Novel approaches for direct exoplanet imaging: Theory, simulations and experiments

Por, E.H.

Chapter 5

High Contrast Imaging for Python (HCIPy)

an open-source adaptive optics and coronagraph simulator

Adapted from
E. H. Por, S. Y. Haffert, V. M. Radhakrishnan, D. S. Doelman, M. A. M van Kooten and S. P. Bos (2018), Proc. SPIE 10703

Abstract
HCIPy is a package written in Python for simulating the interplay between wavefront control and coronagraphic systems. By defining an element which merges values/coefficients with its sampling grid/modal basis into a single object called Field, this minimizes errors in writing the code and makes it clearer to read. HCIPy provides a monochromatic Wavefront and defines a Propagator that acts as the transformation between two wavefronts. In this way a Propagator acts as any physical part of the optical system, be it a piece of free space, a thin complex apodizer or a microlens array.

HCIPy contains Fraunhofer and Fresnel propagators through free space. It includes an implementation of a thin complex apodizer, which can modify the phase and/or amplitude of a wavefront, and forms the basis for more complicated optical elements. Included in HCIPy are wavefront errors (modal, power spectra), complex apertures (VLT, Keck or Subaru pupil), coronagraphs (Lyot, vortex or apodizing phase plate coronagraph), deformable mirrors, wavefront sensors (Shack-Hartmann, Pyramid, Zernike or phase-diversity wavefront sensor) and multi-layer atmospheric models including scintillation).

HCIPy aims to provide an easy-to-use, modular framework for wavefront control and coronagraphy on current and future telescopes, enabling rapid prototyping of the full high-contrast imaging system. Adaptive optics and coronagraphic systems can be easily extended to include more realistic physics. The package includes a complete documentation of all classes and functions, and is available as open-source software.

5.1 Introduction

During the design process of an astronomical instrument, numerical simulations of the optical system play an integral role. All components are tested separately for their functionality, and then often integrated into a complete end-to-end simulation to verify that all subsystems are able to operate seamlessly together. The rapid prototyping provided by computer simulations has sped up the development process of high contrast imaging instruments tremendously. Some examples of simulation packages for adaptive optics (AO) are: YAO (Rigaut, 2002), OCTOPUS (Le Louarn et al., 2006), DASP (Basden et al., 2010), COMPASS (Gratadour et al., 2014) and most recently SOAPY (Reeves, 2016). Some examples of packages that simulate coronagraphs are PROPER (Krist, 2007) and POPPY (Perrin et al., 2012). In each of these packages, either the adaptive optics system or the coronagraph is simulated and and subsystems can only be optimized one at a time.

More recently, people have started looking into optimizing the full system, rather than each element separately. For example, one can optimize the coronagraph to suppress the aberrations that are common for a specific AO system. In this method there is little interaction between coronagraph and the AO system: the AO system can be simulated first, and the resulting data can be used for optimizing the coronagraph. In other cases the interaction can be stronger. For instance, in the case of optimizing the AO system to keep a specific region of interest in the focal plane dark, rather than flattening the wavefront (Radhakrishnan et al., 2018). Another example in this category is end-to-end simulations of post-coronagraphic focal-plane wavefront sensing methods including feedback to the AO system. For examples, see this excellent review (Jovanovic et al., 2018) and references therein. These use cases require both accurate and simultaneous simulation of both the AO system and the downstream coronagraph.

These aforementioned type of simulations place a particular set of requirements on the simulation software. The software needs to give baseline performance, to allow the user to simulate part of the system that he/she is not interested in with minimal effort, while still being modular to allow for inclusion of completely new components. The following guiding principles were used during the design process of HCIPy:

- *Modularity.* Components should be written to work independently of each other. For example, in this way, wavefront sensors can be exchanged, coronagraphs replaced and optical elements be made more

realistic, without having any influence on the operation of the other systems. This allows for rapid prototyping of new components, such as coronagraphs or wavefront sensors, and to compare different components as equally as possible.

- *User friendliness.* Components should be written to have extensive and sensible default parameters. This provides out-of-the-box baseline performance, meaning that even inexperienced users in one of the fields can code a working system.

- *Error avoidance.* Common user errors should be hard to make. That is, HCIPy is designed to handle many mathematical details, such as sampling requirements, automatically and in the background, providing clean and readable code. This allows the user to focus on system architecture rather than the details. HCIPy allows for access to these mathematical details if necessary, but their explicit nature makes mistakes easier to catch.

- *Pythonic.* HCIPy is written in the Python language, an interpreted high-level programming language. This programming language emphasizes code readability, allows for object-oriented programming, and is in use by many astrophysical projects (The Astropy Collaboration et al., 2018).

In Section 5.2 we discuss the core functionality of HCIPy. In Sections 5.3, 5.4, 5.5 and 5.6 we explore the functionality of HCIPy further, discussing optical systems, adaptive optics, coronagraph and more. We conclude with Section 8.5, and look towards future work.

5.2 Core functionality

The main mechanism in HCIPy for following these design principles is the use of `Field`s throughout the code base. These objects behave like a sampled physical field, sampling the value of a physical quantity in space. A `Field` contains a `Grid`, which is used to define the positions of the points in space, on which a `Field` is sampled. A `Coords` object in turn defines the values of the coordinates in a `Grid` object, while not containing any information on how to interpret these coordinates.

