)
'AMP_CONSERVE' (Matlab)
(Optional) If set, the input image is assumed to be the amplitude of a wavefront, not intensity, and the interpolated image will be divided by the magnification to conserve intensity (square of amplitude).

/CONSERVE (IDL)
CONSERVE=*True* or *False* (Python)
'CONSERVE' (Matlab)

(Optional) If set, the field intensity will remain constant. By default, the interpolated values are returned without scaling, leading to a change in the total value. If the input image is complex, then it is assumed that the input is an electric field, so the interpolated result will be divided by the magnification. If the image is not complex, then it is assumed that the input image is in intensity, so the interpolated image will be divided by the square of the magnification (use *AMP_CONSERVE* if the image is amplitude).

/QUICK (IDL)
QUICK=*True* or *False* (Python)
'QUICK' (Matlab)

(Optional) If set, IDL's "interpolate" function (with the CUBIC interpolation keyword set to -0.5) or Python's map_coordinates function (with order=3) or Matlab's interp2 function with cubic interpolation is used instead of the more exact, but much slower, sinc interpolator.

'SIZE_OUT', *size_out* (Matlab)

(Optional) The dimension of the output image (*size_out* by *size_out*) in pixels. By default, the new image size is the same as the old one.

Examples

Magnify old_image by a factor of 3.0 and store the central 256 x 256 pixels in new_image:

IDL:
```
new_image = prop_magnify( old_image, 3.0, 256 )
```

Python:
```
new_image = proper.prop_magnify( old_image, 3.0, 256 )
```

Matlab:
```
new_image = prop_magnify( old_image, 3.0, 'SIZE_OUT', 256 );
```

Using the QUICK method:

IDL:
```
new_image = prop_magnify( old_image, 3.0, 256, /QUICK )
```

Python:
```
new_image = proper.prop_magnify( old_image, 3.0, 256, QUICK=True )
```

Matlab:
```
new_image = prop_magnify( old_image, 3.0, 'SIZE_OUT', 256, 'QUICK' );
```

PROP_MULTIPLY

Multiply the current wavefront by a user-specified value or 2D array

Syntax

IDL: **prop_multiply**, *wavestruct, value*

Python: **proper.prop_multiply**(*wavestruct, value*)

Matlab: *wavestruct_out* = **prop_multiply**(*wavestruct_in, value*);

Returns

wavestruct_out (Matlab)
(Required) The modified wavefront structure.

Arguments

wavestruct (IDL)
wavestruct (Python)
wavestruct_in (Matlab)
(Required) The current wavefront structure.

value
(Required) Scalar value or two-dimensional array containing the values by which the current wavefront will be multiplied. The map in the array is assumed to be centered within the array, and it must have the same dimensions as the current wavefront gridsize (see **PROP_GET_GRIDSIZE**).

Examples

Multiply the wavefront amplitude by a 2D Gaussian with $\sigma = 0.01$ meters:

IDL:
```
r = prop_radius( wavefront )
gauss_amp = exp(-0.5*(r/0.01)^2)
prop_multiply, wavefront, gauss_amp
```

Python:
```
r = proper.prop_radius( wavefront )
gauss_amp = np.exp(-0.5*(r/0.01)**2)
proper.prop_multiply( wavefront, gauss_amp )
```

Matlab:
```
r = prop_radius( wavefront );
gauss_amp = exp(-0.5*(r./0.01).^2);
wavefront = prop_multiply( wavefront, gauss_amp );
```

See Also

PROP_ADD_PHASE, PROP_DIVIDE

PROP_NOLL_ZERNIKES

Return a string array in which each element contains the Zernike polynomial equation corresponding to the index of that element. The polynomials are orthonormal for an unobscured circular aperture. They follow the ordering convention of Noll (J. Opt. Soc. America, 66, 207 (1976)). Element (0) is always blank. The equations contain the variables *r* (normalized radius) and *t* (azimuth in radians). The aberrations defined by the polynomials have an RMS of 1.0 about a mean of 0.0.

Syntax

IDL: *list* = **prop_noll_zernikes**(*max_z*)

Python: *list* = **proper.prop_noll_zernikes**(*max_z*)

Matlab: *list* = **prop_noll_zernikes**(*max_z*);

Returns

list

(Required) An array of strings, each element containing the corresponding Zernike polynomial equation. In IDL and Python the first element (e.g., *list[0]*) is blank, and the first valid equation is stored in the next element (Z1 = piston; e.g., *list[1]*). In Matlab, the first element corresponds to Z1 (e.g., *list[1]*).

Arguments

max_z

(Required) The maximum number of polynomials to return. In IDL and Python, the returned array is dimensioned *max_z+1*, where the 1st element is to be ignored. In Matlab *max_z* elements are returned.

Examples

Display the first few Zernike terms:

IDL:

```
list = prop_noll_zernikes( 5 )
for i = 1, 5 do print, i, ' = ', list(i)
```

This will display the following:

```
1 = 1
2 = 2 * (r) * cos(t)
3 = 2 * (r) * sin(t)
4 = sqrt(3) * (2.0*r^2 - 1.0)
5 = sqrt(6) * (r^2) * sin(2*t)
```

Note that this does the same thing as calling **PROP_PRINT_ZERNIKES**.

See Also

PROP_FIT_ZERNIKES, PROP_PRINT_ZERNIKES, PROP_ZERNIKES

PROP_PIXELLATE

Integrate an image onto detector pixels of a specified size. This is done by convolving the Fourier transform of the input image with a sinc function representing the transfer function of an idealized square pixel. This is then Fourier transformed back, producing an array containing the image values integrated onto detector-sized pixels but at the spacing of the input image. This array is then interpolated to produce the image integrated onto detector-sized pixels with detector-pixel sampling. It is assumed that the spacing and size of the detector pixels are the same.

Syntax

IDL: *result* = **prop_pixellate**(*input_image, input_sampling, output_sampling [, output_dim]*)

Python: *result* = **proper.prop_pixellate**(*input_image, input_sampling, output_sampling [, output_dim]*)

Matlab: *result* = **prop_pixellate**(*input_image, input_sampling, output_sampling [, output_dim]*);

Returns

result

(Required) The pixellated image sampled by *output_sampling* with dimensions *output_dim* × *output_dim*.

Arguments

input_image

(Required) The real-valued image (square) to be pixellated.

input_sampling

(Required) The sampling of the input image, in the same units as *output_sampling*.

output_sampling

(Required) The sampling of the output image (=size of the square detector pixel), in the same units as *output_sampling*.

output_dim

(Optional) Specified the dimension (*output_dim* by *output_dim*) of the pixellated image. By default, the size is set as necessary to include all values in the input image.

Examples

Map an image sampled by 15 μm onto a detector with 30 μm pixels, keeping the image dimension in pixels the same:

IDL:
```
output = prop_pixellate( input_image, 15.0, 30.0 )
```

Python:
```
output = proper.prop_pixellate( input_image, 15.0, 30.0 )
```

Matlab:
```
output = prop_pixellate( input_image, 15.0, 30.0 );
```

PROP_POLYGON

Return a 2D image containing a filled, antialiased polygon. The polygon is symmetrical (the vertices are all the same distance from the center at are at equal angles from each other). This routine does not modify the wavefront structure. The polygon will have one vertex along the +X axis from its center, unless the ROTATION keyword is specified.

Syntax

IDL: *image* = **prop_polygon**(*wavestruct, nvert, radius [, xc, yc] [, /DARK] [, /NORM] [, ROTATION=angle]*)

Python: *image* = **proper.prop_polygon**(*wavestruct, nvert, radius [, xc, yc] [, DARK=True/False] [, NORM=True/False] [, ROTATION=angle]*)

Matlab: *image* = **prop_polygon**(*wavestruct, nvert, radius [, 'XC', xc] [, 'YC', yc] [, 'DARK'] [, 'NORM'] [, 'ROTATION', angle]*);

Returns

image
(Required) An image containing an antialiased, filled polygon.

