HoloPy Documentation

Release 3.5.0

Manoharan Lab, Harvard University

May 24, 2021

Contents

Release 3.5.0

`HoloPy` is a python based tool for working with digital holograms and light scattering. HoloPy can be used to analyze holograms in two complementary ways:

- **Backward propagation of light from a digital hologram to *reconstruct* 3D volumes.**
  - This approach requires no prior knowledge about the scatterer
- **Forward propagation of light from a *scattering calculation* of a predetermined scatterer.**
  - Comparison to a measured hologram with *Bayesian inference* allows precise measurement of scatterer properties and position.

HoloPy provides a powerful and user-friendly python interface to fast scattering and optical propagation theories implemented in Fortran and C code. It also provides a set of flexible objects that make it easy to describe and analyze data from complex experiments or simulations.

HoloPy started as a project in the Manoharan Lab at Harvard University. If you use HoloPy, you may wish to cite one or more of the sources listed in *References and credits*. We also encourage you to sign up for our User Mailing List or join us on GitHub to keep up to date on releases, answer questions, and benefit from other users' questions.

CHAPTER 1

HoloPy Release Notes

1.1 Holopy 3.5

1.1.1 Announcements

If you encounter errors loading prevoiusly saved HoloPy objects, try loading them with HoloPy 3.4 and saving a new version. See deprecation notes below.

1.1.2 New Features

- New `AberratedMieLens` allows for calculating holograms of spheres as imaged through a lens with spherical aberrations.

1.1.3 Improvements

- Calling a numpy ufunc on a Prior object with name kwarg gives the resulting TransformedPrior object that name, e.g. clip_x = np.min(x, name='clipped').
- Cleaned up model parameter names created from TransformedPrior objects.
- CmaStrategy now scales first step size based on initial population, not prior.
- Inference models work with scattering theories that require parameters. See more in the user guide *The HoloPy Scatterer*.
- Interpolation in background images is now robust to adjacent dead pixels.
- Restored ability to call scattering functions on parameterized scatterers.

1.1.4 Documentation

- Updated inference tutorial

1.1.5 Bugfixes

- NmpfitStrategy now correctly accounts for non-uniform priors when optimizing.
- Functional fitting interface no longer lets alpha go to zero.
- Now able to save Model objects whose scatterer attribute contains xarrays.

1.1.6 Compatibility Notes

- HoloPy now assumes dictionaries are ordered, so it requires python>=3.7.

1.1.7 Developer Notes

- The `ScatteringTheory` now performs scattering calculations only, as its single responsibility. This should make it easier to implement new scattering theories. Code that was previously in `ScatteringTheory` that calculated deterimed at which points the scattering matrices or scattered fields needed to be calculated is now in *holopy.scattering.imageformation*.
- The parameter parsing previously done by the *Model* class has now been broken out to a new *hp.core.mapping* module so it can be accessed by non-*Model* objects.
- *prior.py* has been moved from to hp.core module from hp.inference but is still accessible in the hp.inference namespace.

1.1.8 Deprecations

- PerfectLensModel is now deprecated; lens models are now directly fittable with either AlphaModel or Exact-Model. To do so, pass in a `holopy.prior.Prior` object as the *lens_angle*.
- Inference-related deprecations started in 3.4 are now complete. This means that some old *holopy* inference objects are no longer loadable. If you still need to access these objects, *holopy* version 3.4 will let you load old inference objects and save them in the new format that is compatible with this (and future) versions of *holopy*.

1.2 Holopy 3.4

1.2.1 New Features

- New `Lens` scattering theory to model the effect of an objective lens can be applied to any other scattering theory.
- New `TransformedPrior` that applies a function to one or multiple component `Prior` objects and maintains ties in a `Model`.

1.2.2 Improvements

- DDA scattering theories no longer default to printing intermediate C output.
- It is now possible to save all slices of a reconstruction stack as images.
- Rearrangement of some Scatterer properties and methods so they are now accessible by a broader group of classes.

- PerfectLensModel now accepts hologram scaling factor alpha as a parameter for inference.
- It is now possible to pass an inference strategy to the high-level fit() and sample() functions, either by name or as a Strategy object.
- High level inference functions fit() and sample() are now accessible in the root HoloPy namespace as hp.fit() and hp.sample().
- Scatterer.parameters() now matches the arguments to create the scatterer instead of deconstructing composite objects.
- New prior.renamed() method to create an identical prior with a new name.
- New way to easily construct scatterers from model parameters with `model.scatterer_from_parameters()`.
- New `model.initial_guess` attribute which can be used to evaluate initial guess by psasing into `model.scatterer_from_parameters()` or `model.forward()` methods.
- Model parameters now use the names of their prior objects if present.
- Standardized parameter naming across composite objects (eg. list, dict).
- Any model parameters can now be tied, not just specific combinations within Scatterers objects.
- Expanded math operations of *`Prior`* objects, including numpy ufuncs.
- Math operations on `Prior` objects now use *`TransformedPrior`* to maintain ties when used in a *`Model`*.

1.2.3 Documentation

- New user guide on *The HoloPy Scatterer*.
- New user guide on *The HoloPy Scattering Theories*.
- More discussion of scattering theories in tutorial.

1.2.4 Deprecations

- The model.fit() and model.sample() methods have been deprecated in favour of the high-level hp.fit() and hp.sample functions().
- Adjustments to saving of Model objects (and Results objects containing them). Backwards compatibility is supported for now, but be sure to save new copies!
- Scatterer.guess no longer exists. Instead, you must define a model and use: `model.scatterer_from_parameters(model.initial_guess)`.
- Scatterer.from_parameters() is no longer guaranteed to return a definite object.
- Composite scatterers no longer keep track of tied parameters.
- Scattering interface functions such as calc_holo() now require a definite scatterer without priors.

1.2.5 Bugfixes

- Fortran output no longer occasionaly leaks through the output supression context manager used by multiple scattering theories.
- Restored ability to visualize slices through a scatterer object

- Now possible to fit only some elements of a list, eg. Scatterer center
- Models can now include xarray parameters and still support saving/loading.
- The `MieLens` scattering theory now works for both large and small spheres.
- The `Lens` theory works for arbitrary linear polarization of the incoming light. This bug was not present on any releases, only on the development branch.

1.2.6 Compatibility Notes

- Holopy's hard dependencies are further streamlined, and there is improved handling of missing optional dependencies.

1.2.7 Developer Notes

- Documentation now automatically runs sphinx apidoc when building docs.
- New Scatterer attribute `_parameters` provides a view into the scatterer and supports editing.
- `ComplexPrior` now inherits from `TransformedPrior`, but Model maps don't keep track of this, e.g. in *model.scatterer*.

1.3 Holopy 3.3

1.3.1 New Features

- Inference in *holopy* has been overhauled; take a look at the updated docs to check it out! Briefly, the inference and fitting modules have been combined into a unified, object-oriented interface, with several convenience functions available to the user both for the inference strategies and the inference results. One noticeable change with this is that the least-squares based fitting algorithms such as *Nmpfit* now work correctly with priors, including with non-uniform priors. There is also a new, user-friendly functionality for inference in *holopy*. Moreover, the inference pipelines can work with arbitrary user-defined functions instead of just holograms.
- There is a new scattering theory, *holopy.scattering.theory.MieLens*, which describes the effect of the objective lens on recorded holograms of spherical particles. This new theory is especially useful if you want to analyze particles below the microscope focus.
- There are two new inference strategies: a global optimizer CMA-ES strategy, under *holopy.inference.cmaes.CmaStrategy*, and a least-squares strategy which uses *scipy.optimize.leastsq* instead of the *Nmpfit* code.

1.3.2 Deprecations

- The keyword argument *normals* is deprecated in *detector_points*, *detector_grid*, and related functions, as the old implementation was incorrect. This deprecation is effective immediately; calling code with the *normals* keyword will raise a *ValueError*.
- The old fitting interface, in *holopy.fitting*, is in the process of being deprecated (see "New Features" above). Calling the old fitting interface will raise a *UserWarning* but will otherwise work until the next *holopy* release.

1.3.3 Bugfixes

In addition to many minor bugfixes, the following user-facing bugs have been fixed:

- *load_average* now works with a cropped reference image and uses less memory on large image stacks.
- Issues with loss of fidelity on saving and loading objects have been fixed.
- A bug where *hp.propagate* failed when *gradient_filter=True* has been fixed.
- Tied parameters in inference calculations works correctly on edge cases.
- Inference should work with more generic scatterers.
- The Fortran code should be easier to build and install on Windows machines. This is partially done via a post-install script that checks that files are written to the correct location (which corrects some compiler differences between Windows and Linux). We still recommend installing Holopy with conda.

1.3.4 Improvements

- User-facing docstrings have been improved throughout *holopy*.
- *schwimmbad* now handles parallel computations with Python's *multiprocessing* or *mpi*.
- More types of objects can be visualized with *hp.show*.
- DDA default behaviour now has *use_indicators=True* since it is faster and better tested
- The scaling of initial distributions both for Markov-Chain Monte Carlo and for CMA inference strategies can now be specified.

1.3.5 Compatibility Notes

- We are curently phasing out support for pre-3.6 Python versions (due to ordered vs unordered dicts).

1.3.6 Developer Notes

- Test coverage has dramatically increased in *holopy*.
- Tests no longer output extraneous information on running.
- The *ScatteringTheory* class has been refactored to allow for faster, more flexible extension.

1.3.7 Miscellaneous Changes

- Some previously required dependencies are now optional.

CHAPTER 2

Tutorials

Skip to the *Loading Data* tutorial if you already have HoloPy installed and want to get started quickly.

2.1 Getting Started

2.1.1 Installation

As of version 3.0, HoloPy supports only Python 3. We recommend using the anaconda distribution of Python, which makes it easy to install the required dependencies. HoloPy is available on conda-forge, so you can install it with:

```
conda install -c conda-forge holopy
```

in a shell, terminal, or command prompt. Once you have HoloPy installed, open an IPython console or Jupyter Notebook and run:

```
import holopy
```

If this line works, skip to *Using HoloPy* before diving into the tutorials.

You can also build HoloPy from source by following the instructions for *Installing HoloPy for Developers*.

Dependencies

HoloPy's hard dependencies can be found in requirements.txt. Optional dependencies for certain calculations include:

- a-dda (Discrete Dipole calculations of arbitrary scatterers)
- mayavi2 (if you want to do 3D plotting [experimental])

2.1.2 Using HoloPy

You will probably be most comfortable using HoloPy in Jupyter (resembles Mathematica) or Spyder (resembles Matlab) interfaces. HoloPy is designed to be used with an interactive backend. In the console, try running:

```
from holopy import check_display
check_display()
```

You should see an image, and you should be able to change the square to a circle or diamond by using the left/right arrow keys. If you can, then you're all set! Check out our *Loading Data* tutorial to start using HoloPy. If you don't see an image, or if the arrow keys don't do anything, you can try setting your backend with *one* of the following:

```
%matplotlib tk
%matplotlib qt
%matplotlib gtk
%matplotlib gtk3
```

Note that these commands will only work in an IPython console or Jupyter Notebook. If the one that you tried gave an `ImportError`, you should restart your kernel and try another. Note that there can only be one matplotlib backend per ipython kernel, so you have the best chance of success if you restart your kernel and immediately enter the `%matplotlib` command before doing anything else. Sometimes a backend will be chosen for you (that cannot be changed later) as soon as you plot something, for example by running `test_disp()` or *`show()`*. Trying to set to one of the above backends that is not installed will result in an error, but will also prevent you from setting a different backend until you restart your kernel.