While the concepts of `Coords`, `Grid`s and `Field`s might seem esoteric and cumbersome, in most cases the user doesn't have to interact with these classes directly. Instead, the user interacts with functions to create and

modify them, and only uses the classes explicitly when he/she needs more control over the sampling for a specific part of their code. The following subsections present the implementation details of these three objects and explore the flexibility when programming with these objects.

5.2.1 Coords, Grids and Fields

An object of a `Coords` class can yield values for each dimension for each point in a `Grid`. Indexing is done using a single value, rather than one for each dimension. This is done to support `UnstructuredCoords`, which is a set of points with no internal structure. A `SeparatedCoords` object constrains coordinates to be a tensor product of the discretization along each axis. Some mathematical operations, such as Fourier transforms, can be performed much quicker when coordinate axes can be separated. Finally, a `RegularCoords` object constrains regularly-spaced coordinates on all axes. Fourier transforms on such a grid can use a Fast Fourier Transform (FFT) which greatly reduces computation times. Each of these `Coords` classes calculates their values on demand, rather than storing them explicitly. This greatly reduces the amount of required computer memory, especially for larger regularly-spaced grids.

A `Grid` object can be thought of as a discretization of some vector space. It contains a `Coords` class and applies physical meaning to the coordinate values by introducing a coordinate system. Additionally it can provide weights to each of the points: the interval, area, volume or hypervolume that a point covers in its vector space. These weights simplify calculation of integrals and derivatives. HCIPy supports a `CartesianGrid` for Cartesian spaces of any dimension, `PolarGrid` for polar coordinates and `SphericalGrid` for spherical coordinates. Conversion between coordinate systems can be performed by calling the `Grid.as_()` function with the required coordinate system. In addition, `Grid`s can be scaled and shifted by calling their respective functions.

A `Field` object is the discretized version of a physical field. It contains a `Grid` and an array of values. HCIPy supports scalar fields (e.g. an intensity field), vector fields (e.g. a vector electric field) and tensor fields of any dimension (e.g. a Jones matrix field, or a Mueller matrix field). Note that `Field`s in HCIPy do not have to be particle field, but simply a quantity that has a value for each point in space. For a scalar `Field` there is only a single spatial axis in the value array, independent of the dimension of the vector space, as is usually done. The dimensionality is hidden by the grid, rather than the value array. This may seem impractical and unjustified;

nevertheless this is the only way to consistently handle all types of `Coords`, in particular the `UnstructuredGrid`. Moreover, this strengthens the idea that a `Field` can be viewed as a covariant vector. For a vector `Field` the value array is two dimensional, the last axis being the spatial axis, and for tensor `Field`s the value array is n-dimensional for an $(n-1)$-dimensional tensor, again the last axis being the spatial axis. Special functions exist to make handing of vector and tensor fields easier, including but not limited to a `field_dot()` function, which multiplexes a dot product over the spatial axis, a `field_inv`, which takes the inverse of a two-dimensional tensor field for each point separately, and a `field_einsum`, which can calculate all Einstein-summation-convention formulas multiplexed over the spatial axis.

5.2.2 Field generators and visualization

We also use the concept of "field generators", which is a function that accepts a `Grid` as its sole argument and returns a `Field` on that `Grid`. This is in places where an analytical function needs to be evaluated on a `Grid` at another position in the code. A prime example is telescope apertures, which can be evaluated on any `Grid`, and can be user defined.

To make visualization of `Field`s easier, HCIPy includes several functions mimicking, and working in conjunction with, the "Matplotlib" plotting library (Hunter, 2007). The most versatile function is `field_imshow`, which can draw two-dimensional scalar fields. `Field`s with `UnstructuredCoords` are automatically interpolated to be able to draw them. Complex scalar fields are drawn using a custom two-dimensional color scale, which is extremely useful to be able to see both phase and amplitude in electric field plots. Masking of unused parts of the image is supported, and useful for phase plots on a pupil. Additional functions for contour plots for `Field`s are also implemented.

An example script in how to construct and transform `Grid`s, evaluate Field generators, and display `Field`s can be found in Listing 1, along with its output in Figure 5.1.

5.2.3 Fourier transforms

Fourier transform objects act on a `Field`, and return its Fourier transformed version, on the correct `Grid` corresponding to frequency space. HCIPy implements a `NaiveFourierTransform`, which naively implements the full Fourier integral, a `FastFourierTransform`, which uses the FFT algorithm and acts on regularly-spaced grids, and a `MatrixFourierTransform`, which

```python
# Import packages
from hcipy import *
import matplotlib.pyplot as plt
import numpy as np

# Make a separated polar grid
r = np.logspace(-1, 1, 11)
theta = np.linspace(0, 2*np.pi, 11)
coords = SeparatedCoords((r, theta))
polar_grid = PolarGrid(coords)
cartesian_grid = polar_grid.as_('cartesian')
plt.plot(cartesian_grid.x, cartesian_grid.y, '.')
plt.show()

# Create and plot Field
pupil_grid = make_pupil_grid(1024)
aperture = make_magellan_aperture(True)
evaluated_aperture = evaluate_supersampled(aperture, pupil_grid, 8)
imshow_field(evaluated_aperture)
plt.show()
```

Listing 1: An example code showing how to create, manipulate and show fields. A more detailed guide can be found in the online documentation for HCIPy.

uses a discrete-time Fourier transform, also known as a matrix Fourier transform (Soummer et al., 2007), and acts on any separated coordinates, including regularly-spaced grids. These three classes can be used directly or HCIPy can automatically choose which type of Fourier transform is fastest on the required input and output `Grid`s, and use that one.