Arguments

wavestruct
(Required) The current wavefront structure (used to obtain sampling information).

radius
(Required) The distance from any vertex to the center of the polygon (in meters, unless *NORM* switch is set).

xc, yc (IDL)
xc, yc (Python)
(Optional) Center of the polygon relative to center of wavefront grid (in meters, unless *NORM* switch is set). If not specified, it is centered within the grid.

Keywords and Switches

/DARK (IDL)
DARK=*True* or *False* (Python)
'DARK' (Matlab)
(Optional) Switch indicating that the interior of the polygon should be set to 0.0 and the exterior to 1.0. By default, it is 1.0 interior and 0.0 exterior.

/NORM (IDL)
NORM=*True* or *False* (Python)
'NORM' (Matlab)
(Optional) Switch specifying that radius is specified as a fraction of the beam radius at the current surface.

ROTATION=*angle* (IDL)
ROTATION=*angle* (Python)
'ROTATION', *angle* (Matlab)
(Optional) Specifies the degrees counter-clockwise to rotate the polygon around its center.

'XC', *xc*
'YC', *yc* (Matlab)
(Optional) Center of the polygon relative to center of wavefront grid (in meters, unless *NORM* switch is set). If not specified, it is centered within the grid.

Examples

Create an image with a hexagon with vertices 0.5 meters from its center:

IDL:
```
hex = prop_polygon( wave, 6, 0.5 )
```

Python:
```
hex = proper.prop_polygon( wave, 6, 0.5 )
```

Matlab:
```
hex = prop_polygon( wave, 6, 0.5 );
```

See Also

PROP_ELLIPSE, PROP_IRREGULAR_POLYGON, PROP_RECTANGLE

PROP_PRINT_ZERNIKES

Print to the display the equations for the first *N* Noll-ordered Zernike polynomials for an unobscured, circular aperture.

Syntax

IDL: **prop_print_zernikes**, *n*

Python: **proper.prop_print_zernikes**(*n*)

Matlab: **prop_print_zernikes**(*n*);

Arguments

n

(Required) The number of Zernike polynomials to print (1 to n).

Examples

Display the first five Zernike polynomials:

```
IDL:   prop_print_zernikes, 5
```

This produces the following output on the terminal:

```
1  =  1
2  =  2 * (r) * cos(t)
3  =  2 * (r) * sin(t)
4  =  sqrt(3) * (2.0*r^2 - 1.0)
5  =  sqrt(6) * (r^2) * sin(2*t)
```

See Also

PROP_FIT_ZERNIKES, PROP_NOLL_ZERNIKES, PROP_ZERNIKES

PROP_PROPAGATE

Propagate the current wavefront a specified distance (either forward or backward from the current position). The propagator will select the appropriate method (Fresnel or angular spectrum).

Syntax

IDL: **prop_propagate**, *wavestruct, dz [, surface_name] [, /TO_PLANE]*

Python: **proper.prop_propagate**(*wavestruct, dz [, surface_name] [, TO_PLANE=True/False]*)

Matlab: *wavestruct_out* = **prop_propagate**(*wavestruct_in, dz [,'SURFACE_NAME', surface_name] [, 'TO_PLANE']*);

Returns

wavestruct_out (Matlab)
(Required) The modified wavefront structure.

Arguments

wavestruct (IDL)
wavestruct (Python)
wavestruct_in (Matlab)
(Required) The current wavefront structure.

dz
(Required) The distance in meters over which to propagate the current wavefront. Forward propagation is a positive distance.

surface_name (IDL)
surface_name (Python)
(Optional) String containing the name of the surface to which the wavefront is being propagated. The name will be printed out on the display during propagation.

Keywords and Switches

'SURFACE_NAME', *surface_name* (Matlab)
(Optional) String containing the name of the surface to which the wavefront is being propagated. The name will be printed out on the display during propagation.

/TO_PLANE (IDL)
TO_PLANE=*True* or *False* (Python)
'TO_PLANE' (Matlab)
(Optional) Setting this switch forces the wavefront to be propagated to a plane. If the current wavefront is in the far field, it is propagated to the beam waist and then propagated to the final plane using an angular spectrum propagator.

Examples

Propagate the current wavefront forward by 3.0 meters to ‘mirror 1’:

IDL:
```
prop_propagate, wavefront, 3.0, ’mirror 1’
```

Python:
```
proper.prop_propagate( wavefront, 3.0, ’mirror 1’ )
```

Matlab:
```
wavefront = prop_propagate( wavefront, 3.0, ...
                    'SURFACE_NAME', ’mirror 1’ );
```

PROP_PSD_ERRORMAP

Create a realization of a two-dimensional surface, wavefront, or amplitude error map from a specified two-dimensional power spectral density (PSD) profile. This is often called a "phase screen" by ground-based adaptive optics modelers. This map is applied to the current wavefront (added if wavefront or surface, multiplied if amplitude). Because the map is generated using a Fourier transform without any additional processing, the lowest spatial frequency it includes (in cycles/diameter) is equal to the pupil-to-grid size ratio.

Syntax

IDL: **prop_psd_errormap**, *wavestruct, amp, b, c*
[, AMPLITUDE=value] [, FILE=string] [, INCLINATION=angle]
[, MAP=variable] [, MAX_FREQUENCY=value] [, /MIRROR] [, /NO_APPLY]
[, /RMS] [, ROTATION=angle] [, /TPF]

Python: *[map =]* **proper.prop_psd_errormap**(*wavestruct, amp, b, c*
[, AMPLITUDE=value] [, FILE=string] [, INCLINATION=angle]
[, MAX_FREQUENCY=value] [, MIRROR=True/False] [, NO_APPLY=True/False]
[, RMS=True/False] [, ROTATION=angle] [, TPF=True/False])

Matlab: *wavestruct_out =* - OR -
[wavestruct_out, map] =
prop_psd_errormap(*wavestruct_in, amp, b, c*
[, 'AMPLITUDE', value] [, 'FILE', string] [, 'INCLINATION', angle]
[, 'MAX_FREQUENCY', value] [, 'MIRROR'] [, 'NO_APPLY']
[, 'RMS'] [, 'ROTATION', angle] [, 'TPF']);

Returns

wavestruct_out (Matlab)
(Required) The modified wavefront structure.

map (Python)
map (Matlab)
(Optional) The two-dimensional wavefront, surface, or amplitude error map created by this routine.

Arguments

wavestruct (IDL)
wavestruct (Python)
wavestruct_in (Matlab)
(Required) The current wavefront structure.

amp
(Required) The low-spatial-frequency error. By default this is the wavefront error power per areal spatial frequency (units are meters^4). If the *MIRROR* switch is set, this is assumed to be surface error (wavefront error will be twice this due to reflection). If the *AMPLITUDE* keyword is specified, then this is assumed to be the total RMS error of the entire map. If the *RMS* switch is set, then the entire error map will be renormalized to have an RMS value set to this parameter.

b

(Required) The correlation length parameter (cycles/meter). This indicates the spatial frequency where the PSD curve transitions from a flat profile at low frequencies to a sloping one at higher frequencies.

c

(Required) The high-spatial-frequency PSD profile power law exponent.

Keywords and Switches

AMPLITUDE=*value* (IDL)
AMPLITUDE=*value* (Python)
'AMPLITUDE', *value* (Matlab)

(Optional) This keyword, if set to a value, indicates that an amplitude, rather than surface or wavefront, error map with a maximum set by the keyword value is to be generated and applied to the current wavefront. The *RMS* switch is ignored if set, and the PSD parameters in this case are as described above. Note that intensity is the square of amplitude, so *AMPLITUDE=0.9* will result in a maximum intensity transmission of 0.81.