An additional option in Spyder is to change the backend through the menu: Tools > Preferences > IPython console > Graphics. It will not take effect until you restart your kernel, but it will then remember your backend for future sessions, which can be convenient.

Additional options for inline interactive polts in jupyter are:

```
%matplotlib nbagg
%matplotlib widget
```

2.2 Loading Data

HoloPy can work with any image data, but our tutorials will focus on holograms.

2.2.1 Loading and viewing a hologram

We include a couple of example holograms with HoloPy. Lets start by loading and viewing one of them

```
import holopy as hp
from holopy.core.io import get_example_data_path
imagepath = get_example_data_path('image01.jpg')
raw_holo = hp.load_image(imagepath, spacing = 0.0851)
hp.show(raw_holo)
```

The first few lines just specify where to look for an image. The most important line actually loads the image so that you can work with it:

```
raw_holo = hp.load_image(imagepath, spacing = 0.0851)
```

HoloPy can import any image format that can be handled by Pillow.

The spacing argument tells holopy about the scale of your image. Here, we had previously measured that each pixel is a square with side length 0.0851 micrometers. In general, you should specify `spacing` as the distance between adjacent pixel centres. You can load an image without a spacing value if you just want to look at it, but holopy calculations will give incorrect results.

The final line displays the loaded image on your screen with the built-in HoloPy `show()` function. If you don't see anything, you may need to set your matplotlib backend. Refer to *Using HoloPy* for instructions.

2.2.2 Correcting Noisy Images

The raw hologram has some non-uniform illumination and an artifact in the upper-right corner. These can be removed if you take a background image with the same optical setup but without the object of interest. Dividing the raw hologram by the background using `bg_correct()` improves the image a lot.

```
from holopy.core.process import bg_correct
bgpath = get_example_data_path('bg01.jpg')
bg = hp.load_image(bgpath, spacing = 0.0851)
holo = bg_correct(raw_holo, bg)
hp.show(holo)
```

Often, it is beneficial to record multiple background images. In this case, we want an average image to pass into `bg_correct()` as our background.

```
bgpath = get_example_data_path(['bg01.jpg', 'bg02.jpg', 'bg03.jpg'])
bg = hp.core.io.load_average(bgpath, refimg = raw_holo)
holo = bg_correct(raw_holo, bg)
hp.show(holo)
```

Here, we have used `load_average()` to construct an average of the three background images specified in `bgpath`. The `refimg` argument allows us to specify a reference image that is used to provide spacing and other metadata to the new averaged image.

If you are worried about stray light in your optical train, you should also capture a dark-field image of your sample, recorded with no laser illumination. A dark-field image is specified as an optional third argument to `bg_correct()`.

```
dfpath = get_example_data_path('df01.jpg')
df = hp.load_image(dfpath, spacing = 0.0851)
holo = bg_correct(raw_holo, bg, df)
hp.show(holo)
```

Holopy includes some other convenient tools for manipulating image data. See the *HoloPy Tools* page for details.

2.2.3 Telling HoloPy about your Experimental Setup

Recorded holograms are a product of the specific experimental setup that produced them. The image only makes sense when considered with the experimental conditions in mind. When you load an image, you have the option to specify some of this information in the form of *metadata* that is associated with the image. In fact, we already saw an example of this when we specified image spacing earlier. The sample in our image was immersed in water (refractive index 1.33). Illumination was by a red laser with wavelength 660 nm and polarization in the x-direction. We can tell HoloPy all of this information when loading the image:

```
raw_holo = hp.load_image(imagepath, spacing=0.0851, medium_index=1.33, illum_
↪wavelen=0.660, illum_polarization=(1,0))
```

You can then view these metadata values as attributes of `raw_holo`, as in `raw_holo.medium_index`. However, you must use a special function `update_metadata()` to edit them. If we forgot to specify metadata when loading the image, we can use `update_metadata()` to add it later:

```
holo = hp.core.update_metadata(holo, medium_index=1.33, illum_wavelen=0.660, illum_
↪polarization=(1,0))
```

Note: Spacing and wavelength must be given in the same units - micrometers in the example above. Holopy has no built-in length scale and requires only that you be consistent. For example, we could have specified both parameters in terms of nanometers or meters instead.

HoloPy images are stored as xarray DataArray objects. xarray is a powerful tool that makes it easy to keep track of metadata and extra image dimensions, distinguishing between a time slice and a volume slice, for example. While you do not need any knowledge of xarray to use HoloPy, some familiarity will make certain tasks easier. This is especially true if you want to directly manipulate data before or after applying HoloPy's built-in functions.

2.2.4 Saving and Reloading Holograms

Once you have background-divided a hologram and associated it with metadata, you might want to save it so that you can skip those steps next time you are working with the same image:

```
hp.save('outfilename', holo)
```

saves your processed image to a compact HDF5 file. In fact, you can use `save()` on any holopy object. To reload your same hologram with metadata you would write:

```
holo = hp.load('outfilename')
```

If you would like to save your hologram to an image format for easy visualization, use:

```
hp.save_image('outfilename', holo)
```

Additional options of `save_image()` allow you to control how image intensity is scaled. Although HoloPy stores metadata when writing to .tif image files, you should save holograms in HDF5 format using `save()` to avoid rounding.

2.3 Reconstructing Data (Numerical Propagation)

A hologram contains information about the electric field amplitude and phase at the detector plane. Shining light back through a hologram allows reconstruction of the electric field at points upstream of the detector plane. HoloPy performs this function mathematically by numerically propagating a hologram (or electric field) to another position in space.

This allows you to reconstruct 3D sample volumes from 2D images. The light source is assumed to be collimated here, but HoloPy is also capable of *Reconstructing Point Source Holograms*.

2.3.1 Example Reconstruction

```
import numpy as np
import holopy as hp
from holopy.core.io import get_example_data_path, load_average
from holopy.core.process import bg_correct

imagepath = get_example_data_path('image01.jpg')
raw_holo = hp.load_image(imagepath, spacing = 0.0851, medium_index = 1.33, illum_
↪wavelen = 0.66, )
bgpath = get_example_data_path(['bg01.jpg','bg02.jpg','bg03.jpg'])
bg = load_average(bgpath, refimg = raw_holo)
holo = bg_correct(raw_holo, bg)

zstack = np.linspace(0, 20, 11)
rec_vol = hp.propagate(holo, zstack)
hp.show(rec_vol)
```

We'll examine each section of code in turn. The first block:

```
import numpy as np
import holopy as hp
from holopy.core.io import get_example_data_path, load_average
from holopy.core.process import bg_correct
```

loads the relevant modules from HoloPy and NumPy. The second block:

```
imagepath = get_example_data_path('image01.jpg')
raw_holo = hp.load_image(imagepath, spacing = 0.0851, medium_index = 1.33, illum_
↪wavelen = 0.66)
bgpath = get_example_data_path(['bg01.jpg','bg02.jpg','bg03.jpg'])
bg = load_average(bgpath, refimg = raw_holo)
holo = bg_correct(raw_holo, bg)
```

reads in a hologram and divides it by a corresponding background image. If this is unfamiliar to you, please review the *Loading Data* tutorial.

Next, we use numpy's linspace to define a set of distances between the image plane and the reconstruction plane at 2-micron intervals to propagate our image to. You can also propagate to a single distance or to a set of distances obtained in some other fashion. The actual propagation is accomplished with `propagate()`:

```
zstack = np.linspace(0, 20, 11)
rec_vol = hp.propagate(holo, zstack)
```

Here, HoloPy has projected the hologram image through space to each of the distances contained in `zstack` by using the metadata that we specified when loading the image. If we forgot to load optical metadata with the image, we can explicitly indicate the parameters for propagation to obtain an identical object:

```
rec_vol = hp.propagate(holo, zstack, illum_wavelen = 0.660, medium_index = 1.33)
```

2.3.2 Visualizing Reconstructions

You can display the reconstruction with `show()`:

```
hp.show(rec_vol)
```

Pressing the left and right arrow keys steps through volumes slices - propagation to different z-planes. If the left and right arrow keys don't do anything, you might need to set your matplotlib backend. Refer to *Using HoloPy* for instructions.

Reconstructions are actually comprised of complex numbers. `show()` defaults to showing you the amplitude of the image. You can get different, and sometimes better, contrast by viewing the phase angle or imaginary part of the reconstruction:

```
hp.show(rec_vol.imag)
hp.show(np.angle(rec_vol))
```

These phase sensitive visualizations will change contrast as you step through because you hit different places in the phase period. Such a reconstruction will work better if you use steps that are an integer number of wavelengths in medium:

```
med_wavelen = holo.illum_wavelen / holo.medium_index
rec_vol = hp.propagate(holo, zstack*med_wavelen)
hp.show(rec_vol.imag)
```

2.3.3 Cascaded Free Space Propagation

HoloPy calculates reconstructions by performing a convolution of the hologram with the reference wave over the distance to be propagated. By default, HoloPy calculates a single transfer function to perform the convolution over the specified distance. However, a better reconstruction can sometimes be obtained by iteratively propagating the hologram over short distances. This cascaded free space propagation is particularly useful when the reconstructions have fine features or when propagating over large distances. For further details, refer to Kreis 2002.

To implement cascaded free space propagation in HoloPy, pass a `cfsp` argument into `propagate()` indicating how many times the hologram should be iteratively propagated. For example, to propagate in three steps over each distance, we write:

```
rec_vol = hp.propagate(holo, zstack, cfsp = 3)
```

2.4 Reconstructing Point Source Holograms

Holograms are typically reconstructed optically by shining light back through them. This corresponds mathematically to propagating the field stored in the hologram to some different plane. The propagation performed here assumes that the hologram was recorded using a point source (diverging spherical wave) as the light source. This is also known as lens-free holography. Note that this is different than propagation calculations where a collimated light source (plane wave) is used. For recontructions using a plane wave see *Reconstructing Data (Numerical Propagation)*.

This point-source propagation calculation is an implementation of the algorithm that appears in Jericho and Kreuzer 2010. Curently, only square input images and propagation through media with a refractive index of 1 are supported.

2.4.1 Example Reconstruction

```
import holopy as hp
import numpy as np
from holopy.core.io import get_example_data_path
from holopy.propagation import ps_propagate
from scipy.ndimage.measurements import center_of_mass

imagepath = get_example_data_path('ps_image01.jpg')
bgpath = get_example_data_path('ps_bg01.jpg')
```

(continues on next page)

(continued from previous page)

```
L = 0.0407 # distance from light source to screen/camera
cam_spacing = 12e-6 # linear size of camera pixels
mag = 9.0 # magnification
npix_out = 1020 # linear size of output image (pixels)
zstack = np.arange(1.08e-3, 1.18e-3, 0.01e-3) # distances from camera to reconstruct

holo = hp.load_image(imagepath, spacing=cam_spacing, illum_wavelen=406e-9, medium_
↪index=1) # load hologram
bg = hp.load_image(bgpath, spacing=cam_spacing) # load background image
holo = hp.core.process.bg_correct(holo, bg+1, bg) # subtract background (not divide)
beam_c = center_of_mass(bg.values.squeeze()) # get beam center
out_schema = hp.core.detector_grid(shape=npix_out, spacing=cam_spacing/mag) # set␣
↪output shape

recons = ps_propagate(holo, zstack, L, beam_c, out_schema) # do propagation
hp.show(abs(recons[:,350:550,450:650])) # display result
```

We'll examine each section of code in turn. The first block:

```
import holopy as hp
import numpy as np
from holopy.core.io import get_example_data_path
from holopy.propagation import ps_propagate
from scipy.ndimage.measurements import center_of_mass
```

loads the relevant modules. The second block:

```
imagepath = get_example_data_path('ps_image01.jpg')
bgpath = get_example_data_path('ps_bg01.jpg')
L = 0.0407 # distance from light source to screen/camera
cam_spacing = 12e-6 # linear size of camera pixels
mag = 9.0 # magnification
npix_out = 1020 # linear size of output image (pixels)
zstack = np.arange(1.08e-3, 1.18e-3, 0.01e-3) # distances from camera to reconstruct
```

defines all parameters used for the reconstruction. Numpy's linspace was used to define a set of distances at 10-micron intervals to propagate our image to. You can also propagate to a single distance or to a set of distances obtained in some other fashion. The third block:

```
holo = hp.load_image(imagepath, spacing=cam_spacing, illum_wavelen=406e-9, medium_
↪index=1) # load hologram
bg = hp.load_image(bgpath, spacing=cam_spacing) # load background image
holo = hp.core.process.bg_correct(holo, bg+1, bg) # subtract background (not divide)
beam_c = center_of_mass(bg.values.squeeze()) # get beam center
out_schema = hp.core.detector_grid(shape=npix_out, spacing=cam_spacing/mag) # set␣
↪output shape
```

reads in a hologram and subtracts the corresponding background image. If this is unfamiliar to you, please review the *Loading Data* tutorial. The third block also finds the center of the reference beam and sets the size and pixel spacing of the output images.