5.2.4 Mode bases

Similar to field generators, HCIPy implements a wide range of mode bases, among others the Zernike modes, Fourier modes and disk harmonics. A `ModeBasis` is a list of objects, that form a linear mode basis. The modes of a `ModeBasis` can be easily put into a matrix, providing easy access to projection and deprojection of a function onto the mode basis. A code example for constructing several mode bases can be found in Listing 2 along with its output in Figure 5.2.

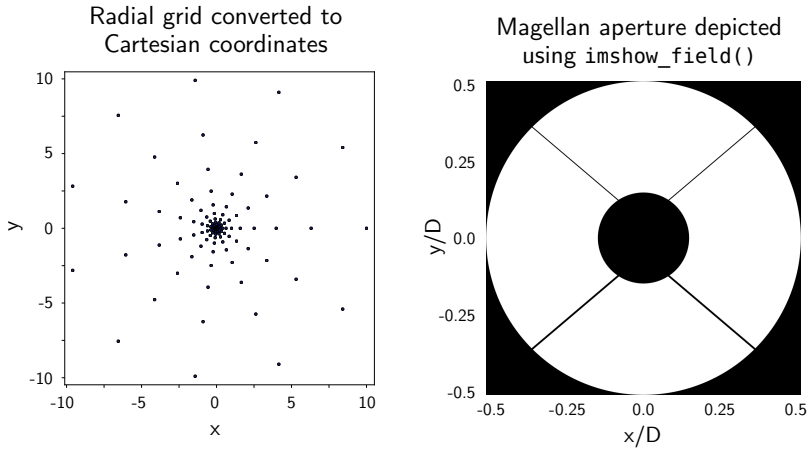

Figure 5.1: The resulting plots from the code in Listing 1. (*left*) The points for a separated grid using polar coordinates. (*right*) The Magellan aperture evaluated using supersampling on a pupil grid.

5.3 Optical systems

`Wavefront`s represent a monochromatic optical wavefront and combine an electric field with a wavelength. To support polarization, the electric field can be both scalar and vectorial, being a two-dimensional Jones vector or a full three-dimensional electric field. `Wavefront`s support many methods to make visualization easier, such as direct access to the phase, amplitude, intensity and power of the electric field.

These `Wavefront`s can be propagated through an optical system using `OpticalElement`s. An `OpticalSystem` can represent any physical optical element, such as `Apodizer`s, `SurfaceAberration`s, `MicrolensArray`s and even `DeformableMirror`s. All `OpticalElement`s support both forward and inverse propagation, as well as calculation of their transformation matrix, ie. the matrix of the linear transformation from input to output planes. The latter is extremely useful for post-coronagraphic wavefront control calculations and coronagraph optimization.

Propagations through space are also `OpticalElement`s, specifically `Propagator`s. Currently implemented are a `FresnelPropagator`, which is a near-field, paraxial propagator using an angular-spectrum method, and a `FraunhoferPropagator`, which is a far-field, paraxial propagator, which can be used to simulate lenses of specific focal lengths.

```
grid = make_pupil_grid(256)
zernike_modes = make_zernike_basis(num_modes=16, D=1, grid=grid)

for i, m in enumerate(zernike_modes):
    plt.subplot(4, 4, i + 1)
    imshow_field(m, cmap='RdBu')
plt.show()

gaussian_hermite_modes = make_gaussian_hermite_modes(grid, num_modes=16,
↪  mode_field_diameter=0.3)
gaussian_laguerre_modes = make_gaussian_laguerre_modes(grid, pmax=5, lmax=2,
↪  mode_field_diameter=0.3)
```

Listing 2: An example code demonstrate the implementation of mode bases in HCIPy. A more detailed guide can be found in the online documentation.

Light can be detected by `Detector`s, which can simulate camera defects and noise. A `NoisyDetector` simulates simple, empirical noise on the detector images, consisting of a flat field, dark current, photon noise and read noise. More complicated noise models are under consideration. An image from a detector can be processed using a `FrameCorrector`. This object performs dark and flat field corrections to any incoming image.

To support changing optics, a `DynamicOpticalSystem` class is available. This class allows for sub-sampled temporal integrations to allow for a fractional number of frames lag in extreme adaptive optics simulations. Updates to the optical system are scheduled and the light is internally propagated through the system. These updates are for example a changing atmosphere, a change in actuator positions for a deformable mirror, or readout of a detector. This class forms the basis for an AO system class.