FILE=*string* (IDL)
FILE=*string* (Python)
'FILE', *string* (Matlab)

(Optional) This keyword, if set to a value, specifies that the error map generated by this routine will be saved to a FITS file with the provided name (exclude the filename extension). Subsequent calls to this routine with the same filename will cause the error map in the file to be read in and applied to the wavefront, rather than a new map being generated. In this instance the other parameters will be ignored, except when the *AMPLITUDE* keyword is set to a value, in which case the map is assumed to be an amplitude map and will have a maximum specified by the *AMPLITUDE* keyword value but the same RMS value about the mean as before.

Note that there are no checks made to ensure that the parameter values used to create the map in a file are the same ones given in subsequent calls.

INCLINATION=*angle* (IDL)
INCLINATION=*angle* (Python)
'INCLINATION', *angle* (Matlab)

(Optional) This keyword specifies the inclination in degrees from the Y axis in the Y-Z plane that the surface described by the PSD makes relative to the direction of the incident beam. An inclination of zero (the default) indicates that the surface is perpendicular to the propagation direction. The sign of the inclination is not important. Specifying the inclination alters the projection of the surface onto the wavefront. When the *ROTATION* angle is also specified, the surface is first inclined and then rotated.

MAP=*variable* (IDL)

(Optional) The wavefront, surface, or amplitude error map created by this routine will be returned in the variable assigned to this optional keyword.

MAX_FREQUENCY=*variable* (IDL)
MAX_FREQUENCY=*variable* (Python)
'MAX_FREQUENCY', *variable* (Matlab)

(Optional) Maximum spatial frequency (cycles/meter) in the generated map. This can be used to prevent high spatial frequency components from generating aliasing errors when a map is resampled.

/MIRROR (IDL)
MIRROR=*True* or *False* (Python)
'MIRROR' (Matlab)

(Optional) This switch, if set, indicates that the PSD specifies the surface, not wavefront, error of a mirror. The wavefront error is twice the surface error due to reflection.

/NO_APPLY (IDL)
NO_APPLY=*True* or *False* (Python)
'NO_APPLY' (Matlab)

(Optional) This switch, if set, will cause a map to be generated but not applied to the wavefront. This is useful if you wish to obtain a map that you will modify yourself.

/RMS (IDL)
RMS=*True* or *False* (Python)
'RMS' (Matlab)

(Optional) This switch, if set, indicates that the generated error map is to be renormalized to have an RMS value specified by the *amp* parameter.

ROTATION=*angle* (IDL)
ROTATION=*angle* (Python)
'ROTATION', *angle* (Matlab)

(Optional) This keyword specifies the counter-clockwise rotation in degrees of the surface in the X-Y plane. This only has an effect if the *INCLINATION* keyword is also defined.

/TPF (IDL)
TPF=*True* or *False* (Python)
'TPF' (Matlab)

(Optional) This switch, if set, indicates that the Terrestrial Planet Finder (TPF) PSD profile specification is to be used.

Examples

Apply a PSD-defined wavefront error map to the current wavefront assuming the TPF PSD form specification and return the generated map in the variable `errormap`:

IDL:

```
amp = 9.6e-19
b = 4.0
c = 3.0
prop_psd_errormap, wavefront, amp, b, c, /TPF, MAP=errormap
```

Python:

```
amp = 9.6e-19
b = 4.0
c = 3.0
errormap = proper.prop_psd_errormap( wavefront, amp, b, c, TPF=True )
```

Matlab:
```
amp = 9.6e-19;
b = 4.0;
c = 3.0;
[wavefront, errormap] = prop_psd_errormap( wavefront, amp, b, c, 'TPF' );
```

Multiply the current wavefront by a PSD-defined amplitude error map with a maximum value of 0.9 and an RMS of 0.001 about the mean to the current wavefront:

IDL:
```
amp = 0.001
b = 1.0
c = 4.0
prop_psd_errormap, wavefront, amp, b, c, AMPLITUDE=0.9
```

Python:
```
amp = 0.001
b = 1.0
c = 4.0
proper.prop_psd_errormap( wavefront, amp, b, c, AMPLITUDE=0.9 )
```

Matlab:
```
amp = 0.001;
b = 1.0;
c = 4.0;
wavefront = prop_psd_errormap( wavefront, amp, b, c, 'AMPLITUDE', 0.9 );
```

Apply a PSD-defined mirror surface error map, saving the result to the file ‘primary’ (the error map will be read from the file on subsequent calls if the same filename is specified):

IDL:
```
prop_psd_errormap, wavefront, amp, b, c, FILE=’primary’
```

Python:
```
proper.prop_psd_errormap( wavefront, amp, b, c, FILE=’primary’ )
```

Matlab:
```
wavefront = prop_psd_errormap( wavefront, amp, b, c, 'FILE', ’primary’ );
```

See Also

PROP_ERRORMAP, PROP_ZERNIKES

PROP_RADIUS

Return a 2D array in which the value of each element corresponds to the distance of that element from the center of the current wavefront. By default, the distance is in meters unless the *NORM* switch is set, in which case it is normalized to the current radius of the beam as determined by the Gaussian tracer beam. The center of the wavefront array is defined to be at the center pixel of the array.

Syntax

IDL: *radius* = **prop_radius**(*wavestruct [, /NORM]*)

Python: *radius* = **proper.prop_radius**(*wavestruct [, NORM=True/False]*)

Matlab: *radius* = **proper.prop_radius**(*wavestruct [, 'NORM']*);

Returns

radius
(Required) A 2D array in which the value of each element corresponds to the distance of that element from the center of the current wavefront.

Arguments

wavestruct
(Required) The current wavefront structure.

Keywords and Switches

/NORM (IDL)
NORM=*True* or *False* (Python)
'NORM' (Matlab)
(Optional) Indicates that the returned array is to contain distances normalized to the beam radius. This assumes that the radius of the pilot tracer beam accurately reflects the size of the actual beam in the wavefront array, which will not be true in the case of significant aberrations.

Examples

Multiply the wavefront amplitude by a 2D Gaussian with σ = 0.01 meters:

IDL:
```
r = prop_radius( wavefront )
gauss_amp = exp(-0.5*(r/0.01)^2)
prop_multiply, wavefront, gauss_amp
```

Python:
```
r = proper.prop_radius( wavefront )
gauss_amp = np.exp(-0.5*(r/0.01)**2)
proper.prop_multiply( wavefront, gauss_amp )
```

Matlab:

```
r = prop_radius( wavefront );
gauss_amp = exp(-0.5 .* (r./0.01).^2);
wavefront = prop_multiply( wavefront, gauss_amp );
```

See Also

PROP_GET_BEAMRADIUS

PROP_READMAP

Read an error map from a FITS image file. Note that the map is not applied to the current wavefront (use **PROP_ERRORMAP** to read in a map and do that). The map will be interpolated as necessary to match the sampling of the current wavefront.

Syntax

IDL: **prop_readmap**, *wavestruct, filename, map [, xshift, yshift] [,SAMPLING=value] [, XC_MAP=value, YC_MAP=value]*

Python: *map* = **proper.prop_readmap**(*wavestruct, filename[, xshift, yshift] [,SAMPLING=value] [, XC_MAP=value, YC_MAP=value]*)

Matlab: *map* = **prop_readmap**(*wavestruct, filename[, 'XSHIFT', xshift] ['YSHIFT', yshift] [, 'SAMPLING', value] [, 'XC_MAP', value] [, 'YC_MAP', value]*);

Returns

map (Python)
map (Matlab)
(Required) The error map.

Arguments

wavestruct
(Required) The current wavefront structure.

filename
(Required) The name of a FITS image file containing the error map.

map (IDL)
(Required) A variable in which the error map is returned.

xshift, yshift (IDL)
xshift, yshift (Python)
(Optional) amounts to shift the map in meters in the wavefront coordinate system.

Keywords and Switches

SAMPLING=*value* (IDL)
SAMPLING=*value* (Python)
'SAMPLING', *value* (Matlab)
(Optional) Keyword that specifies the sampling of the map in meters. This will override any sampling specified in the data file header. This must be specified if no sampling is specified in the header using the PIXSIZE keyword. Note that the sampling is not returned by this keyword. If the header value

RADPIX is defined (the radius of the beam in the map in pixels), then that value will override any other sampling specifiers, including the *SAMPLING* keyword.