Finally, the actual propagation is accomplished with *ps_propagate()* and a cropped region of the result is displayed. See the *Reconstructing Data (Numerical Propagation)* page for details on visualizing the reconstruction results.

```
recons = ps_propagate(holo, zstack, L, beam_c, out_schema) # do propagation
hp.show(abs(recons[:,350:550,450:650])) # display result
```

2.4.2 Magnification and Output Image Size

Unlike the case where a collimated beam is used as the illumination and the pixel spacing in the reconstruction is the same as in the original hologram, for lens-free reconstructions the pixel spacing in the reconstruction can be chosen arbitrarily. In order to magnify the reconstruction the spacing in the reconstruction plane should be smaller than spacing in the original hologram. In the code above, the magnification of the reconstruction can be set using the variable `mag`, or when calling *`ps_propagate()`* directly the desired pixel spacing in the reconstruction is specified through the spacing of `out_schema`. Note that the output spacing will not be the spacing of `out_schema` exactly, but should be within a few percent of it. We recommend calling *`get_spacing()`* on `recons` to get the actual spacing used.

Note that the total physical size of the plane that is reconstructed remains the same when different output pixel spacings are used. This means that reconstructions with large output spacings will only have a small number of pixels, and reconstructions with small output spacings will have a large number of pixels. If the linear size (in pixels) of the total reconstruction plane is smaller than `npix_out`, the entire reconstruction plane will be returned. However, if the linear size of total reconstruction plane is larger than `npix_out`, only the center region of the reconstruction plane with linear size `npix_out` is returned.

In the current version of the code, the amount of memory needed to perform a reconstruction scales with `mag`^{2}. Presumably this limitation can be overcome by implementing the steps described in the *Convolution* section of the *Appendix* of Jericho and Kreuzer 2010.

2.5 Scattering Calculations

Optical physicists and astronomers have worked out how to compute the scattering of light from many kinds of objects. HoloPy provides an easy interface for computing scattered fields, intensities, scattering matrices, cross-sections, and holograms generated by microscopic objects. (Computing scattering from macroscopic objects is computationally difficult and is not well-supported in HoloPy.)

2.5.1 A Simple Example

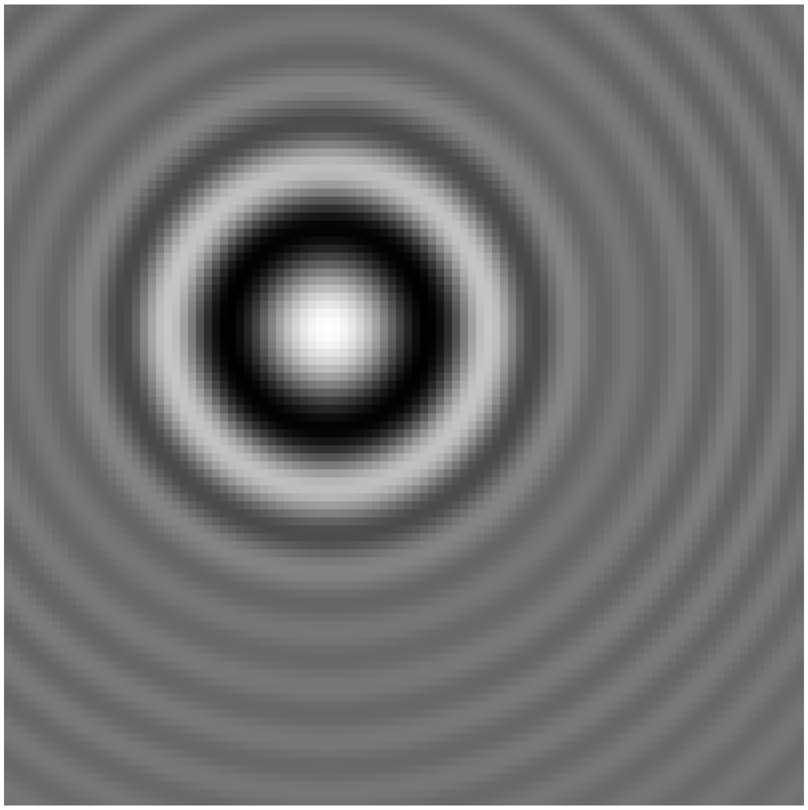
Let's start by calculating an in-line hologram generated by a plane wave scattering from a microsphere.

```
import holopy as hp
from holopy.scattering import calc_holo, Sphere


sphere = Sphere(n=1.59, r=0.5, center=(4, 4, 5))

medium_index = 1.33
illum_wavelen = 0.660
illum_polarization = (1, 0)
detector = hp.detector_grid(shape=100, spacing=0.1)

holo = calc_holo(detector, sphere, medium_index, illum_wavelen,
                 illum_polarization, theory='auto')
hp.show(holo)
```



(You may need to call `matplotlib.pyplot.show()` if you can't see the hologram after running this code.)

To calculate a hologram, HoloPy needs to know two things: the *scatterer* that is scattering the light and the *experimental setup* under which the hologram is recorded. With those two, HoloPy chooses an appropriate *scattering theory* that calculates the hologram from the scatterer and the experimental setup; advanced users may want to choose the theory themselves. We'll examine each section of code in turn.

The first few lines :

```
import holopy as hp
from holopy.scattering import calc_holo, Sphere
```

load the relevant modules from HoloPy that we'll need for doing our calculation.

The next line describes the *scatterer* we would like to model:

```
sphere = Sphere(n=1.59, r=0.5, center=(4, 4, 5))
```

Scatterers are described in HoloPy by a `Scatterer` object. Here, we use a `Sphere` as the scatterer object. A `Scatterer` object contains information about the geometry (position, size, shape) and optical properties (refractive index) of the object that is scattering light. We've defined a spherical scatterer with radius 0.5 microns and index of refraction 1.59. This refractive index is approximately that of polystyrene.

Next, we need to describe the *experimental setup*, including how we are illuminating our sphere, and how that light will be detected:

```
medium_index = 1.33
illum_wavelen = 0.66
illum_polarization = (1, 0)
detector = hp.detector_grid(shape=100, spacing=0.1)
```

We are going to be using red light (wavelength = 660 nm in vacuum) polarized in the x-direction to illuminate a sphere immersed in water (refractive index = 1.33). Refer to *Units* and *Coordinate System* if you're confused about how the wavelength and polarization are specified.

The scattered light will be collected at a detector, which is frequently a digital camera mounted onto a microscope. We defined our detector as a 100 x 100 pixel array, with each square pixel of side length .1 microns. The `shape` argument tells HoloPy how many pixels are in the detector and affects computation time. The `spacing` argument tells HoloPy how far apart each pixel is. Both parameters affect the absolute size of the detector.

Finally, we need to specify the *scattering theory* which knows how to calculate the hologram from the experimental setup and the scatterer. By setting `theory='auto'`, we let HoloPy automatically select a theory. If no theory is specified, HoloPy will automatically select a theory as well.

After getting everything ready, the actual scattering calculation is straightforward:

```
holo = calc_holo(detector, sphere, medium_index, illum_wavelen,
                 illum_polarization, theory='auto')
hp.show(holo)
```

Congratulations! You just calculated the in-line hologram generated at the detector plane by interference between the scattered field and the reference wave. For an in-line hologram, the reference wave is simply the part of the field that is not scattered or absorbed by the particle.

You might have noticed that our scattering calculation requires much of the same metadata we specified when loading an image. If we have an experimental image from the system we would like to model, we can use that as an argument in `calc_holo()` instead of our `detector` object created from `detector_grid()`. HoloPy will calculate a hologram image with pixels at the same positions as the experimental image, and so we don't need to worry about making a `detector_grid()` with the correct `shape` and `spacing` arguments.

```
from holopy.core.io import get_example_data_path
imagepath = get_example_data_path('image0002.h5')
exp_img = hp.load(imagepath)
holo = calc_holo(exp_img, sphere)
```

Note that we didn't need to explicitly specify illumination information when calling `calc_holo()`, since our image contained saved metadata and HoloPy used its values. Passing an image to a scattering function is particularly useful when comparing simulated data to experimental results, since we can easily recreate our experimental conditions exactly.

So far all of the images we have calculated are holograms, or the interference pattern that results from the superposition of a scattered wave with a reference wave. Holopy can also be used to examine scattered fields on their own. Simply replace `calc_holo()` with `calc_field()` to look at scattered electric fields (complex) or `calc_intensity()` to look at field amplitudes, which is the typical measurement in a light scattering experiment.

2.5.2 More Complex Scatterers

Let's proceed to a few examples with different `Scatterer` objects. You can find a more thorough desccription of all their functionalities in the user guide on *The HoloPy Scatterer*.

Coated Spheres

HoloPy can also calculate holograms from coated (or multilayered) spheres. Constructing a coated sphere differs only in specifying a list of refractive indices and outer radii corresponding to the layers (starting from the core and working outwards).

```
coated_sphere = Sphere(center=(2.5, 5, 5), n=(1.59, 1.42), r=(0.3, 0.6))
holo = calc_holo(exp_img, coated_sphere)
hp.show(holo)
```

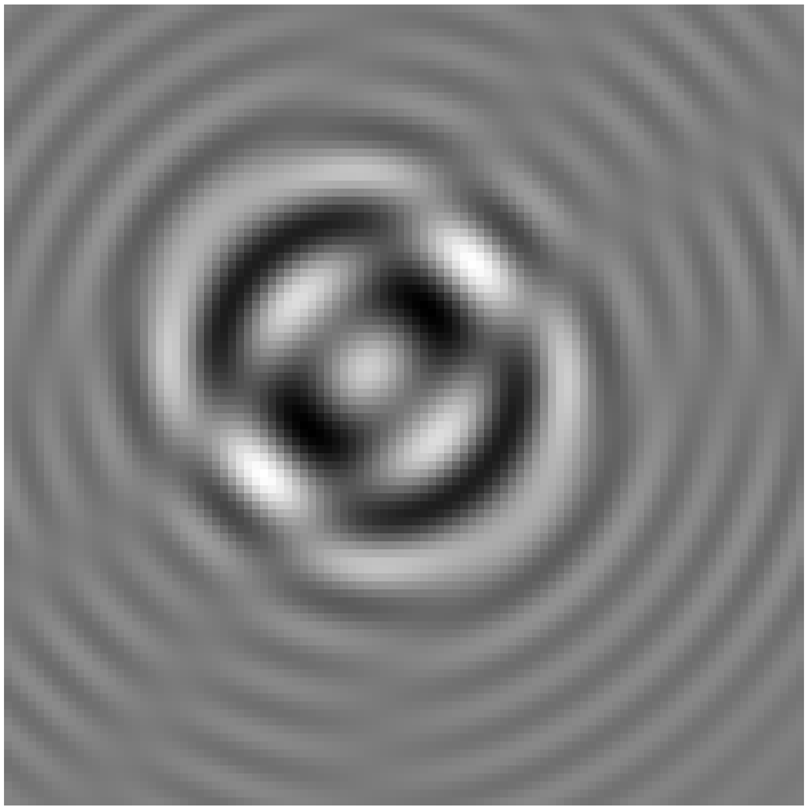
If you prefer thinking in terms of the thickness of subsequent layers, instead of their distance from the center, you can use *LayeredSphere* to achieve the same result:

```
from holopy.scattering import LayeredSphere
coated_sphere = LayeredSphere(center=(2.5, 5, 5), n=(1.59, 1.42), t=(0.3, 0.3))
```