5.4 Adaptive optics

5.4.1 Atmospheric modeling

To simulate the atmosphere above the telescope, HCIPy uses infinitely-thin phase screens, implemented as `AtmosphericLayer` objects. These phase screens can move over the telescope aperture, according to the "Frozen Flow" approximation (Taylor, 1938). The phase pattern can be generated in two different ways, extendable by the user. Subharmonics (Lane et al., ????) can be added by using a multi-scale Fourier transform method

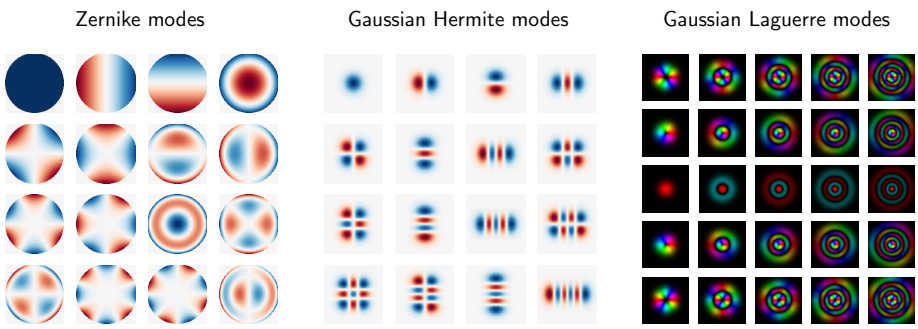

Figure 5.2: Some of the supported mode bases in HCIPy. This is the result of Listing 2.

by `FiniteAtmosphericLayer`. This yields excellent results for short-time simulations, where the phase screen doesn't move that much. For longer simulations, a `InfiniteAtmosphericLayer` is used, which extrudes a long ribbon of phase screen by adding rows sequentially, based on previous samples (Assémat et al., 2006). This method can also simulate non-stationary turbulence.

These single layers can be combined into a `MultiLayerAtmosphere` object. This object simulated the propagation between individual layers using Fresnel propagation, which can simulate scintillation. An example script introducing how to construct and use atmospheric layers and models can be found in Listing 3, along with an example output in Figure 5.3.

5.4.2 Wavefront sensing

Wavefront sensing is done using four operations from different classes. A `WavefrontSensorOptics` class simulates the optics in front of the detector for a certain wavefront sensor. Afterwards, a `Detector` class is used to get the wavefront sensor image. This image can be corrected for flat-field and dark effects using a `FrameCorrector` object. Finally, a `WavefrontSensorEstimator` is used to reduce the corrected detector image into an estimate of the wavefront. This can be either slopes, or actual wavefront, or anything that varies with wavefront. A `WavefrontSensor` object is available for folding all the different steps into a single object. This separation into optics and estimator makes it possible to play back and estimate wavefronts from real wavefront sensor images. An example script on how to construct and use a Pyramid wavefront sensor is presented in

```
# Create an atmospheric layer
layer = InfiniteAtmosphericLayer(pupil_grid, Cn_squared, L0, velocity,
↪  height)

# Propagate a wavefront through the layer
wf = layer(wf)

imshow_field(wf.phase, cmap='RdBu')
plt.show()

# Make a multi-layer atmosphere
layer = make_standard_atmospheric_layers(pupil_grid, L0)
atmos = MultiLayerAtmosphere(layers, scintillation=True)

wf = atmos(wf)
imshow_field(wf.intensity)
plt.show()
```

Listing 3: An example code showing the creation of an atmospheric model including scintillation. A more detailed guide can be found in the online documentation for HCIPy.

Listing 4.

HCIpy supports several wavefront sensors: among others, a Shack-Hartmann, Pyramid (Ragazzoni, 1996; Ragazzoni et al., 2002) (only unmodulated), Zernike (Bloemhof & Wallace, 2003; Zernike, 1935) and (generalized) optical-differentiation wavefront sensor (Haffert, 2016; Sprague & Thompson, 1972). Example detector images are shown for each of these wavefront sensors in Figure 5.4.

5.4.3 Wavefront control

All wavefront controllers are derived from a base `Controller` class. This class can transform measured wavefront sensor estimates from one or more `WavefrontSensorEstimator`s into actuator voltages for one or more `DeformableMirror`s. We will implement an integral controller, which uses a (leaky) integrator, and a full PID controller.

The wavefront sensor estimates can be optionally filtered by an `Observer`, which allows for separation of estimation and control. We will implement a modal reconstructor, which reconstructs modes using a linear transformation matrix, a Kalman filter and a linear minimum mean squared error

```
# Create the optics for the WFS
wfso = PyramidWavefrontSensorOptics(pupil_grid, pupil_separation=1.5,
↪  num_pupil_pixels=64, refractive_index=1.5)

# Create detector model
detector = NoisyDetector(input_grid=wfso.output_grid, flat_field=0.01)

# Create the frame corrector
frame_corrector = BasicFrameCorrector(flat_field=detector.flat_field)

# Create the WFS estimator
wfse = PyramidWavefrontSensorEstimator(aperture=circular_aperture(1),
↪  output_grid=wfso.output_grid)

# Combine all into a single object
wfs = WavefrontSensor(wfso, detector, frame_corrector, wfse)

# Measure an incoming wavefront using a 1sec exposure
wfs.integrate(incoming_wavefront, 1)
slopes = wfs.read_out()
```

Listing 4: An example code showing the setup and reading out of an unmodulated Pyramid wavefront sensor. A more detailed guide can be found in the online documentation for HCIPy.