XC_MAP=*value* (IDL)
YC_MAP=*value*
XC_MAP=*value* (Python)
YC_MAP=*value*
'XC_MAP', *value* (Matlab)
'YC_MAP', *value*

(Optional) Keyword values that specify the center of the wavefront in the map (prior to the map being shifted if *xshift* and *yshift* are given) in pixels. The center of the first pixel is (0.0, 0.0). By default, the map is assumed to be centered at (*n/2, n/2)*.

Examples

Read an error map from the file 'surface1.fits', which has a sampling of 1 mm:

IDL:
```
prop_readmap, wavefront, 'surface1.fits', map, SAMPLING=0.001
```

Python:
```
map = proper.prop_readmap( wavefront, 'surface1.fits', SAMPLING=0.001 )
```

Matlab:
```
map = prop_readmap( wavefront, 'surface1.fits', 'SAMPLING', 0.001 );
```

See Also

PROP_ERRORMAP, PROP_PSD_ERRORMAP

PROP_RECTANGLE

Return a 2D image containing a filled, antialiased rectangle. This routine does not modify the wavefront structure. This function is intended for use by other PROPER routines. Users should use **PROP_RECTANGULAR_APERTURE** and **PROP_RECTANGULAR _OBSCURATION** if possible.

NOTE: Prior to PROPER version 3.0, the manual incorrectly stated that when using the *NORM* option, the X and Y widths of the rectangle were relative to the beam **diameter**; they were (and are now) actually relative to the beam **radius.**

Syntax

IDL: *image* = **prop_rectangle**(*wavestruct, xwidth, ywidth [, xoff, yoff] [, /DARK] [, /NORM] [, ROTATION=angle]*)

Python: *image* = **proper.prop_rectangle**(*wavestruct, xwidth, ywidth [, xoff, yoff] [, DARK=True/False] [, NORM=True/False] [, ROTATION=angle]*)

Matlab: *image* = **prop_rectangle**(*wavestruct, xwidth, ywidth [, 'XC', xoff] ['YC', yoff] [, 'DARK'] [, 'NORM', 0 or 1] [, 'ROTATION', angle*);

Returns

image
(Required) An image the same size as the current wavefront containing an antialised, filled rectangle.

Arguments

wavestruct
(Required) The current wavefront structure (used to obtain sampling information).

xwidth, ywidth
(Required) Width of obscuration along X and Y image axes in meters, or if the *NORM* switch is set, the widths in terms of the fraction of the beam *radius* at the current surface.

xoff, yoff (IDL)
xoff, yoff (Python)
(Optional) X and Y axis offsets of the rectangle center from the center of the wavefront grid. These are specified in meters unless the *NORM* switch is set, in which case they are in fractions of the current beam *radius*. By default, the rectangle is centered at the center of the wavefront grid.

Keywords and Switches

/DARK (IDL)
DARK=*True* or *False* (Python)
'DARK' (Matlab)

(Optional) Switch indicating that the interior of the rectangle should be set to 0.0 and the exterior to 1.0. By default, it is 1.0 interior and 0.0 exterior.

/NORM (IDL)
NORM=*True* or *False* (Python)
'NORM', 0 or 1 (Matlab)

(Optional) Switch specifying that the X and Y widths and X and Y centers are specified as fractions of the beam *radius*. By default, they are in meters.

ROTATION=*angle* (IDL)
ROTATION=*angle* (Python)
'ROTATION', *angle* (Matlab)

(Optional) Specifies the degrees counter-clockwise to rotate the rectangle around its center.

'XC', xoff
'YC', yoff (Matlab)

(Optional) X and Y axis offsets of the rectangle center from the center of the wavefront grid. These are specified in meters unless the *NORM* switch is set, in which case they are in fractions of the current beam *radius*. By default, the rectangle is centered at the center of the wavefront grid.

Examples

Create an image with a 1.2 m by 0.5 m rectangle in it offset from the center by 0.2, 0.3 in X, Y:

IDL:
```
rect = prop_rectangle( wavefront, 1.2, 0.5, 0.2, 0.3 )
```

Python:
```
rect = proper.prop_rectangle( wavefront, 1.2, 0.5, 0.2, 0.3 )
```

Matlab:
```
rect = prop_rectangle( wavefront, 1.2, 0.5, 'XC', 0.2, 'YC', 0.3 );
```

See Also

PROP_CIRCULAR_APERTURE, PROP_CIRCULAR_OBSCURATION, PROP_ELLIPSE, PROP_ELLIPTICAL_APERTURE, PROP_ELLIPTICAL_OBSCURATION, PROP_RECTANGULAR_APERTURE, PROP_RECTANGULAR_OBSCURATION

PROP_RECTANGULAR_APERTURE

Multiply the current wavefront by a rectangular aperture (clear inside, dark outside). The rectangle is antialiased (the value of an edge pixel varies between 0.0 and 1.0 in proportion to the amount of a pixel covered by the rectangle).

NOTE: Prior to PROPER version 3.0, the manual incorrectly stated that when using the *NORM* option, the X and Y widths of the rectangle were relative to the beam **diameter**; they were (and are now) actually relative to the beam **radius.**

Syntax

IDL: **prop_rectangular_aperture**, *wavestruct, xwidth, ywidth* [, *xoff, yoff*] [, */NORM*] [, *ROTATION=angle*]

Python: **proper.prop_rectangular_aperture**(*wavestruct, xwidth, ywidth* [, *xoff, yoff*] [, *NORM=True/False*] [, *ROTATION=angle*])

Matlab: *wavestruct_out* = **prop_rectangular_aperture**(*wavestruct_in, xwidth, ywidth [, 'XC', xoff] ['YC', yoff] [, 'NORM'] [, 'ROTATION', angle*);

Returns

wavestruct_out (Matlab)
(Required) The modified wavefront structure.

Arguments

wavestruct (IDL)
wavestruct (Python)
wavestruct_in (Matlab)
(Required) The current wavefront structure.

xwidth, ywidth
(Required) Width of obscuration along X and Y image axes in meters, or if the *NORM* switch is set, the widths in terms of the fraction of the beam *radius* at the current surface.

xoff, yoff (IDL)
xoff, yoff (Python)
(Optional) X and Y axis offsets of the rectangle center from the center of the wavefront grid. These are specified in meters unless the *NORM* switch is set, in which case they are in fractions of the current beam *radius*. By default, the rectangle is centered at the center of the wavefront grid.

Keywords and Switches

/NORM (IDL)
NORM=*True* or *False* (Python)
'NORM' (Matlab)
(Optional) Switch specifying that the X and Y widths and X and Y centers are specified as fractions of the beam *radius*. By default, they are in meters.

ROTATION=*angle* (IDL)
ROTATION=*angle* (Python)
'ROTATION', *angle* (Matlab)
(Optional) Specifies the degrees counter-clockwise to rotate the rectangle around its center.

'XC', xoff
'YC', yoff (Matlab)
(Optional) X and Y axis offsets of the rectangle center from the center of the wavefront grid. These are specified in meters unless the *NORM* switch is set, in which case they are in fractions of the current beam *radius*. By default, the rectangle is centered at the center of the wavefront grid.

Examples

Multiply the wavefront by a rectangle of 10.0 mm by 5 mm and centered +1 mm from the wavefront center along the X axis:

```
IDL:    prop_rectangular_aperture, wave, 0.010, 0.005, 0.001, 0.0

Python: proper.prop_rectangular_aperture( wave, 0.010, 0.005, 0.001, 0.0 )

Matlab: wave = prop_rectangular_aperture( wave, 0.010, 0.005, 'XC', 0.001, 'YC', 0.0 );
```

See Also

PROP_CIRCULAR_APERTURE, PROP_CIRCULAR_OBSCURATION, PROP_ELLIPSE, PROP_ELLIPTICAL_OBSCURATION, PROP_RECTANGULAR_APERTURE, PROP_RECTANGULAR_OBSCURATION

PROP_RECTANGULAR_OBSCURATION

Multiply the current wavefront by a rectangular obscuration (clear outside, dark inside). The rectangle is antialiased (the value of an edge pixel varies between 0.0 and 1.0 in proportion to the amount of a pixel covered by the rectangle).