Collection of Spheres

If we want to calculate a hologram from a collection of spheres, we must first define the spheres individually, and then combine them into a *Spheres* object:

```
from holopy.scattering import Spheres
s1 = Sphere(center=(5, 5, 5), n = 1.59, r = .5)
s2 = Sphere(center=(4, 4, 5), n = 1.59, r = .5)
collection = Spheres([s1, s2])
holo = calc_holo(exp_img, collection)
hp.show(holo)
```

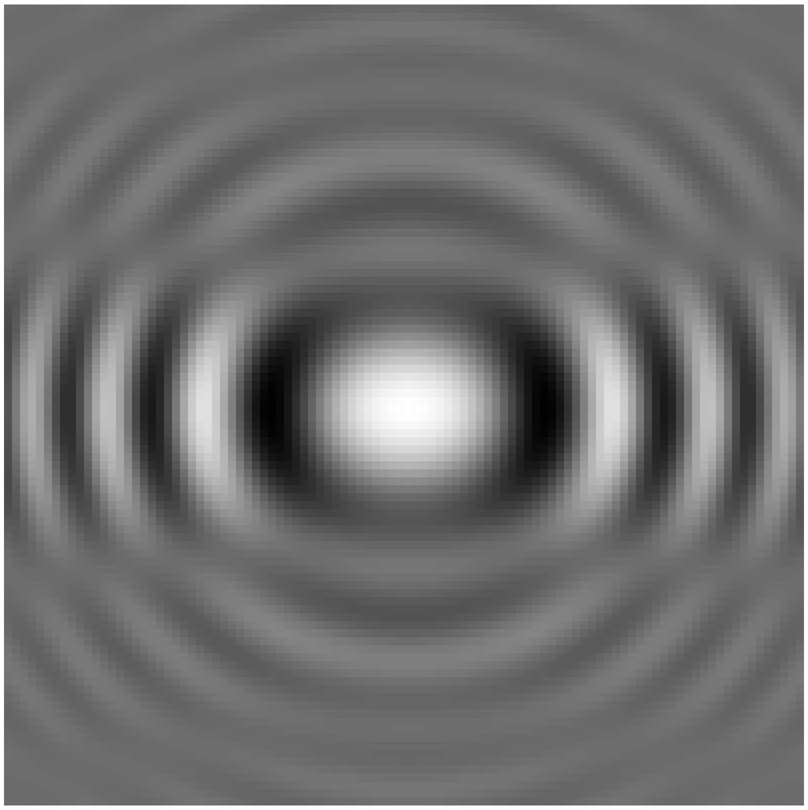


Adding more spheres to the cluster is as simple as defining more sphere objects and passing a longer list of spheres to the *Spheres* constructor.

Non-spherical Objects

To define a non-spherical scatterer, use `Spheroid` or `Cylinder` objects. These axisymmetric scatterers are defined by two dimensions, and can describe scatterers that are elongated or squashed along one direction. By default, these objects are aligned with the z-axis, but they can be rotated into any orientation by passing a set of Euler angles to the `rotation` argument when defining the scatterer. See *Rotations of Scatterers* for information on how these angles are defined. As an example, here is a hologram produced by a cylinder aligned with the vertical axis (`x-axis` according to the HoloPy *Coordinate System*). Note that the hologram image is elongated in the horizontal direction since the sides of the cylinder scatter light more than the ends.

```
import numpy as np
from holopy.scattering import Cylinder
c = Cylinder(center=(5, 5, 7), n = 1.59, d=0.75, h=2, rotation=(0,np.pi/2, 0))
holo = calc_holo(exp_img, c)
hp.show(holo)
```



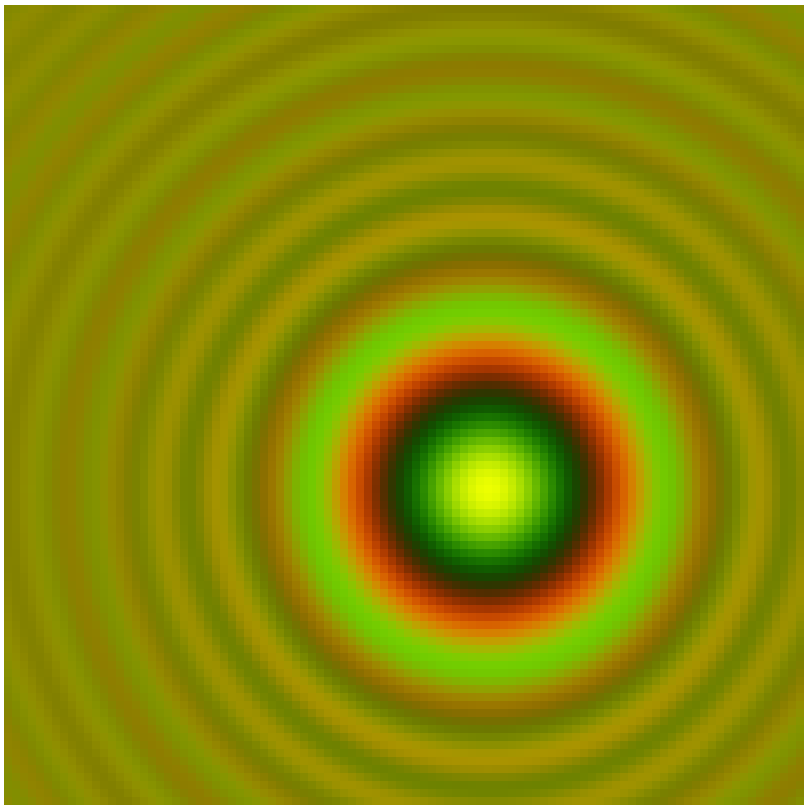
2.5.3 More Complex Experimental Setups

While the examples above will be sufficient for most purposes, there are a few additional options that are useful in certain scenarios.

Multi-channel Holograms

Sometimes a hologram may include data from multiple illumination sources, such as two separate wavelengths of incident light. In this case, the extra arguments can be passed in as a dictionary object, with keys corresponding to dimension names in the image. You can also use a multi-channel experimental image in place of calling `detector_grid()`.

```
illum_dim = {'illumination':['red', 'green']}
n_dict = {'red':1.58,'green':1.60}
wl_dict = {'red':0.690,'green':0.520}
det_c = hp.detector_grid(shape=200, spacing=0.1, extra_dims = illum_dim)
s_c = Sphere(r=0.6, n=n_dict, center=[6,6,6])
holo = calc_holo(det_c, s_c, illum_wavelen=wl_dict, illum_polarization=(0,1), medium_
↪index=1.33)
```



Scattering Theories in HoloPy

HoloPy contains a number of scattering theories to model the scattering from different kinds of scatterers. You can specifiy a scattering theory by setting the `theory` keyword to a *`ScatteringTheory`* object, rather than setting the theory to `'auto'`. For instance, to force HoloPy to calculate the hologram of a sphere using Mie theory (the theory which exactly describes scattering from a spherical particle), we set the `theory` keyword to an instance of the *`Mie`* class:

```
from holopy.scattering.theory import Mie
theory = Mie()
holo = calc_holo(detector, sphere, medium_index, illum_wavelen,
                 illum_polarization, theory=theory)
```

HoloPy has multiple scattering theories which work for different types of scatterers and which describe particle scattering and interactions with the optical train in varying degrees of complexity. HoloPy has scattering theories that describe scattering from individual spheres, layered spheres, clusters of spheres, spheroids, cylinders, and arbitrary objects. Some of these scattering theories can take parameters to modify how the theory performs the calculation (by, *e.g.*, making certain approximations or specifying properties of the optical train). For a more thorough description of these scattering theories and how HoloPy chooses default scattering theories, see the user guide, *The HoloPy Scattering Theories*.

Detector Types in HoloPy

The `detector_grid()` function we saw earlier creates holograms that display nicely and are easily compared to experimental images. However, they can be computationally expensive, as they require calculations of the electric field at many points. If you only need to calculate values at a few points, or if your points of interest are not arranged in a 2D grid, you can use `detector_points()`, which accepts either a dictionary of coordinates or indvidual coordinate dimensions:

```
x = [0, 1, 0, 1, 2]
y = [0, 0, 1, 1, 1]
z = -1
coord_dict = {'x': x, 'y': y, 'z': z}
detector = hp.detector_points(x = x, y = y, z = z)
detector = hp.detector_points(coord_dict)
```

The coordinates for `detector_points()` can be specified in terms of either Cartesian or spherical coordinates. If spherical coordinates are used, the `center` value of your scatterer is ignored and the coordinates are interpreted as being relative to the scatterer.

2.5.4 Static light scattering calculations

Scattering Matrices

In a static light scattering measurement you record the scattered intensity at a number of locations. A common experimental setup contains multiple detectors at a constant radial distance from a sample (or a single detector on a goniometer arm that can swing to multiple angles.) In this kind of experiment you are usually assuming that the detector is far enough away from the particles that the far-field approximation is valid, and you are usually not interested in the exact distance of the detector from the particles. So, it's most convenient to work with amplitude scattering matrices that are angle-dependent. (See [Bohren1983] for further mathematical description.)

```
import numpy as np
from holopy.scattering import calc_scat_matrix

detector = hp.detector_points(theta = np.linspace(0, np.pi, 100), phi = 0)
distant_sphere = Sphere(r=0.5, n=1.59)
matr = calc_scat_matrix(detector, distant_sphere, medium_index, illum_wavelen)
```

Here we omit specifying the location (center) of the scatterer. This is only valid when you're calculating a far-field quantity. Similarly, note that our detector, defined from a `detector_points()` function, includes information about direction but not distance. It is typical to look at scattering matrices on a semilog plot. You can make one as follows:

```
import matplotlib.pyplot as plt
plt.figure()
plt.semilogy(np.linspace(0, np.pi, 100), abs(matr[:,0,0])**2)
plt.semilogy(np.linspace(0, np.pi, 100), abs(matr[:,1,1])**2)
plt.show()
```

You are usually interested in the intensities of the scattered fields, which are proportional to the modulus squared of the amplitude scattering matrix. The diagonal elements give the intensities for the incident light and the scattered light both polarized parallel and perpendicular to the scattering plane, respectively.

Scattering Cross-Sections

The scattering cross section provides a measure of how much light from an incident beam is scattered by a particular scatterer. Similar to calculating scattering matrices, we can omit the position of the scatterer for calculation of cross sections. Since cross sections integrates over all angles, we can also omit the `detector` argument entirely:

```
from holopy.scattering import calc_cross_sections
x_sec = calc_cross_sections(distant_sphere, medium_index, illum_wavelen, illum_
↪polarization)
```

x_sec returns an array containing four elements. The first element is the scattering cross section, specified in terms of the same units as wavelength and particle size. The second and third elements are the absorption and extinction cross sections, respectively. The final element is the average value of the cosine of the scattering angle.