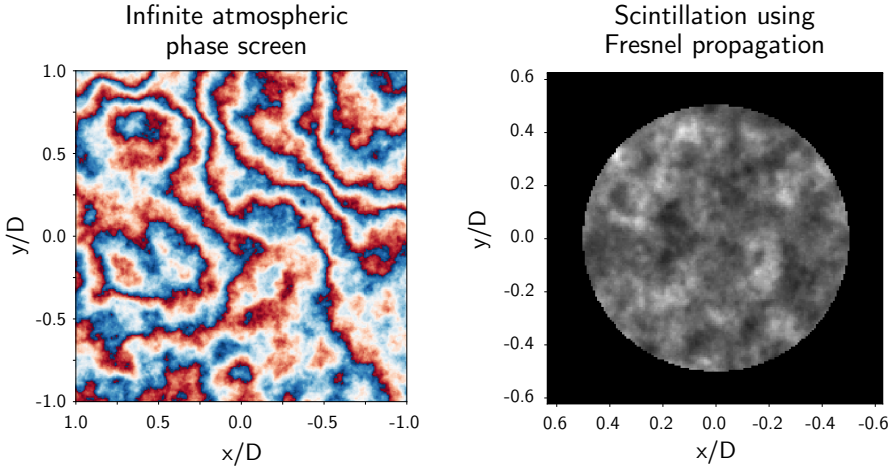

Figure 5.3: The simulated images from Listing 3. These images show a simulated phase screen that can be extruded infinitely in any direction (*left*), and the scintillation from a series of phase screens with Fresnel propagation between layers (*right*).

(LMMSE) estimator. All observers are designed to be used outside of wavefront control as well.

5.5 Coronagraphy

Coronagraphs are implemented as optical elements, most of them taking a pupil-plane `Wavefront` as input and outputting a post-coronagraphic pupil-plane `Wavefront`. A `LyotCoronagraph` and `OccultedLyotCoronagraph` implement both cases of Lyot coronagraphs. The first implements a Lyot coronagraph with a small occulting mask size, and assumes that the part outside of the focal-plane mask is transmitted. The second case assumes that the part outside of the focal-plane mask is occulted. Both of these are in use by, for example, the shaped-pupil Lyot coronagraph. The focal-plane and Lyot-stop masks can be replaced by realistic `OpticalElements` to more accurately describe a specific implementation of a Lyot-style coronagraph.

The `VortexCoronagraph` class implements a propagation through an optical vortex (Foo et al., 2005; Mawet et al., 2005). It uses a multi-scale optical propagation scheme to resolve the vortex singularity. Functions for

generating pupil and Lyot-stop masks for the ring-apodized vortex coronagraph (Mawet et al., 2013) are implemented.

The `ApodizingPhasePlateCoronagraph` implements an APP coronagraph (Codona & Angel, 2004; Kenworthy et al., 2007; Snik et al., 2012). Several functions for optimizing this coronagraph, ie. calculating a phase pattern that creates a dark zone in a region of interest in the focal plane, are implemented. These methods include a quick-and-versatile method based on a modified Gerchberg-Saxton algorithm by Christoph Keller (in prep.), and a globally optimal linear optimization method (Por, 2017). The latter of these requires the installation of Gurobi (Gurobi Optimization, 2016) including its Python interface.

A `ShapedPupilCoronagraph` is also available, including methods for optimizing those, based on global linear optimization (Carlotti et al., 2011). The same function can optimize shaped-pupil Lyot coronagraphs (Zimmerman et al., 2016) as well, for fixed focal-plane and Lyot-stop masks.

A `PerfectCoronagraph` is also implemented. The implementation is based on fitting and subsequent subtraction of the electric field of an unaberrated PSF (Cavarroc et al., 2006). Higher-order perfect coronagraphs are implemented as well using a similar method (Guyon et al., 2006). These coronagraphs are highly computationally efficient and can be used for quick comparisons with other coronagraphs, or if a specific coronagraph model is not needed.

5.6 Miscellaneous

5.6.1 Polarization

Polarization is supported in HCIPy using Jones calculus. This includes the implementation of polarizers and waveplates with (spatially) varying fast-axis orientation, retardance and circularity. This has allowed for research into broadband vector Apodizing Phase Plates for polarimetry and polarization aberrations, see Figure 5.6.

5.6.2 Performance

While computational performance is not the main goal of HCIPy, care has been taken to retain as much performance as possible. Algorithmic improvements are always preferred in cases where they don't compromise on readability and/or generality. This can be seen in the judicious usage of the Matrix Fourier transform, wherever possible. When performing