NOTE: Prior to PROPER version 3.0, the manual incorrectly stated that when using the *NORM* option, the X and Y widths of the rectangle were relative to the beam **diameter**; they were (and still are now) actually relative to the beam **radius.**

Syntax

IDL: **prop_rectangular_obscuration**, *wavestruct, xwidth, ywidth* [, *xoff, yoff*] [, */NORM*] [, *ROTATION=angle*]

Python: **proper.prop_rectangular_obscuration**(*wavestruct, xwidth, ywidth* [, *xoff, yoff*] [, *NORM=True/False*] [, *ROTATION=angle*])

Matlab: *wavestruct_out* = **prop_rectangular_obscuration**(*wavestruct_in, xwidth, ywidth [, 'XC', xoff] ['YC', yoff] [, 'NORM'] [, 'ROTATION', angle*);

Returns

wavestruct_out (Matlab)

(Required) The modified wavefront structure.

Arguments

wavestruct (IDL)
wavestruct (Python)
wavestruct_in (Matlab)

(Required) The current wavefront structure.

xwidth, ywidth

(Required) Width of obscuration along X and Y image axes in meters, or if the *NORM* switch is set, the widths in terms of the fraction of the beam *radius* at the current surface.

xoff, yoff (IDL)
xoff, yoff (Python)

(Optional) X and Y axis offsets of the rectangle center from the center of the wavefront grid. These are specified in meters unless the *NORM* switch is set, in which case they are in fractions of the current beam *radius*. By default, the rectangle is centered at the center of the wavefront grid.

Keywords and Switches

/NORM (IDL)
NORM=*True* or *False* (Python)
'NORM' (Matlab)
(Optional) Switch specifying that the X and Y widths and X and Y centers are specified as fractions of the beam *radius*. By default, they are in meters.

ROTATION=*angle* (IDL)
ROTATION=*angle* (Python)
'ROTATION', *angle* (Matlab)
(Optional) Specifies the degrees counter-clockwise to rotate the rectangle around its center.

'XC', xoff
'YC', yoff (Matlab)
(Optional) X and Y axis offsets of the rectangle center from the center of the wavefront grid. These are specified in meters unless the *NORM* switch is set, in which case they are in fractions of the current beam *radius*. By default, the rectangle is centered at the center of the wavefront grid.

Examples

Multiply the wavefront by a rectangular obscuration with a width of 5 mm and height of 3 mm:

IDL:
```
prop_rectangular_obscuration, wavefront, 0.005, 0.003
```

Python:
```
proper.prop_rectangular_obscuration( wavefront, 0.005, 0.003 )
```

Matlab:
```
wavefront = prop_rectangular_obscuration( wavefront, 0.005, 0.003 );
```

See Also

PROP_CIRCULAR_APERTURE, PROP_CIRCULAR_OBSCURATION,
PROP_ELLIPSE, PROP_ELLIPTICAL_OBSCURATION,
PROP_RECTANGULAR_APERTURE,
PROP_RECTANGULAR_OBSCURATION

PROP_RESAMPLEMAP

Resample a map using cubic convolution interpolation onto a grid with the same size and sampling as the current wavefront. Optionally, shift the map. In IDL, the resampled map replaces the input map. *Note that the map is not applied to the wavefront.* The input map must be defined over a large enough region that when resampled and shifted there will be no extrapolated values inside the beam.

Syntax

IDL: **prop_resamplemap**, *wavestruct, map, sampling [, xc, yc [, xshift, yshift]]*

Python: *new_map =* **proper.prop_resamplemap**(*wavestruct, old_map, sampling [, xc, yc [, xshift, yshift]]*)

Matlab: *new_map =* **prop_resamplemap**(*wavestruct, old_map, sampling [, xc, yc [, xshift, yshift]]*);

Returns

new_map (Python)
new_map (Matlab)
(Optional) The resampled map.

Arguments

wavestruct
(Required) The current wavefront structure. Used to obtain the current wavefront sampling.

map (IDL)
old_map (Python)
old_map (Matlab)
(Required) The variable containing the map to be resampled. In IDL this will be replaced by the resampled map. The input map must be defined over a large enough region that when resampled and shifted there will be no extrapolated values inside the beam.

sampling
(Required) The spacing between points in the input map in meters

xc, yc
(Optional) The pixel coordinates in the input map where the wavefront is centered (the 1st pixel is centered at (0.0,0.0)). The default assumes it is centered on the central pixel.

xshift, yshift
(Optional) The amounts to shift the map in meters in the wavefront coordinate system

See Also

PROP_ERRORMAP, PROP_READMAP

PROP_ROTATE

Rotate and shift a real-valued or complex-valued image via interpolation.

Syntax

IDL: *new_image* = **prop_rotate(** *old_image, angle*
[, CUBIC=value] [, MISSING=value]
[, XC=value, YC=value] [, XSHIFT=value, YSHIFT=value])

Python: *new_image* = **proper.prop_rotate(** *old_image, angle*
[, CUBIC=True/False] [, MISSING=value]
[, XC=value, YC=value] [, XSHIFT=value, YSHIFT=value])

Matlab: *new_image* = **prop_rotate(** *old_image, angle*
[, 'CUBIC'] ['METH', string] [, 'MISSING', value]
[, 'XC', value] [, 'YC', value] [, 'XSHIFT', value] [, 'YSHIFT', value]);

Returns

new_image

(Required) Rotated, and possibly shifted, two-dimensional image.

Arguments

old_image

(Required) Two-dimensional, real-valued image to be rotated and/or shifted.

angle

(Required) The angle to rotate the image, measured counter-clockwise in degrees.

Keywords and Switches

CUBIC=*value* (IDL)

(Optional) Specifies that cubic convolution interpolation is to be used (bilinear is the default). The value of this keyword affects the shape of the convolution kernel. It is usually recommended to be set to -0.5 (see the *INTERPOLATE* function description in the IDL Reference Manual).

CUBIC=*True* or *False* (Python)

(Optional) Specifies that Python's *map_coordinates* function with order=3 be used for interpolation (bilinear is the default).

'CUBIC' (Matlab)

(Optional) Specifies that Matlab's *interp2* function with the 'CUBIC' (cubic convolution) method be used for interpolation (bilinear is the default). This is the same as using `'METH','cubic'`.

'METH', *value* (Matlab)

(Optional) String specifying the type of interpolation that Matlab's *interp2* function will use (options are 'cubic', 'nearest', 'linear', 'spline'; the default is 'linear' for bilinear interpolation).

MISSING=*value* (IDL)
MISSING=*value* (Python)
'MISSING', *value* (Matlab)

(Optional) Specifies the value of extrapolated data points.

XC=*value*, (IDL)
YC=*value*
XC=*value*, (Python)
YC=*value*
'XC', *value*, (Matlab)
'YC', *value*

(Optional) Center of rotation in image pixels; (0,0) is the center of the 1st pixel. If not specified, the center of rotation is assumed to be (n/2,n/2).

XSHIFT=*value*, (IDL)
YSHIFT=*value*
XSHIFT=*value*, (Python)
YSHIFT=*value*
XSHIFT=*value*, (Matlab)
YSHIFT=*value*

(Optional) Amount in pixels to shift the image after rotation.

See Also

PROP_RESAMPLEMAP

PROP_ROUNDED_RECTANGLE

Return a two-dimensional image, with the same dimensions as the current wavefront array, containing a rectangular mask (0 outside, 1 inside) with rounded corners. This routine was created for modeling the TPF-C primary mirror. Note that the mask is not applied to the current wavefront.