2.6 Scattering from Arbitrary Structures with DDA

The discrete dipole approximation (DDA) lets us calculate scattering from any arbitrary object by representing it as a closely packed array of point dipoles. In HoloPy you can make use of the DDA by specifying a general *`Scatterer`* with an indicator function (or set of functions for a composite scatterer containing multiple media).

HoloPy uses ADDA to do the actual DDA calculations, so you will need to install ADDA and be able to run:

```
adda
```

at a terminal for HoloPy DDA calculations to succeed. To install ADDA, first download or clone the code from GitHub. In a terminal window, go to the directory 'adda/src' and compile using one of three options:

```
make seq
```

or:

```
make
```

or:

```
make OpenCL
```

`make seq` will not take advantage of any parallel processing of the cores on your computer. `make` uses mpi for parallel processing. `make OpenCL` uses OpenCL for parallel processing. If the make does not work due to missing packages, you will have to download the specified packages and install them.

Next, you must modify your path in your .bashrc or /bash_profile (for mac). Add the line:

```
export PATH=$PATH:userpath/adda/src/seq
```

or:

```
export PATH=$PATH:userpath/adda/src/mpi
```

or:

```
export PATH=$PATH:userpath/adda/src/OpenCL
```

where you should use the path that matches the make you chose above.

A lot of the code associated with DDA is fairly new so be careful; there are probably bugs. If you find any, please report them.

2.6.1 Defining the geometry of the scatterer

To calculate the scattering pattern for an arbitrary object, you first need an indicator function which outputs 'True' if a test coordinate lies within your scatterer, and 'False' if it doesn't. The indicator function is an argument of the constructor of your scatterer.

For example, if you wanted to define a dumbbell consisting of the union of two overlapping spheres you could do so like this:

```
import holopy as hp
from holopy.scattering import Scatterer, Sphere, calc_holo
import numpy as np
s1 = Sphere(r = .5, center = (0, -.4, 0))
s2 = Sphere(r = .5, center = (0, .4, 0))
detector = hp.detector_grid(100, .1)
dumbbell = Scatterer(lambda point: np.logical_or(s1.contains(point), s2.
↪contains(point)),
                     1.59, (5, 5, 5))
holo = calc_holo(detector, dumbbell, medium_index=1.33, illum_wavelen=.66, illum_
↪polarization=(1, 0))
```

Here we take advantage of the fact that Spheres can tell us if a point lies inside them. We use `s1` and `s2` as purely geometrical constructs, so we do not give them indices of refraction, instead specifying n when defining `dumbbell`.

HoloPy contains convenient wrappers for many built-in ADDA constructions. The dumbbell defined explicitly above could also have been defined with the HoloPy *Bisphere* class instead. Similar classes exist to define an *Ellipsoid*, *Cylinder*, or *Capsule*.

2.6.2 Mutiple Materials: A Janus Sphere

You can also provide a set of indicators and indices to define a scatterer containing multiple materials. As an example, lets look at a janus sphere consisting of a plastic sphere with a high index coating on the top half:

```
from holopy.scattering.scatterer import Indicators
import numpy as np
s1 = Sphere(r = .5, center = (0, 0, 0))
s2 = Sphere(r = .51, center = (0, 0, 0))
def cap(point):
    return(np.logical_and(np.logical_and(point[...,2] > 0, s2.contains(point)),
           np.logical_not(s1.contains(point))))
indicators = Indicators([s1.contains, cap],
                        [[-.51, .51], [-.51, .51], [-.51, .51]])
janus = Scatterer(indicators, (1.34, 2.0), (5, 5, 5))
holo = calc_holo(detector, janus, medium_index=1.33, illum_wavelen=.66, illum_
↪polarization=(1, 0))
```

We had to manually set up the bounds of the indicator functions here because the automatic bounds determination routine gets confused by the cap that does not contain the origin.

We also provide a `JanusSphere` scatterer which is very similar to the scatterer defined above, but can also take a rotation angle to specify other orientations:

```
from holopy.scattering import JanusSphere
janus = JanusSphere(n = [1.34, 2.0], r = [.5, .51], rotation = (-np.pi/2, 0),
                    center = (5, 5, 5))
```

2.7 Fitting Models to Data

As we have seen, we can use HoloPy to perform *Scattering Calculations* from many types of objects. Here, the goal is to compare these calculated holograms to a recorded experimental hologram, and adjust the parameters of the simulated scatterer to get a good fit for the real hologram.

2.7.1 A Simple Least Squares Fit

We start by loading and processing data using many of the functions outlined in the tutorial on *Loading Data*.

```
import holopy as hp
import numpy as np
from holopy.core.io import get_example_data_path, load_average
from holopy.core.process import bg_correct, subimage, normalize
from holopy.scattering import Sphere, Spheres, calc_holo
from holopy.inference import prior, ExactModel, CmaStrategy, EmceeStrategy

# load an image
imagepath = get_example_data_path('image01.jpg')
raw_holo = hp.load_image(imagepath, spacing = 0.0851, medium_index = 1.33,
                         illum_wavelen = 0.66, illum_polarization = (1,0))
bgpath = get_example_data_path(['bg01.jpg','bg02.jpg','bg03.jpg'])
bg = load_average(bgpath, refimg = raw_holo)
data_holo = bg_correct(raw_holo, bg)

# process the image
data_holo = subimage(data_holo, [250,250], 200)
data_holo = normalize(data_holo)
```

Next we define a scatterer that we wish to model as our initial guess. We can calculate the hologram that it would produce if it were placed in our experimental setup, as in the previous tutorial on *Scattering Calculations*. Fitting works best if your initial guess is close to the correct result. You can find guesses for *x* and *y* coordinates with `center_find()`, and a guess for *z* with `propagate()`.

```
guess_sphere = Sphere(n=1.58, r=0.5, center=[24,22,15])
initial_guess = calc_holo(data_holo, guess_sphere)
hp.show(data_holo)
hp.show(initial_guess)
```

Finally, we can adjust the parameters of the sphere in order to get a good fit to the data. Here we adjust the center coordinates (x, y, z) of the sphere and its radius, but hold its refractive index fixed. By default `fit()` will adjust all parameters, so we can omit the argument if that is what we want.

```
parameters_to_fit = ['x', 'y', 'z', 'r']
fit_result = hp.fit(data_holo, guess_sphere, parameters=parameters_to_fit)
```

The `fit()` function automatically runs `calc_holo()` on many different sets of parameter values to find the combination that gives the best match to the experimental `data_holo`. We get back a `FitResult` object that knows how to summarize the results of the fitting calculation in various ways, and can be saved to a file with `hp.save`()` :

```
best_fit_values = fit_result.parameters
initial_guess_values = fit_result.guess_parameters
best_fit_sphere = fit_result.scatterer
best_fit_hologram = fit_result.hologram
```

(continues on next page)

(continued from previous page)

```
best_fit_lnprob = fit_result.max_lnprob
hp.save('results_file.h5', fit_result)
```

If we look at `best_fit_values` or `best_fit_sphere`, we see that our initial guess of the sphere's position of (24, 22, 15) was corrected to (24.16, 21.84, 16.35). Note that we have achieved sub-pixel position resolution!

2.7.2 Customizing the model

Sometimes you might want a bit more control over how the parameters are varied. You can customize the parameters with a *`Model`* object that describes parameters as *`Prior`* objects instead of simply passing in your best guess scatterer and the names of the parameters you wish to vary. For example, we can set bounds on the coordinate parameters and use a Gaussian prior for the radius - here, with a mean of 0.5 and standard deviation of 0.05 micrometers.

```
x = prior.Uniform(lower_bound=15, upper_bound=30, guess=24)
y = prior.Uniform(15, 30, 22)
z = prior.Uniform(10, 20)
par_sphere = Sphere(n=1.58, r=prior.Gaussian(0.5, 0.05), center=[x, y, z])
model = ExactModel(scatterer=par_sphere, calc_func=calc_holo)
fit_result = hp.fit(data_holo, model)
```

Here we have used an *`ExactModel`* which takes a function `calc_func` to apply on the *`Scatterer`* (we have used *`calc_holo()`* here). The *`ExactModel`* isn't actually the default when we call *`fit()`* directly. Instead, HoloPy uses an *`AlphaModel`*, which includes an additional fitting parameter to control the hologram contrast intensity - the same as calling *`calc_holo()`* with a *scaling* argument. You can fit for the extra parameters in these models by defining them as *`Prior`* objects. Likewise, if the scattering theory you are using requires fittable parameters (such as the *lens_angle* for the *`MieLens`* theory or the *spherical_aberration* for the *`AberratedMieLens`* theory), you can fit for these by defining them as *`Prior`* objects as well.

The model in our example has read in some metadata from `data_holo` (illumination wavelength & polarization, medium refractive index, and image noise level). If we want to override those values, or if we loaded an image without specifying metadata, we can pass them directly into the *`Model`* object by using keywords when defining it.

Advanced Parameter Specification

You can use the *`Model`* framework to more finely control parameters, such as specifying a complex refractive index :

```
n = prior.ComplexPrior(real=prior.Gaussian(1.58, 0.02), imag=1e-4)
```

When this refractive index is used to define a *`Sphere`*, *`fit()`* will fit to the real part of index of refraction while holding the imaginary part fixed. You could fit it as well by specifying a *`Prior`* for `imag`.

You may desire to fit holograms with *tied parameters*, in which several physical quantities that could be varied independently are constrained to have the same (but non-constant) value. A common example involves fitting a model to a multi-particle hologram in which all of the particles are constrained to have the same refractive index, but the index is determined by the fitter. This may be done by defining a parameter and using it in multiple places. Other tools for handling tied parameters are described in the user guide on *The HoloPy Scatterer*.

```
n1 = prior.Gaussian(1.58, 0.02)
sphere_cluster = Spheres([
Sphere(n = n1, r = 0.5, center = [10., 10., 20.]),
Sphere(n = n1, r = 0.5, center = [9., 11., 21.])])
```

2.7.3 Transforming Priors

Sometimes you might want to apply mathematical operations to transform one prior into another, for example in the following use cases:

- You want two parameters to vary together with values that are related but unequal, such as the length and radius of a cylinder with known aspect ratio, or the z-coordinates of two vertically stacked particles.
- You want a parameter that is distrbuted according to some other distribution, such as a cylinder axis evenly distributed in spherical coordinates or a sphere with uniformly distributed volume (not radius).
- You want to reparamaterize a problem to reduce covariances between fitting parameters, such as finding the separation distance between two closely interacting particles or to find the positions of particles confined to a plane with a slight tilt as compared to the x-y plane.

If you have an explicit transformation function you can use it to define a *TransformedPrior* object, for example to define a polar angle with the appropriate distribution we might do:

```
def uniform2polar(u):
    # we want polar angle \theta distributed according to sin(\theta)
    # Use inverse transform sampling, correct for \theta in [0, pi]
    symmetric_polar = np.arcsin(u)
    return symmetric_polar + np.pi/2
director = prior.Uniform(-1, 1, name='director')
polar_angle = prior.TransformedPrior(uniform2polar, director, name='polar')
```

You can use *TransformedPrior* objects when defining a *Scatterer* just like regular priors. They share some attributes with basic priors as well, such as the *TransformedPrior.sample()* method, but you cannot directly calculate probabilities or log-probabilities of a *TransformedPrior* taking on a particular value.