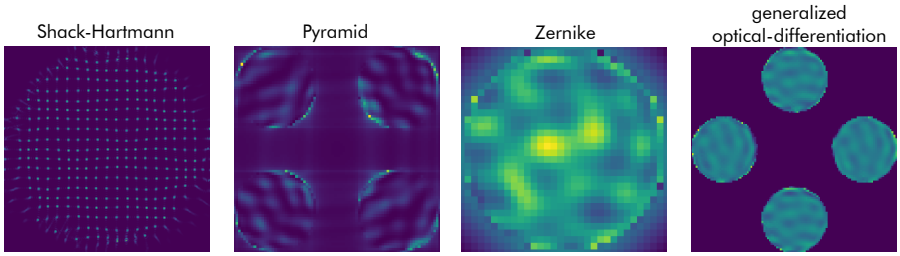

Figure 5.4: Simulated images for different wavefront sensors using HCIPy. All wavefront sensors see the same phase aberration. The amplitude of the aberration was scaled to approximately the linear range of the wavefront sensor, for visualization purposes.


```
# Create the pupil
pupil_grid = make_pupil_grid(2048, 4)
pup = circular_aperture(1)(pupil_grid)

# Create a vortex coronagraph
coro = VortexCoronagraph(pupil_grid, charge=4)

# Create some aberrations
aber = SurfaceAberration(pupil_grid, ptp=1/8, diameter=1)

# Create a Lyot stop mask
lyot = lambda grid: circular_aperture(0.95)(grid) +
    circular_aperture(0.05, [1.8, 0])(grid)
lyot = Apodizer(lyot(pupil_grid))

# Create post-coronagraphic image
img = prop(lyot(coro(aber(Wavefront(pup)))))
```

Listing 5: An example code showing the simulation of an image for a self-coherent camera (SCC; Baudoz et al., 2005) behind a charge 4 vortex coronagraph. A more detailed guide can be found in the online documentation.

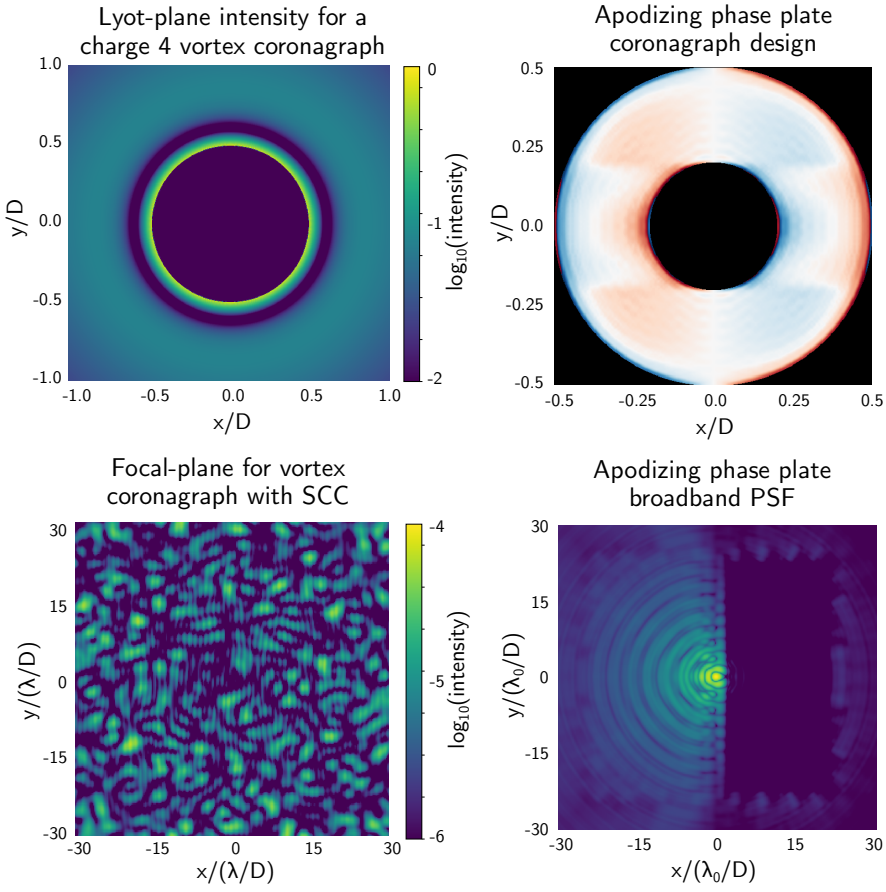

Figure 5.5: A series of simulated coronagraphic images with HCIPy. (*top left*) The Lyot-plane field for a charge 4 vortex coronagraph. (*bottom left*) The focal-plane image of the vortex coronagraph, with a self-coherent camera (Baudoz et al., 2005) Lyot-stop mask. The code for generating this image can be found in Listing 5. (*top right*) An example apodizing phase plate design optimized using HCIPy. (*bottom right*) The broadband point spread function for this APP design.

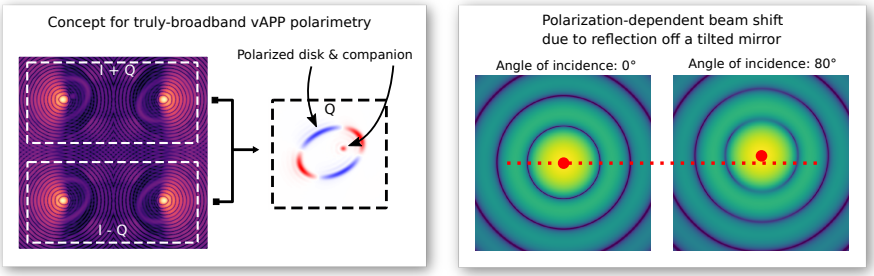

Figure 5.6: Some examples of polarization effects that can be simulated. (*left*) A truly-broadband vAPP implementation for polarimetry. A double difference detection scheme is simulated. This figure is adapted from Bos et al. (2018). (*right*) The effect of polarization due to reflection off a tilted aluminum-coated mirror. This figure is courtesy of Rob van Holstein, work to appear in Van Holstein et al. in prep..

a Fourier transform, HCIPy automatically determines whether a Matrix Fourier transform or a zero-padded Fast Fourier transform would yield the shortest computation time.