Syntax

IDL: *array* = **prop_rounded_rectangle(** *wavestruct, corner_radius, width, height [, xc, yc]*)

Python: *array* = **proper.prop_rounded_rectangle(** *wavestruct, corner_radius, width, height [, xc, yc]*)

Matlab: *array* = **prop_rounded_rectangle(** *wavestruct, corner_radius, width, height*
[, 'XC', xc] ['YC', yc]);

Returns

array
(Required) The 2D array containing the rounded rectangle.

Arguments

wavestruct
(Required) The current wavefront structure, used to obtain sampling and grid size information.

corner_radius
(Required) The radius in meters of a rounded corner (90º section of a circle).

width, height
(Required) The width and height in meters of the rectangular mask.

xc, yc (IDL)
xc, yc (Python)
(Optional) The offset in meters of the center of the rectangle from the center of the optical axis.

Keywords and Switches

'XC', *xc* (Matlab)
'YC', *yc*
(Optional) The offset in meters of the center of the rectangle from the center of the optical axis.

See Also

PROP_RECTANGLE

PROP_RUN

Execute a prescription containing PROPER procedure calls. For running a prescription in parallel, see **PROP_RUN_MULTI**.

Syntax

IDL: **prop_run**, *prescription*, *result, wavelength, gridsize [, sampling_m]*
[, PASSVALUE=value] [, /PHASE_OFFSET] [, /PRINT_INTENSITY]
[, /QUIET] [, /TABLE] [, /VERBOSE]

Python: (*result, sampling_m*) = **proper.prop_run**(*prescription*, *wavelength, gridsize*
[, PASSVALUE=value] [, PHASE_OFFSET=True/False]
[, PRINT_INTENSITY=True/False] [, QUIET=True/False]
[, TABLE=True/False] [, VERBOSE=True/False])

Matlab: *result = - OR -*
[*result, sampling_m*] =
prop_run(*prescription*, *wavelength, gridsize [, 'PASSVALUE', value]*
[, 'PHASE_OFFSET'] [, 'PRINT_INTENSITY'] [, 'QUIET']
[, 'TABLE'] [, 'VERBOSE']);

Returns

result (Python)
result (Matlab)

(Required) A variable to hold the result of the propagation.

sampling_m (Python)
sampling_m (Matlab)

(Required) A variable in which the sampling of the result in meters is returned (assuming the user has set the corresponding parameter in the prescription).

Arguments

prescription

(Required) The name of the procedure containing the calls to PROPER routines. The file containing this procedure must have the same rootname and a `.pro` suffix for IDL, `.py` suffix for Python, or `.m` suffix for Matlab.

result (IDL)

(Required) A variable to contain the result of the propagation.

wavelength

(Required) Either the wavelength in ***microns*** at which to propagate, or the name of a text file containing a list of wavelength (microns) and weight pairs, one pair per line. In the latter case, the prescription is run once for each wavelength in the list and the result is the weighted sum of the individual results.

gridsize

(Required) The size of the two-dimensional grid that describes the wavefront (gridsize by gridsize pixels). This must be a factor of 2 (e.g. 1024, 2048, 4096).

sampling_m (IDL)

(Optional) Variable in which the sampling of the result in meters is returned (assuming the user has set the corresponding parameter in the prescription).

Keywords and Switches

PASSVALUE=*value* (IDL)
PASSVALUE=*value* (Python)
'PASSVALUE', *value* (Matlab)

(Optional) This keyword is set to variable (which could be a structure containing many variables; in Matlab it needs to be a structure) that is passed to the prescription for use within the prescription. This is useful for passing parameters in addition to the required ones. In IDL the prescription can change and return the value in the same variable. In Python the value must be a dictionary.

/PHASE_OFFSET (IDL)
PHASE_OFFSET=*True* or *False* (Python)
'PHASE_OFFSET' (Matlab)

(Optional) If set, the wavefront phase at all points will be incremented by the propagation distance. By default, there is no offset.

/PRINT_INTENSITY (IDL)
PRINT_INTENSITY=*True* or *False* (Python)
'PRINT_INTENSITY' (Matlab)

(Optional) If set, the total intensity of the wavefront will be printed out after each propagation.

/QUIET (IDL)
QUIET=*True* or *False* (Python)
'QUIET' (Matlab)

(Optional) If set, intermediate messages will be suppressed.

/TABLE (IDL)
TABLE=*True* or *False* (Python)
'TABLE' (Matlab)

(Optional) If set, a table of the beam dimensions and sampling at each surface will be displayed on the terminal.

/VERBOSE (IDL)
VERBOSE=*True* or *False* (Python)
'VERBOSE' (Matlab)

(Optional) If set, status messages from the PROP propagation routines will be printed.

Examples

Run the prescription in ‘telescope.pro’ at a wavelength of 0.5 μm using a wavefront gridsize of 1024 by 1024:

IDL:
```
prop_run, ’telescope’, psf, 0.5, 1024, sampling
```

Python:
```
(psf, sampling) = proper.prop_run( ’telescope’, 0.5, 1024 )
```

Matlab:
```
[psf, sampling] = prop_run( ’telescope’, 0.5, 1024 );
```

Run the prescription in ‘telescope.pro’ at a wavelength of 0.5 μm using a wavefront gridsize of 1024 by 1024, returning the final sampling in the variable *scale* and passing the focal length and diameter via the *PASSVALUE* keyword:

IDL:
```
prop_run, ’telescope’, psf, 0.5, 1024, scale, PASSVALUE={fl:10.0, diam:2.0}
```

Python:
```
(psf, scale) = proper.prop_run( ’telescope’, 0.5, 1024,
                PASSVALUE={'fl':10.0, 'diam':2.0} )
```

Matlab:
```
pars.fl = 10.0;
pars.diam = 2.0;
[psf, scale] = prop_run( ’telescope’, 0.5, 1024, 'PASSVALUE', pars );
```

PROP_RUN_MULTI

Execute multiple instances in parallel of a prescription containing PROPER procedure calls.

Syntax

IDL: **prop_run_multi**, *prescription*, *result, wavelength, gridsize [, sampling_m]*
[, /NO_SHARED_MEMORY] [, PASSVALUE=value] [, /PHASE_OFFSET]
[, /QUIET] [, /KEEP_THREADS]

Python: (*result, sampling_m*) = **proper.prop_run_multi**(*prescription*, *wavelength, gridsize*
[, NCPUS=value] [, PASSVALUE=value] [, PHASE_OFFSET=True/False]
[, QUIET=True/False])

Matlab: *result = - OR -*
[*result, sampling_m*] =
prop_run_multi(*prescription*, *wavelength, gridsize [, 'PASSVALUE', value]*
[, 'PHASE_OFFSET'] [, 'QUIET'] [, 'KEEP_THREADS']);

Returns

result (Python)
result (Matlab)

(Required) A variable to hold the result of the propagation. This will be a three-dimensional array where the 3rd dimension is equal to the number of wavelengths and/or optional parameter (*PASSVALUE*) entries. This is part of the (*result, sampling_m*) tuple.

sampling_m (Python)
sampling_m (Matlab)

(Required) A variable in which the sampling of the result in meters is returned (assuming the user has set the corresponding parameter in the prescription). This is an array of size equal to the number of wavelengths and/or optional parameter (*PASSVALUE*) entries. This is part of the (*result, sampling_m*) tuple.

Arguments

prescription

(Required) The name of the procedure containing the calls to PROPER routines. The file containing this procedure must have the same rootname and a `.pro` suffix for IDL, `.py` suffix for Python, or `.m` suffix for Matlab.

result (IDL)

(Required) A variable to hold the result of the propagation. This will be a three-dimensional array where the 3rd dimension is equal to the number of wavelengths and/or optional parameter (PASSVALUE) entries.

wavelength

(Required) The wavelength in ***microns*** at which to propagate. This may be an array of values, in which case the prescription will be run in parallel for each wavelength. If it is a single value, then the PASSVALUE parameter should contain multiple entries.

gridsize

(Required) The size of the two-dimensional grid that describes the wavefront (gridsize by gridsize pixels). This must be a factor of 2 (e.g. 1024, 2048, 4096, etc.).

sampling_m (IDL)

(Optional) A variable in which the sampling of the result in meters is returned (assuming the user has set the corresponding parameter in the prescription). This is an array of size equal to the number of wavelengths and/or optional parameter (*PASSVALUE*) entries.