Besides explicitly defining them, you can also create *TransformedPrior* objects by using numpy ufuncs and built-in operators on priors:

```
sphere_area = prior.Uniform(1, 2, name='base_area')
diameter = np.sqrt(sphere_area / np.pi)
radius = diameter / 2
shell = radius + 0.1
```

All of our derived objects are *TransformedPrior* objects, even though we didn't explicitly define them that way. If we use more than one of `sphere_area`, `diameter`, `radius`, or `shell` in a fitting or inference calculation, they will all be derived from a single parameter (`sphere_area` in this case) even though they take on different values. Note that we could have expressed `shell` in one line if we didn't care about the intermediate values:

```
shell = np.sqrt(prior.Uniform(1, 2, name='base_area')) / 2 + 0.1
shell.name = 'shell'
```

It's always a good idea to assign your priors names when working with *TransformedPrior* objects to keep track of their relationships. Parameter names will be generated if none are provided but they might not be very informative. To help with naming, you can even assign names to *TransformedPrior* objects when using numpy ufuncs!

```
diameter = np.sqrt(sphere_area / np.pi, name='diameter')
```

2.7.4 Bayesian Parameter Estimation

Often, we aren't just interested in the best-fit (MAP) parameter values, but in the full range of parameter values that provide a reasonable fit to an observed hologram. This is best expressed as a Bayesian posterior distribution, which we can sample with a Markov Chain Monte Carlo (MCMC) algorithm. The approach and formalism used by

HoloPy are described in more detail in [Dimiduk2016]. For more information on Bayesian inference in general, see [Gregory2010].

A sampling calculation uses the same model and data as the fitting calculation in the preceding section, but we replace the function *fit()* with *sample()* instead. Note that this calculation without further modifications might take an unreasonably long time! There are some tips on how to speed up the calculation further down on this page.

The *sample()* calculation returns a *SamplingResult* object, which is similar to the *FitResult* returned by *fit()*, but with some additional features. We can access the sampled parameter values and calculated log-probabilities with `SamplingResult.samples` and `SamplingResult.lnprobs`, respectively. Usually, the MCMC samples will take some steps to converge or "burn-in" to a stationary distribution from your initial guess. This is most easily seen in the values of `SamplingResult.lnprobs`, which will rise at first and then fluctuate around a stationary value after having burned in. You can remove the early samples with the built-in method *SamplingResult.burn_in()*, which returns a new *SamplingResult* with only the burned-in samples.

2.7.5 Customizing the algorithm

The *fit()* and *sample()* functions follow algorithms that determine which sets of parameter values to simulate and compare to the experimental data. You can specify a different algorithm by passing a *strategy* keyword into either function. Options for *fit()* currently include the default Levenberg-Marquardt (`strategy="nmpfit"`), as well as cma-es (`strategy="cma"`) and scipy least squares (`strategy="scipy lsq"`). Options for *sample()* include the default without tempering (`strategy="emcee"`), tempering by changing the number of pixels evaluated (`strategy="subset tempering"`), or parallel tempered MCMC (`strategy="parallel tempering"`) [not currently implemented]. You can see the available strategies in your version of HoloPy by calling `hp.inference.available_fit_strategies` or `hp.inference.available_sampling_strategies`.

Each of these algorithms runs with a set of default values, but these may need to be adjusted for your particular situation. For example, you may want to set a random seed, control parallel computations, customize an initial guess, or specify hyperparameters of the algorithm. To use non-default settings, you must define a *Strategy* object for the algorithm you would like to use. You can save the strategy to a file for use in future calculations or modify it during an interactive session.

```
cma_fit_strategy = CmaStrategy(popsize=15, parallel=None)
cma_fit_strategy.seed = 1234
hp.save('cma_strategy_file.h5', cma_fit_strategy)
strategy_result = fit(data_holo, model, cma_fit_strategy)
```

In the example above, we have adjusted the `popsize` hyperparameter of the cma-es algorithm, prevented the calculation from running as a parallel computation, and set a random seed for reproducibility. The calculation returns a *FitResult* object, just like a direct call to *fit()*.

Similarly, we can customize a MCMC computation to sample a posterior by calling *sample()* with a *EmceeStrategy* object. Here we perform a MCMC calculation that uses only 500 pixels from the image and runs 50 walkers each for 2000 samples. We set the initial walker distribution to be one tenth of the prior width. In general, the burn-in time for a MCMC calculation will be reduced if you provide an initial guess position and width that is as close as possible to the eventual posterior distribution. You can use *Model.generate_guess()* to generate an initial sampling to pass in as an initial guess to your *EmceeStrategy* object.

```
nwalkers = 50
initial_guess = model.generate_guess(nwalkers, scaling=0.1)
emcee_strategy = EmceeStrategy(npixels=500, nwalkers=nwalkers,
    nsamples=2000, walker_initial_pos=initial_guess)
hp.save('emcee_strategy_file.h5', emcee_strategy)
emcee_result = hp.sample(data_holo, model, emcee_strategy)
```

Random Subset Fitting

In the most recent example, we evaluated the holograms at the locations of only 500 pixels in the experimental image. This is because a hologram usually contains far more information than is needed to estimate your parameters of interest. You can often get a significantly faster fit with little or no loss in accuracy by fitting to only a random fraction of the pixels in a hologram.

You will want to do some testing to make sure that you still get acceptable answers with your data, but our investigations have shown that you can frequently use random fractions of 0.1 or 0.01 with little effect on your results and gain a speedup of 10x or greater.

2.8 Developer's Guide

2.8.1 Installing HoloPy for Developers

If you are going to hack on holopy, you probably only want to compile the scattering extensions.

For Mac and Linux:

Download or clone the latest version of HoloPy from Git Hub at https://github.com/manoharan-lab/holopy.

Let's say you downloaded or cloned HoloPy to `/home/me/holopy`. Then open a terminal, `cd` to `/home/me/holopy` and run:

```
python setup.py develop
```

This puts the extensions inside the source tree, so that you can work directly from `/home/me/holopy` and have the changes reflected in the version of HoloPy that you import into python.

Note for Mac users: gfortran may put its library in a place python can't find it. If you get errors including something like `can't find /usr/local/libgfortran.3.dynlib` you can symlink them in from your install. You can do this by running:

```
sudo ln -s /usr/local/gfortran/lib/libgfortran.3.dynlib /usr/local/lib
sudo ln -s /usr/local/gfortran/lib/libquadmath.3.dynlib /usr/local/lib
```

For Windows: Installation on Windows is still a work in progress, but we have been able to get HoloPy working on Windows 10 with an AMD64 architecture (64-bit) processor.

1. Install Anaconda with Python 3.6 and make sure it is working.
2. Install the C compiler. It's included in Visual Studio 2015 Community. Make sure it is working with a C helloworld.
3. From now on, make sure any command prompt window invokes the right environment conditions for compiling with VC. To do this, make sure `C:\Program Files (x86)\Microsoft Visual Studio 14.0\VC\vcvarsall.bat` is added to the system path variable. This batch detects your architecture, then runs another batch that sets the path include the directory with the correct version of the VC compiler.
4. Install cython and made sure it works.
5. Install Intel's Fortran compiler. A good place to start is the trial version of Parallel Studio XE. Make sure it is working with a Fortran helloworld.
6. Install mingw32-make, which does not come with Anaconda by default.
7. Download or clone the master branch of HoloPy from https://github.com/manoharan-lab/holopy.

8. Open the command prompt included in Intel's Parallel Studio. Run `holopy/setup.py`. It is necessay to use Intel's Parallel Studio command prompt to avoid compiling errors.
9. Install the following dependencies that don't come with Anaconda:

```
conda install xarray dask netCDF4 bottleneck
conda install -c astropy emcee=2.2.1
```

10. Open an iPython console where holopy is installed and try `import holopy`.

If the above procedure doesn't work, or you find something else that does, please let us know so that we can improve these instructions.

2.8.2 How HoloPy Stores Data

Images in HoloPy are stored in the format of xarray DataArrays. Spatial information is tracked in the DataArray's `dims` and `coords` fields according to the HoloPy *Coordinate System*. Additional dimensions are sometimes specified to account for different z-slices, times, or field components, for example. Optical parameters like refractive index and illumination wavelength are stored in the DataArray's `attrs` field.

The *`detector_grid()`* function simply creates a 2D image composed entirely of zeros. In contrast, the *`detector_points()`* function creates a DataArray with a single dimension named 'point'. Spatial coordinates (in either Cartesian or spherical form) track this dimension, so that each data value in the array has its own set of coordinates unrelated to its neighbours. This type of one-dimensional organization is sometimes used for 2D images as well. Inference and fitting methods typically use only a subset of points in an image (see random_subset), and so it makes sense for them to keep track of lists of location coordinates instead of a grid. Furthermore, HoloPy's scattering functions accept coordinates in the form of a 3xN array of coordinates. In both of these cases, the 2D image is flattened into a 1D DataArray like that created by *`detector_points()`*. In this case the single dimension is 'flat' instead of 'point'. HoloPy treats arrays with these two named dimensions identically, except that the 'flat' dimension can be unstacked to restore a 2D image or 3D volume.

HoloPy's use of DataArrays sometimes assigns smaller DataArrays in `attrs`, which can lead to problems when saving data to a file. When saving a DataArray to file, HoloPy converts any DataArrays in `attrs` to numpy arrays, and keeps track of their dimension names separately. HoloPy's *`save_image()`* writes a yaml dump of `attrs` (along with spacing information) to the `imagedescription` field of .tif file metadata.

infer_tutorial returns a lot of information, which is stored in the form of a *`SamplingResult`* object. This object stores the model and *`EmceeStrategy`* that were used in the inference calculation as attributes. An additional attribute named `dataset` is an xarray Dataset that contains both the data used in the inference calculation, as well as the raw output. The parameter values at each step of the sampling chain and the calculated log-probabilities at each step are stored here under the `samples` and `lnprobs` namespaces.

2.8.3 Adding a new scattering theory

Adding a new scattering theory is relatively straightforward. You just need to define a new scattering theory class and implement one or two methods to compute the raw scattering values:

```
class YourTheory(ScatteringTheory):
  def can_handle(self, scatterer):
    # Your code here

  def raw_fields(self, positions, scatterer, medium_wavevec, medium_index, illum_
↪polarization):
    # Your code here
```

(continues on next page)

(continued from previous page)

```
  def raw_scat_matrs(self, scatterer, pos, medium_wavevec, medium_index):
    # Your code here

  def raw_cross_sections(self, scatterer, medium_wavevec, medium_index, illum_
↪polarization):
    # Your code here
```

You can get away with just defining one of either `raw_scat_matrs` or `raw_fields` if you just want holograms, fields, or intensities. If you want scattering matrices you will need to implement `raw_scat_matrs`, and if you want cross sections, you will need to implement `raw_cross_sections`. We separate out `raw_fields` from `raw_scat_matrs` to allow for faster fields calculation for specific cases, such as the Mie, MieLens, and Multisphere theories (and you might want to do so for your theory as well); the base *`ScatteringTheory`* class calculates the fields from the scattering matrices by default.

You can look at the Mie theory in HoloPy for an example of calling Fortran functions to compute scattering (C functions will look similar from the python side) or DDA for an an example of calling out to an external command line tool by generating files and reading output files.

If you want to fit parameters in your scattering theory, you also need to define a class attribute *parameter_names* that contains the fittable attributes of the scattering theory. Once you do this, fitting should work natively with your new scattering theory: you should be able to specify the parameters as a `prior.Prior` object and *holopy*'s inference `Model` will auto-detect them as fittable parameters. For an example of this, see the *`Lens`*, *`MieLens`*, or *`AberratedMieLens`* classes.