Care has been taken with the numerical efficiency as well. The library "NumPy" is used for arrays and simple linear algebra operations (Walt et al., 2011). The library "SciPy" is used for more intricate linear algebra (Jones et al., 2014). The library "PyFFTW" is used to calculate FFTs using the well-known FFTW library (Frigo & Johnson, 2005). HCIPy does not internally use multiprocessing or multithreading yet, apart from the parallelization by linear algebra packages and FFTW. This seemingly strange design decision has its basis in the level at which to parallelize. It is much easier and computationally efficient to parallelize at the highest level possible, as in most cases the workload is embarrassingly parallel. Think for instance at multiplexing different wavelengths for a broadband simulation, or the different parameter sets in a parameter study. It is much harder to efficiently parallelize the Fast Fourier transform. The task of parallelizing is therefore left to the user: he/she now has to write the code for parallelization. GPU support is under development.

5.7 Conclusions

5.7.1 Overview

HCIPy is an object-oriented framework written in Python for performing end-to-end simulations of high-contrast imaging instruments. It is built around the concept of Fields, a sampled version of physical fields. This union allows the user to focus on the high-level structure of their own code, rather than worry about sampling details.

HCIPy defines wavefronts, optical elements and optical systems. Propagators are optical elements that are used for propagation through free space; HCIPy implements both Fraunhofer and Fresnel diffraction propagators. Jones calculus is used for polarization calculations, with polarizers and waveplates supported out of the box. Atmospheric turbulence is simulated using thin infinitely-long phase screens. Scintillation is modeled using Fresnel propagation between individual layers. Implemented wavefront sensors include the Shack-Hartmann and Pyramid wavefront sensors. For coronagraphy the vortex, Lyot and APP coronagraphs are implemented, with methods for globally optimizing pupil-planes, both in phase and amplitude, based on linear optimization.

As the AO and coronagraphy can be used in series in the same software package, HCIPy allows for end-to-end simulations of post-coronagraphic focal-plane wavefront sensing including feedback and implementation of AO control algorithms specifically designed for high-contrast imaging applications. HCIPy is available as open-source software on GitHub, licensed under the MIT license, at https://github.com/ehpor/hcipy.

5.7.2 Future plans

The following subjects are actively considered or in development:

- *GPGPU support.* With the advances in the processing power contained in Graphics Processing Units (GPUs) and the almost universal support for General Purpose computations on GPUs (Nickolls et al., 2008; Stone et al., 2010), support for GPGPU cannot be neglected in any major simulation package. Especially the GPUs affinity for linear algebra operations cannot be ignored. GPGPU support is currently under development for HCIPy. As most heavy numerical calculations are performed on Fields, it makes sense to seamlessly do calculations on Fields in the background on the GPU.

- *Data reduction.* With ground-based observations post-processing techniques play an important role. This cannot be neglected in simulations: we would want to optimize our AO system and coronagraph for post-processed contrast rather than pre-processed contrast. Data-reduction techniques will preferably be included in HCIPy with wrappers to more advanced and mature data-reduction packages such as pyKLIP (Wang et al., 2015) and VIP (Gonzalez et al., 2017).

- *Improved detector modeling.* Noise behavior of detectors have a significant influence on the AO system performance, especially at low flux levels. Improved detector models, such as an (empirical) CCD and CMOS (Konnik & Welsh, 2014), or EMCCD (Hirsch et al., 2013) noise model would be helpful in characterizing these effects.

- *Non-paraxial vector diffraction.* Taking into account the vector nature of light goes further than just supporting polarization-sensitive optical elements: the propagation of light needs to be changed as well, especially in low F-number beams where non-paraxial effects play a much larger role (Bos et al., 2017).

- *More wavefront sensors.* Support for more wavefront sensors is always valued. Currently the coronagraphic Modal Wavefront Sensor (Wilby et al., 2017) and phase diversity (Gonsalves, 1982) are planned.

Community input and extensions are warmly appreciated.