Keywords and Switches

/KEEP_THREADS (IDL)
'KEEP_THREADS' (Matlab)

(Optional) If set, the parallel processes will not be destroyed after the call is finished (by default, they are). On subsequent calls to PROP_RUN_MULTI with this set, the previously-created processes will be used instead, saving substantial time when remote license servers are used. Active threads can be destroyed by 1) calling PROP_FREE_THREADS; 2) calling PROP_RUN_MULTI without KEEP_THREADS (old processes are deleted and new ones created); or 3) exiting IDL or Matlab. For more information, see ***Running multiple instances of a prescription in parallel with PROP_RUN_MULTI.***

NCPUS=*value* (Python)

(Optional) Integer that sets the maximum number of CPUs that can be used. This should be equal to or greater than the number of entries in *wavelength* or *PASSVALUE*.

/NO_SHARED_MEMORY (IDL)

As of PROPER Version 3.0, the default method for transferring information from the child process is via shared memory. Previously it was done via a variable, which took longer due to the need to make an additional copy. There have been instances where in other programs the author has had the machine hang when the shared memory was being freed. Setting this switch uses the old slow-but-safe method.

PASSVALUE=*value*

(Optional) This keyword is set to variable (which could be a structure containing many variables) that is passed to the prescription for use within the prescription. This is useful for passing parameters in addition to the required ones. The prescription CANNOT return the value in the same variable (this is allowed with IDL in **PROP_RUN**). If this is an array, the prescription will be run in parallel for each entry. If both this and *wavelength* are arrays, they must have the same number of entries and each *PASSVALUE* entry will be used for the corresponding *wavelength* entry. In Python the value must be a dictionary or list of dictionaries (one for each process running in parallel).

/PHASE_OFFSET (IDL)
PHASE_OFFSET=*True* or *False* (Python)
'PHASE_OFFSET' (Matlab)

If set, the wavefront phase at all points will be incremented by the propagation distance. By default, there is no offset.

/QUIET (IDL)
QUIET=*True* or *False* (Python)
'QUIET' (Matlab)

If set, intermediate messages will be suppressed.

PROP_SET_ANTIALIASING

Set the number of samples, *n*, along each dimension (*n* by *n*) that a pixel along the edge of a shape (circle, ellipse, rectangle, polygon) will be subdivided to determine the fractional coverage of the pixel by the shape (antialiasing). The value of that pixel will be the fractional coverage of that pixel, varying from 0.0 to 1.0. The default is *n* = 11. Note that *n* must be an odd-valued whole number. Increasing the value will improve the accuracy of the result at the expense of additional time.

NOTE: The procedure should be called after **PROP_BEGIN** but before any other PROPER shape-drawing routines. The default antialiasing factor is set in **PROP_BEGIN**.

Syntax

IDL: **prop_set_antialiasing,** *n*

Python: **proper.prop_set_antialiasing(** *n*)

Matlab: **prop_set_antialiasing(** *n*);

Arguments

n

The edge pixel subsampling factor along each dimension. This must be an odd-valued whole number.

See Also

PROP_CIRCULAR_APERTURE, PROP_CIRCULAR_OBSCURATION, PROP_ELLIPSE, PROP_ELLIPTICAL_APERTURE, PROP_ELLIPTICAL_OBSCURATION, PROP_HEX_WAVEFRONT, PROP_IRREGULAR_POLYGON, PROP_POLYGON, PROP_RECTANGLE, PROP_RECTANGULAR_APERTURE, PROP_RECTANGULAR_OBSCURATION, PROP_ROUNDED_RECTANGLE

PROP_SHIFT_CENTER

Shift an *n* by *n* array by (*n*/2, *n*/2) in X and Y (*n* must be even). This effectively shifts the center of an image to the lower-left corner of the array.

Syntax

IDL: *result* = **prop_shift_center**(*image*)

Python: *result* = **proper.prop_shift_center**(*image*)

Matlab: *result* = **prop_shift_center**(*image*);

Returns

result

The two-dimensional shifted array.

Arguments

image

The two-dimensional array to be shifted.

PROP_STATE

Save the current propagation state to a file for the current wavelength, if one does not already exist. If one does exist, read it in and use it to define the current wavefront. See the section "Save States" for more information. This cannot be used with **PROP_RUN_MULTI**.

Syntax

IDL: **prop_state**, *wavestruct*

Python: **proper.prop_state**(*wavestruct*)

Matlab: *wavestruct_out* = **prop_state**(*wavestruct_in*);

Returns

wavestruct_out (Matlab)
The new wavefront structure (usually the same variable as the input wavefront structure).

Arguments

wavestruct (IDL)
wavestruct (Python)
wavestruct_in (Matlab)
The current wavefront structure.

See Also

PROP_END_SAVESTATE, PROP_INIT_SAVESTATE,
PROP_IS_STATESAVED

PROP_USE_FFTI (IDL, Python)

In IDL, this enables or disables the use of the Intel Math Kernel Library FFT routine instead of the built-in one for propagation. The changes remain in effect over multiple IDL sessions until otherwise altered by calling this routine.

In Python, this checks that the Intel Math Kernel Library has been installed and is available for use, then enables it.

NOTE: Whenever you install a new version of PROPER in Python, you will need to again activate the use of the Intel FFT with this routine.

Syntax

IDL: **prop_use_ffti** *[, /DISABLE]*

Python: **proper.prop_use_ffti(** *[DISABLE=True] [, MKL_DIR=directory]*)

Keywords and Switches

/DISABLE (IDL)
DISABLE=*True* (Python)
Disables the Intel MKL routines so that the built-in FFT (Numpy for Python) is used instead.

MKL_DIR=*directory* (Python)
Specifies the directory containing the MKL libraries. If not given, PROPER will look in the following directories:

Linux, Unix, MacOS: `/opt/intel/mkl/lib/intel64`
Windows: `C:/Program Files(x86)/IntelSWTools/compilers_and_libraries/windows/mkl/lib/intel64`

See Also

PROP_COMPILE_FFTI

PROP_USE_FFTW

In IDL, this enables or disables the use of the FFTW library's FFT routine instead of the built-in IDL one for propagation. The changes remain in effect over multiple IDL sessions until otherwise altered by calling this routine. If the Intel Math Kernel Library FFT is also enabled, then it will be used instead of FFTW.

In Python, this checks that the FFTW library has been installed and is available for use, then enables its use. The pyFFTW package is used.

NOTE: Whenever you install a new version of PROPER in Python, you will need to again activate the use of the Intel FFT with this routine.

Syntax

IDL: **prop_use_fftw** *[, /DISABLE]*

Python: **proper.prop_use_fftw**(*[DISABLE=True]*)

Keywords and Switches

/DISABLE (IDL)
DISABLE=*True* (Python)
Disables the FFTW routines so that the built-in FFT (Numpy for Python) is used instead.

See Also

PROP_COMPILE_FFTW, PROP_FFTW_WISDOM

PROP_WRITEMAP

Write an error map (surface, wavefront, or amplitude) to a FITS image file. Such a map can be read later by **PROP_ERRORMAP**.

Syntax

IDL: **prop_writemap**, *map, filename [, /AMPLITUDE] [, /MIRROR] [, RADIUS_PIX=value] [, SAMPLING=value] [, /WAVEFRONT]*

Python: **proper.prop_writemap**(*map, filename [, AMPLITUDE=True/False] [, MIRROR=True/False] [, RADIUS_PIX=value] [, SAMPLING=value] [, WAVEFRONT=True/False]*)

Matlab: **prop_writemap**(*map, filename [, 'AMPLITUDE'] [, 'MIRROR'] [, 'RADIUS_PIX', value] [, 'SAMPLING', value] [, 'WAVEFRONT']*);

Arguments

map

(Required) The error map to write out. Must be a two-dimensional real-valued array (not complex valued).

filename

(Required) String name of the FITS file to write.