2.8.4 Adding a new inference model

To perform inference, you need a noise model. You can make a new noise model by inheriting from `NoiseModel`. This class has all the machinery to compute likelihoods of observing data given some set of parameters and assuming Gaussian noise.

To implement a new model, you just need to implement one function: `forward`. This function receives a dictionary of parameter values and a data shape schema (defined by *`detector_grid()`*, for example) and needs to return simulated data of shape specified. See the `_forward` function in `AlphaModel` for an example of how to do this.

If you want to use some other noise model, you may need to override _lnlike and define the probablity given your uncertainty. You can reference _lnlike in `NoiseModel`.

2.8.5 Running Tests

HoloPy comes with a suite of tests that ensure everything has been built correctly and that it's able to perform all of the calculations it is designed to do. To run these tests, navigate to the root of the package (e.g. `/home/me/holopy`) and run:

```
python run_nose.py
```

CHAPTER 3

User Guide

Skip to the *Loading Data* tutorial if you already have HoloPy installed and want to get started quickly.

3.1 The HoloPy Scatterer

The HoloPy `Scatterer` class defines objects that are described by numerical quantities (e.g. dimension, location, refractive index) and have known light-scattering behaviour described by a `ScatteringTheory`.

Scatterers are generally used in two scenarios:

- All numerical properties (e.g. dimension, location, refractive index) are fixed to simulate a specific light scattering experiment.
- Some numerical properties are defined as `Prior` objects, representing unknown values to be determined in an inference calculation.

You can find examples of these use cases in the *Scattering Calculations* and *Fitting Models to Data* tutorials.

Scatterer objects in HoloPy are inherited from two base classes:

- *CenteredScatterer* describes a single object, with an optional location specified
- *Scatterers* describes a collection of individual scatterers

3.1.1 Scatterer Attributes

All HoloPy Scatterer classes have the following properties/methods:

General manipulation

- `x`, `y`, `z` Components of scatterer center
- `translated()` New scatterer with location coordinates shifted by a vector

Inference calculations

- `parameters` Dictionary of all values needed to describe the scatterer. Values described as `Prior` objects will appear that way here as well.
- `from_parameters()` New scatterer built from a dictionary of parameters

Discretization

- `indicators` Function(s) to describe region(s) of space occupied by scatterer domain(s)
- `index_at()` Scatterer's refractive index at given coordinates
- `in_domain()` Which domain of the scatterer the given coordinates are in
- `contains()` Check whether a particular point is in any domains of the scatterer
- `num_domains` Number of domains of the scatterer
- `bounds` Extents of the scatterer in each dimension
- `voxelate()` 3D voxel grid representation of the scatterer containing its refractive index at each point

3.1.2 Individual Scatterers

`CenteredScatterer` objects are not instantiated directly, but instead in one of the subclasses:

- `Sphere` Can contain multiple concentric layers defined by their outer radius
- `LayeredSphere` Defines multiple concentric layers by their layer thickness
- `Cylinder`
- `Ellipsoid`
- `Spheroid`
- `Bisphere` Union of two spheres
- `Capsule` Cylinder with semi-spherical caps on either end
- `JanusSphere_Uniform` Sphere with a semi-spherical outer layer of constant thickness
- `JanusSphere_Tapered` Sphere with a semi-spherical outer layer that has a crescent profile
- `CsgScatterer` Allows for construction of an arbitrary scatterer by constructive solid geometry

3.1.3 Composite Scatterers

`Scatterers` objects contain multiple individual scatterers,and support the following features in addition to those shared with `CenteredScatterer`:

Component scatterer handling

- Support for selecting component scatterers with square brackets and python slicing syntax
- `add()` Adds a new scatterer to the composite in-place
- `rotated()` New scatterer rotated about its center according to *HoloPy rotation conventions*

There are two specific composite scatterer classes for working with collections of spheres that have additional functionality:

`Spheres` A collection of spherical scatterers, with the following properties:

- `overlaps` List of pairs of component spheres that overlap
- `largest_overlap` Maximum overlap distance between component spheres

`RigidCluster` A collection of spherical scatterers in fixed relative positions. The entire cluster can be translated and/or rotated. `RigidCluster.scatterers` and `RigidCluster.from_parameters()` both return `Spheres` type objects.

3.2 The HoloPy Scattering Theories

The HoloPy `ScatteringTheory` classes know how to calculate scattered fields from detector and scatterer information. Each scattering theory is only able to work with certain specific scatterers.

There are two broad classes of scattering theories in HoloPy: *lens-free* theories which treat the recorded fields as the magnified image of the fields at the focal plane, and *lens-based* theories which use a more detailed description of the effects of the objective lens. The lens-free theories usually do not need any additional parameters specified, whereas the lens theories need the lens's acceptance angle, which can be specified as either a fixed number or a `Prior` object, representing an unknown value to be determined in an inference calculation.

All scattering theories in HoloPy inherit from the `ScatteringTheory` class.

Not sure how to choose a scattering theory? See the *Which Scattering Theory should I use?* section.

3.2.1 ScatteringTheory Methods

HoloPy Scattering theories calculate the scattered fields through one of the following methods.

- `_raw_fields()` Calculates the scattering fields.
- `_raw_scat_matrs()` Calculates the scattering matrices.
- `_raw_cross_sections()` Calculates the cross sections.
- `_can_handle()` Checks if the theory is compatible with a given scatterer.

If a theory is asked for the raw fields, but does not have a `_raw_fields` method, the scattering theory attempts to calculate them via the scattering matrices, as called by `_raw_scat_matrs`. More than one of these methods may be implemented for performance reasons.

Be advised that the `ScatteringTheory` class is under active development, and these method names may change.

3.2.2 Lens-Free Scattering Theories

- **`DDA`**
    - Can handle every `Scatterer` object in HoloPy
    - Computes scattered fields using the discrete dipole approximation, as implemented by ADDA.
    - Requires the ADDA package to be installed separately, as detailed in the *DDA section*
    - Functions in two different ways, as controlled by the `use_indicators` flag. If the `use_indicators` flag is `True`, the scatterer is voxelated within HoloPy before passing to ADDA. If the flag is `False`, ADDA's built-in scatterer geometries are used for things like spheres, cylinders, ellipsoids, etc.
- **`Mie`**
    - Can handle `Sphere` objects, `LayeredSphere` objects, or `Spheres` through superposition.
    - Computes scattered fields using Mie theory.
- **`Multisphere`**

  - Can handle `Spheres` objects.
  - Cannot handle `Spheres` objects composed of layered spheres.
  - Computes scattered fields through a T-matrix-based solution of scattering, accounting for multiple scattering between spheres to find a (numerically) exact solution.

- **`Tmatrix`**
  - Can handle `Sphere`, `Cylinder`, or `Spheroid` objects.
  - Computes scattered fields by calculating the T-matrix for axisymmetric scatterers, to find a (numerically) exact solution.
  - Occasionally has problems due to Fortran compilations.

3.2.3 Lens-Based Scattering Theories

- **`Lens`**
  - Create by including one of the *Lens-Free* theories.
  - Can handle whatever the additional included theory can handle.
  - Considerably slower than the normal scattering theory.
  - Performance can be improved if the numexpr package is installed.
- **`MieLens`**
  - Can handle `Sphere` objects, or `Spheres` through superposition.
  - Computes scattered fields using Mie theory, but incorporates diffractive effects of a perfect objective lens.
  - Used for performance; `MieLens(lens_angle)` is much faster than calling `Lens(lens_angle, Mie())` and slightly faster than `Mie()`.
- **`AberratedMieLens`**
  - Can handle `Sphere` objects, or `Spheres` through superposition.
  - Computes scattered fields using Mie theory, but incorporates both diffractive effects of an objective lens and arbitrary-order spherical aberration.
  - `AberratedMieLens` and `MieLens` have the same computational cost, although `AberratedMieLens` requires more parameters for fitting.

3.2.4 Which Scattering Theory should I use?

HoloPy chooses a default scattering theory based off the scatterer type, currently determined by the function `determine_default_theory_for()`. If you're not satisfied with HoloPy's default scattering theory selection, you should choose the scattering theory based off of (1) the scatterer that you are modeling, and (2) whether you want to describe the effect of the lens on the recorded hologram in detail.

An Individual Sphere

For single spheres, the default is to calculate scattering using Mie theory, implemented in the class `Mie`. Mie theory is the exact solution to Maxwell's equations for the scattered field from a spherical particle, originally derived by Gustav Mie and (independently) by Ludvig Lorenz in the early 1900s.

Multiple Spheres

A scatterer composed of multiple spheres can exhibit multiple scattering and coupling of the near-fields of neighbouring particles. Mie theory doesn't include these effects, so `Spheres` objects are by default calculated using the `Multisphere` theory, which accounts for multiple scattering by using the SCSMFO package from Daniel Mackowski. This calculation uses T-matrix methods to give the exact solution to Maxwell's equation for the scattering from an arbitrary arrangement of non-overlapping spheres.

Sometimes you might want to calculate scattering from multiple spheres using Mie theory if you are worried about computation time or if your spheres are widely separated (such that optical coupling between the spheres is negligible) You can specify Mie theory manually when calling the `calc_holo()` function, as the following code snippet shows:

```
import holopy as hp
from holopy.core.io import get_example_data_path
from holopy.scattering import (
    Sphere,
    Spheres,
    Mie,
    calc_holo)

s1 = Sphere(center=(5, 5, 5), n = 1.59, r = .5)
s2 = Sphere(center=(4, 4, 5), n = 1.59, r = .5)
collection = Spheres([s1, s2])

imagepath = get_example_data_path('image0002.h5')
exp_img = hp.load(imagepath)

holo = calc_holo(exp_img, collection, theory=Mie)
```

Note that the multisphere theory does not work with collections of multi-layered spheres; in this case HoloPy defaults to using Mie theory with superposition.

Non-spherical particles

HoloPy also includes scattering theories that can calculate scattering from non-spherical particles. For cylindrical or spheroidal particles, by default HoloPy calculates scattering from cylindrical or spheroidal particles by using the `Tmatrix` theory, which uses the T-matrix code from Michael Mishchenko.

```
from holopy.scattering.theory import Tmatrix
from holopy.scattering.scatterer import Spheroid

spheroid = Spheroid(n=1.59, r=(1., 2.), center=(4, 4, 5))
theory = Tmatrix()
holo = calc_holo(exp_img, spheroid, theory=theory)
```

Holopy can also access a discrete dipole approximation (DDA) theory to model arbitrary non-spherical objects. See the *Scattering from Arbitrary Structures with DDA* tutorial for more details.

Including the effect of the lens

Most of the scattering theories in HoloPy treat the fields on the detector as a (magnified) image of the fields at the focal plane. While these theories usually provide a good description of holograms of particles far above the focus, when the particle is near near the focus subtle optical effects can cause deviations between the recorded hologram and theories which do not specifically describe the effects of the lens. To deal with this, HoloPy currently offers two scattering theories which describe the effects of a perfect lens on the recorded hologram. Both of these scattering theories need information about the lens to make predictions, specifically the acceptance angle of the lens. The acceptance angle β is related to the numerical aperture or NA of the lens by $\beta = \arcsin(NA/n_f)$, where n_f is the refractive of the immersion fluid. For more details on the effect of the lens on the recorded hologram, see our papers here and here.

The `Lens` theory allows HoloPy to include the effects of a perfect objective lens with any scattering theory. The Lens theory works by wrapping a normal scattering theory. For instance, to calculate the image of a sphere in an objective lens with an acceptance angle of 1.0, do

```
from holopy.scattering.theory import Lens, Mie
lens_angle = 1.0
theory = Lens(lens_angle, Mie())
```

This theory can then be passed to `calc_holo()` just like any other scattering theory. However, calculations with the `Lens` theory are very slow, orders of magnitude slower than calculations without the lens.