References

Assémat, F., Wilson, R. W., & Gendron, E. 2006, Optics express, 14, 988
Basden, A., Geng, D., Myers, R., & Younger, E. 2010, Applied optics, 49, 6354
Baudoz, P., Boccaletti, A., Baudrand, J., & Rouan, D. 2005, Proceedings of the International Astronomical Union, 1, 553
Bloemhof, E. E., & Wallace, J. K. 2003, in Astronomical Adaptive Optics Systems and Applications, Vol. 5169, International Society for Optics and Photonics, 309–321
Bos, S. P., Doelman, D. S., De Boer, J., et al. 2018, in Proc. SPIE, Vol. 10706, Advances in Optical and Mechanical Technologies for Telescopes and Instrumentation III
Bos, S. P., Haffert, S. Y., & Keller, C. U. 2017, in Polarization Science and Remote Sensing VIII, Vol. 10407, International Society for Optics and Photonics, 1040709
Carlotti, A., Vanderbei, R., & Kasdin, N. 2011, Optics Express, 19, 26796
Cavarroc, C., Boccaletti, A., Baudoz, P., Fusco, T., & Rouan, D. 2006, Astronomy & Astrophysics, 447, 397
Codona, J. L., & Angel, R. 2004, The Astrophysical Journal Letters, 604, L117
Foo, G., Palacios, D. M., & Swartzlander, G. A. 2005, Optics letters, 30, 3308
Frigo, M., & Johnson, S. G. 2005, Proceedings of the IEEE, 93, 216
Gonsalves, R. A. 1982, Optical Engineering, 21, 215829
Gonzalez, C. A. G., Wertz, O., Absil, O., et al. 2017, The Astronomical Journal, 154, 7
Gratadour, D., Puech, M., Vérinaud, C., et al. 2014, in Adaptive Optics Systems IV, Vol. 9148, International Society for Optics and Photonics, 91486O
Gurobi Optimization, I. 2016, Gurobi Optimizer Reference Manual. http://www.gurobi.com
Guyon, O., Pluzhnik, E., Kuchner, M., Collins, B., & Ridgway, S. 2006, The Astrophysical Journal Supplement Series, 167, 81
Haffert, S. 2016, Optics express, 24, 18986
Hirsch, M., Wareham, R. J., Martin-Fernandez, M. L., Hobson, M. P., & Rolfe, D. J. 2013, PLoS One, 8, e53671
Hunter, J. D. 2007, Computing In Science & Engineering, 9, 90, doi: 10.1109/MCSE.2007.55
Jones, E., Oliphant, T., & Peterson, P. 2014
Jovanovic, N., Absil, O., Baudoz, P., et al. 2018, in Proc. SPIE, Vol. 10703, Adaptive Optics Systems VI
Kenworthy, M. A., Codona, J. L., Hinz, P. M., et al. 2007, The Astrophysical Journal, 660, 762
Konnik, M., & Welsh, J. 2014, arXiv preprint arXiv:1412.4031
Krist, J. E. 2007, in Optical Modeling and Performance Predictions III, Vol. 6675, International Society for Optics and Photonics, 66750P
Lane, R., Glindemann, A., Dainty, J., et al. ????

Le Louarn, M., Vérinaud, C., Korkiakoski, V., Hubin, N., & Marchetti, E. 2006, in Advances in Adaptive Optics II, Vol. 6272, International Society for Optics and Photonics, 627234

Mawet, D., Pueyo, L., Carlotti, A., et al. 2013, The Astrophysical Journal Supplement Series, 209, 7

Mawet, D., Riaud, P., Absil, O., & Surdej, J. 2005, The Astrophysical Journal, 633, 1191

Nickolls, J., Buck, I., Garland, M., & Skadron, K. 2008, in ACM SIGGRAPH 2008 classes, ACM, 16

Perrin, M. D., Soummer, R., Elliott, E. M., Lallo, M. D., & Sivaramakrishnan, A. 2012, in Proc. SPIE, Vol. 8442, Space Telescopes and Instrumentation 2012: Optical, Infrared, and Millimeter Wave, 84423D

Por, E. H. 2017, in Techniques and Instrumentation for Detection of Exoplanets VIII, Vol. 10400, International Society for Optics and Photonics, 104000V

Radhakrishnan, V. M., Doelman, N., & Keller, C. U. 2018, in Proc. SPIE, Vol. 10703, Adaptive Optics Systems VI

Ragazzoni, R. 1996, Journal of modern optics, 43, 289

Ragazzoni, R., Diolaiti, E., & Vernet, E. 2002, Optics communications, 208, 51

Reeves, A. 2016, in Adaptive Optics Systems V, Vol. 9909, International Society for Optics and Photonics, 99097F

Rigaut, F. 2002, YAO Adaptive Optics Simulation Package. `https://github.com/frigaut/yao`

Snik, F., Otten, G., Kenworthy, M., et al. 2012, in Modern Technologies in Space- and Ground-based Telescopes and Instrumentation II, Vol. 8450, International Society for Optics and Photonics, 84500M

Soummer, R., Pueyo, L., Sivaramakrishnan, A., & Vanderbei, R. J. 2007, Optics Express, 15, 15935

Sprague, R. A., & Thompson, B. J. 1972, Applied optics, 11, 1469

Stone, J. E., Gohara, D., & Shi, G. 2010, Computing in science & engineering, 12, 66

Taylor, G. I. 1938, Proceedings of the Royal Society of London. Series A, Mathematical and Physical Sciences, 476

The Astropy Collaboration, et al. 2018, ArXiv e-prints. `https://arxiv.org/abs/1801.02634`

Walt, S. v. d., Colbert, S. C., & Varoquaux, G. 2011, Computing in Science & Engineering, 13, 22

Wang, J. J., Ruffio, J.-B., De Rosa, R. J., et al. 2015, Astrophysics Source Code Library

Wilby, M. J., Keller, C. U., Snik, F., Korkiakoski, V., & Pietrow, A. G. 2017, Astronomy & Astrophysics, 597, A112

Zernike, F. 1935, Tech. Phys, 16, 454

Zimmerman, N. T., Riggs, A. E., Kasdin, N. J., Carlotti, A., & Vanderbei, R. J. 2016, Journal of Astronomical Telescopes, Instruments, and Systems, 2, 011012