Keywords and Switches

/AMPLITUDE (IDL)
AMPLITUDE=*True* or *False* (Python)
'AMPLITUDE' (Matlab)

(Optional) Indicates *map* is an amplitude map. By default it is assumed to be a wavefront error map. Only one of *AMPLITUDE*, *MIRROR*, or *WAVEFRONT* should be set.

/MIRROR (IDL)
MIRROR=*True* or *False* (Python)
'MIRROR' (Matlab)

(Optional) Indicates *map* is a surface error map for a mirror (1/2 the wavefront error). By default it is assumed to be a wavefront error map. Only one of *AMPLITUDE*, *MIRROR*, or *WAVEFRONT* should be set.

RADIUS_PIX=*value* (IDL)
RADIUS_PIX=*value* (Python)
'RADIUS_PIX', *value* (Matlab)

Specifies the beam radius in units of map pixels. If given, the value of *SAMPLING* is ignored. When the file is read by **PROP_ERRORMAP**, the map will be resampled to match the current beam size.

SAMPLING=*value* (IDL)
SAMPLING=*value* (Python)
'SAMPLING', *value* (Matlab)

The sampling of the map in meters; ignored if *RADIUS_PIX* is defined. The file header keyword PIXSIZE will be set to this value.

/WAVEFRONT (IDL)
WAVEFRONT=*True* or *False* (Python)
'WAVEFRONT' (Matlab)

(Optional) Indicates *map* is a wavefront error map. By default it is assumed to be a wavefront error map, so this option is meant to allow the user to be clear about the map type. Only one of *AMPLITUDE*, *MIRROR*, or *WAVEFRONT* should be set.

See Also

PROP_ERRORMAP, PROP_READMAP

PROP_ZERNIKES

Apply circular Zernike polynomial aberrations to the current wavefront phase or amplitude component. The polynomial ordering established by Noll (J. Opt. Soc. Am., 66, 207 (1976)) is assumed. An arbitrary number of polynomials normalized for an unobscured aperture can be used, or just the first 22 for a centrally obscured aperture. Obscured Zernikes are used if the user specifies the *eps* parameter. The polynomial equations for the unobscured Zernikes can be printed using **PROP_PRINT_ZERNIKES**.

The first 22 Zernike aberrations are:

Number	Name	Number	Name
1	Piston	12	5th order 0º astigmatism
2	X tilt	13	5th order 45º astigmatism
3	Y tilt	14	X quadrafoil
4	Focus	15	Y quadrafoil
5	45º astigmatism	16	5th order X coma
6	0º astigmatism	17	5th order Y coma
7	Y coma	18	5th order X clover
8	X coma	19	5th order Y clover
9	Y clover (trefoil)	20	X pentafoil
10	X clover (trefoil)	21	Y pentafoil
11	3rd order spherical	22	5th order spherical

The coordinate system used to define the aberration patterns is aligned with that in the wavefront array (X and Y image axes). The azimuth of the aberration is measured counterclockwise from the +X axis. The Zernikes are normalized over a circular aperture with a radius equal to that of the beam at the current surface (the beam size returned by **PROP_GET_BEAMRADIUS**, which is not necessarily the illuminated beam radius). The user can specify an alternative normalization radius with the *RADIUS* keyword.

Syntax

IDL: **prop_zernikes**, *wavestruct, zernike_num, zernike_val [, obscuration_ratio] [, /AMPLITUDE] [, MAP=wfe] [, NAME=string] [, /NO_APPLY] [, RADIUS=value]*

Python: *[wfe =]* **proper.prop_zernikes**(*wavestruct, zernike_num, zernike_val [, obscuration_ratio] [, AMPLITUDE=True/False] [, NAME=string] [, NO_APPLY=True/False] [, RADIUS=value]*)

Matlab: *wavestruct_out = - OR -*
[*wavestruct_out, wfe*] = **prop_zernikes**(*wavestruct_in, zernike_num, zernike_val [, 'AMPLITUDE'] [, 'OBSCURATION_RATIO', obscuration_ratio] [,' NAME', string] [, 'NO_APPLY'] [, 'RADIUS', value]*);

Returns

wavestruct_out (Matlab)

(Required) The modified wavefront structure.

wfe (Python)
wfe (Matlab)

(Optional) A variable that will contain the two-dimensional wavefront error map created by this routine (values in meters RMS of wavefront error).

Arguments

wavestruct (IDL)
wavestruct (Python)
wavestruct_in (Matlab)
(Required) The current wavefront structure.

zernike_num
(Required) Scalar or 1D array of Zernike polynomial indices. The first 22 are listed above.

zernike_val
(Required) Scalar or 1D array of Zernike polynomial coefficients corresponding to the Zernike polynomials indexed by *zernike_num*. The values must be in meters of RMS phase error or dimensionless RMS amplitude error.

obscuration_ratio (IDL)
obscuration_ratio (Python)
'OBSCURATION_RATIO', *value* (Matlab)
(Optional) The central obscuration ratio ($D_{obscuration}/D_{pupil}$, valued 0.0 to 1.0). The default is 0.0 (clear aperture). If specified, only the first 22 Zernike polynomials can be used. The Zernike polynomials will be properly normalized for a centrally obscured aperture.

Keywords and Switches

/AMPLITUDE (IDL)
AMPLITUDE=*True* or *False* (Python)
'AMPLITUDE' (Matlab)
(Optional) Keyword that specifies that the Zernike values in *zernike_val* represent the wavefront RMS amplitude (rather than phase) variation. The current wavefront will be multiplied by the generated map.

MAP=*wfe* (IDL)
(Optional) Keyword set to a variable that will contain the two-dimensional wavefront error map created by this routine (values in meters RMS of wavefront error).

NAME=*string* (IDL)
NAME=*string* (Python)
'NAME', *string* (Matlab)
(Optional) Keyword set to a string containing the name of the current surface. If given, this routine will print 'Applying aberrations to *name*' when it is called.

/NO_APPLY (IDL)
NO_APPLY=*True* or *False* (Python)
'NO_APPLY' (Matlab)
(Optional) If this switch is set, then an aberration map will be generated but not applied to the wavefront. This is useful if you wish to create a map that will be applied later after some modification.

RADIUS=*value* (IDL)
RADIUS=*value* (Python)
'RADIUS', *value* (Matlab)

(Optional) Keyword specifying the radius in meters to which the Zernike polynomials are normalized. If this is not specified, the pilot beam radius, returned by **PROP_GET_BEAMRADIUS**, is used.

Examples

Add 0.5 μm RMS of defocus (Z_4) and 0.2 μm of X coma (Z_8) to the current wavefront:

```
IDL:     prop_zernikes, wavefront, [4,8], [0.5,0.2]*1.0e-6
Python:  proper.prop_zernikes( wavefront, [4,8], [0.5,0.2]*1.0e-6 )
Matlab:  wavefront = prop_zernikes( wavefront, [4,8], [0.5,0.2]*1.0e-6 );
```

Add 0.5 μm RMS of defocus (Z_4) to the current wavefront, and store the added wavefront error map into the variable `mp`:

```
IDL:     prop_zernikes, wavefront, 4, 0.5e-6, MAP=mp
Python:  mp = proper.prop_zernikes( wavefront, 4, 0.5e-6 )
Matlab:  [wavefront, mp] = prop_zernikes( wavefront, 4, 0.5e-6 );
```

See Also

PROP_ADD_PHASE, PROP_ERRORMAP, PROP_NOLL_ZERNIKES, PROP_PRINT_ZERNIKES, PROP_PSD_ERRORMAP, PROP_READMAP