To get around the slow speed of the `Lens` theory, HoloPy offers an additional theory, `MieLens`, specifically for spherical particles imaged with a perfect lens. For spherical particles, some analytical simplifications are possible which greatly speed up the description of the objective lens – in fact, the `MieLens` theory's implementation is slightly faster than `Mie` theory's. The following code creates a `MieLens` theory, which can be passed to `calc_holo()` just like any other scattering theory:

```
from holopy.scattering.theory import MieLens
lens_angle = 1.0
theory = MieLens(lens_angle)
```

In addition, *holopy* supports the calculation of holograms of spherical particles when the imaging objective lens has spherical aberrations of arbitrary order. Currently only spherical aberrations are supported, and only for the image of spherical scatterers. The following code creates a `AberratedMieLens` theory with aberrations up to 8th order in the phase. This theory can be passed to `calc_holo()` just like any other scattering theory:

```
from holopy.scattering.theory import AberratedMieLens
aberration_coefficients = [1.0, 2.0, 3.0]
lens_angle = 1.0
theory = AberratedMieLens(
    spherical_aberration=aberration_coefficients,
    lens_angle=lens_angle)
```

My Scattering theory isn't here?!?!

Add your own scattering theory to HoloPy! See *Adding a new scattering theory* for details. If you think your new scattering theory may be useful for other users, please consider submitting a pull request.

3.3 HoloPy Tools

Holopy contains a number of tools to help you with common tasks when analyzing holograms. This page provides a summary of the tools available, while full descriptions can be found in the relevant code reference.

3.3.1 General Image Processing Tools

The tools described here are frequently used when analyzing holgrams. They are available from the `holopy.core.process` namespace.

The *normalize()* function divides an image by its average, returning an image with a mean pixel value of 1. Note that this is the same normalization convention used by HoloPy when calculating holograms with *calc_holo()*.

Cropping an image introduces difficulties in keeping track of the relative coordinates of features within an image and maintaining metadata. By using the *subimage()* function, the image origin is maintained in the cropped image, so coordinate locations of features (such as a scatterer) remain unchanged.

Since holograms of particles usually take the form of concentric rings, the location of a scatterer can usually be found by locating the apparent center(s) of the image. Use *center_find()* to locate one or more centers in an image.

You can remove isolated dead pixels with zero intensity (e.g. for a background division) by using *zero_filter()*. This function replaces the dead pixel with the average of its neighbours, and fails if adjacent pixels have zero intensity.

The *add_noise()* function allows you to add Gaussian-correlated random noise to a calculated image so that it more closely resembles experimental data.

To find gradient values at all points in an image, use *image_gradient()*. To simply remove a planar intensity gradient from an image, use *detrend()*. Note that this gives a mean pixel value of zero.

Frequency space analysis provides a powerful tool for working with images. Use *fft()* and *ifft()* to perform fourier transforms and inverse fourier transforms, respectively. These make use of `numpy.fft` functions, but are wrapped to correctly interpret HoloPy objects. HoloPy also includes a Hough transform (*hough()*) to help identify lines and other features in your images.

3.3.2 Math Tools

HoloPy contains implementations of a few mathematical functions related to scattering calculations. These functions are available from the `holopy.core.math` namespace.

To find the distance between two points, use *cartesian_distance()*.

To rotate a set of points by arbitrary angles about the three coordinate axes, use *rotate_points()*. You can also calculate a rotation matrix with *rotation_matrix()* to save and use later.

To convert spherical coordinates into Cartesian coordinates, use *to_cartesian()*. To convert Cartesian coordinates into spherical coordinates, use `to_spherical()`.

When comparing data to a model, the chi-squared and r-squared values provide measures of goodness-of-fit. You can access these through *chisq()* and *rsq()*.

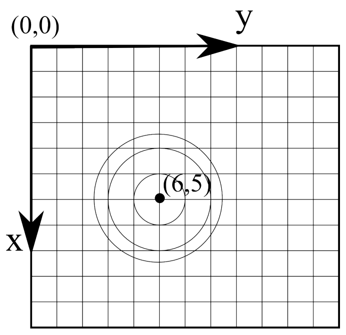
3.4 HoloPy Concepts

3.4.1 Units

HoloPy does **not** enforce any particular set of units. As long as you are consistent, you can use any set of units, for example pixels, meters, or microns. So if you specify the wavelength of your red imaging laser as 658 then all other units (x, y, z position coordinates, particle radii, etc.) must also be specified in nanometers.

3.4.2 Coordinate System

For image data (data points arrayed in a regular grid in a single plane), HoloPy defaults to placing the origin, (0,0), at the top left corner as shown below. The x-axis runs vertically down, the y-axis runs horizontally to the right, and the z-axis points out of the screen, toward you. This corresponds to the way that images are treated by most computer software.



In sample space, we choose the z axis so that distances to objects from the camera/focal plane are positive (have positive z coordinates). The price we pay for this choice is that the propagation direction of the illumination light is then negative. In the image above, light travels from a source located in front of the screen, through a scatterer, and onto a detector behind the screen.

More complex detector geometries will define their own origin, or ask you to define one.

3.4.3 Rotations of Scatterers

Certain scattering calculations in HoloPy require specifying the orientation of a scatterer (such as a Janus sphere) relative to the HoloPy coordinate system. We do this in the most general way possible by specifying three Euler angles and a reference orientation. Rotating a scatterer initially in the reference orientation through the three Euler angles α, β, and γ (in the active transformation picture) yields the desired orientation. The reference orientation is specified by the definition of the scatterer.

The Euler rotations are performed in the following way:

1. Rotate the scatterer an angle α about the HoloPy z axis.
2. Rotate the scatterer an angle β about the HoloPy y axis.
3. Rotate the scatterer an angle γ about the HoloPy z axis.

The sense of rotation is as follows: each angle is a rotation in the *clockwise* direction about the specified axis, viewed along the positive direction of the axis from the origin. This is the usual sense of how rotations are typically defined in math:

$$\mathbf{v}''' = \begin{pmatrix} \cos\gamma & \sin\gamma & 0 \\ -\sin\gamma & \cos\gamma & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos\beta & 0 & -\sin\beta \\ 0 & 1 & 0 \\ \sin\beta & 0 & \cos\beta \end{pmatrix} \begin{pmatrix} \cos\alpha & \sin\alpha & 0 \\ -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix} \mathbf{v}.$$

CHAPTER 4

holopy package

HoloPy is a set of tools for working with digital holograms and light scattering. It contains tools for working loading data and associating it with expermiental metadata, reconstructing holograms, calculating light scattering, fitting scattering models to data, and visualizing images and calculations.

4.1 Subpackages

4.1.1 holopy.core package

Loading, saving, and basic processing of data.

Subpackages

holopy.core.io package

Submodules

Common entry point for holopy io. Dispatches to the correct load/save functions.

class Accumulator

Bases: `object`

Calculates average and coefficient of variance for numerical data in one pass using Welford's algorithim.

cv()

The coefficient of variation

mean()

push(*x*)

default_extension(*inf*, *defext='.h5'*)

get_example_data(*name*)

get_example_data_path(*name*)

load(*inf*, *lazy=False*)

Load data or results

Parameters **inf** (*string*) – String specifying an hdf5 file containing holopy data

Returns **obj** – The array object contained in the file

Return type xarray.DataArray

load_average(*filepath*, *refimg=None*, *spacing=None*, *medium_index=None*, *illum_wavelen=None*, *illum_polarization=None*, *noise_sd=None*, *channel=None*, *image_glob='*.tif'*)

Average a set of images (usually as a background)

Parameters

- **filepath** (*string or list(string)*) – Directory or list of filenames or filepaths. If filename is a directory, it will average all images matching image_glob.
- **refimg** (*xarray.DataArray*) – reference image to provide spacing and metadata for the new image.
- **spacing** (*float*) – Spacing between pixels in the images. Used preferentially over refimg value if both are provided.
- **medium_index** (*float*) – Refractive index of the medium in the images. Used preferentially over refimg value if both are provided.
- **illum_wavelen** (*float*) – Wavelength of illumination in the images. Used preferentially over refimg value if both are provided.
- **illum_polarization** (*list-like*) – Polarization of illumination in the images. Used preferentially over refimg value if both are provided.
- **image_glob** (*string*) – Glob used to select images (if images is a directory)

Returns **averaged_image** – Image which is an average of images noise_sd attribute contains average pixel stdev normalized by total image intensity

Return type xarray.DataArray

load_image(*inf*, *spacing=None*, *medium_index=None*, *illum_wavelen=None*, *illum_polarization=None*, *noise_sd=None*, *channel=None*, *name=None*)

Load data or results

Parameters

- **inf** (*string*) – File to load.
- **spacing** (*float or (float, float) (optional)*) – pixel size of images in each dimension - assumes square pixels if single value. set equal to 1 if not passed in and issues warning.
- **medium_index** (*float (optional)*) – refractive index of the medium
- **illum_wavelen** (*float (optional)*) – wavelength (in vacuum) of illuminating light
- **illum_polarization** (*(float, float) (optional)*) – (x, y) polarization vector of the illuminating light
- **noise_sd** (*float (optional)*) – noise level in the image, normalized to image intensity

- **channel** (*int or tuple of ints (optional)*) – number(s) of channel to load for a color image (in general 0=red, 1=green, 2=blue) name : str (optional) name to assign the xr.DataArray object resulting from load_image

Returns **obj**

Return type xarray.DataArray representation of the image with associated metadata

pack_attrs(*a*, *do_spacing=False*)

save(*outf*, *obj*)

Save a holopy object

Will save objects as yaml text containing all information about the object unless outf is a filename with an image extension, in which case it will save an image, truncating metadata.

Parameters

- **outf** (*basestring or file*) – Location to save the object
- **obj** (*holopy.core.holopy_object.HoloPyObject*) – The object to save

save_image(*filename*, *im*, *scaling='auto'*, *depth=8*)

Save an ndarray or image as a tiff.

Parameters

- **filename** (*basestring*) – filename in which to save image. If im is an image the function should default to the image's name field if no filename is specified
- **im** (ndarray or `holopy.image.Image`) – image to save.
- **scaling** (*'auto', None, or (Int, Int)*) – How the image should be scaled for saving. Ignored for float output. It defaults to auto, use the full range of the output format. Other options are None, meaning no scaling, or a pair of integers specifying the values which should be set to the maximum and minimum values of the image format.
- **depth** (*8, 16 or 'float'*) – What type of image to save. Options other than 8bit may not be supported for many image types. You probably don't want to save 8bit images without some kind of scaling.

save_images(*filenames*, *ims*, *scaling='auto'*, *depth=8*)

Saves a volume as separate images (think of reconstruction volumes).

Parameters

- **filenames** (*list*) – List of filenames. There have to be the same number of filenames as of images to save. Each image will be saved in the corresponding file with the same index.
- **ims** (ndarray or `holopy.image.Image`) – Images to save, with separate z-coordinates from which they each will be selected.
- **scaling** (*'auto', None, or (Int, Int)*) – How the images should be scaled for saving. Ignored for float output. It defaults to auto, use the full range of the output format. Other options are None, meaning no scaling, or a pair of integers specifying the values which should be set to the maximum and minimum values of the image format.
- **depth** (*8, 16 or 'float'*) – What type of image to save. Options other than 8bit may not be supported for many image types. You probably don't want to save 8bit images without some kind of scaling.

unpack_attrs(*a*)

Reading and writing of yaml files.

yaml files are structured text files designed to be easy for humans to read and write but also easy for computers to read. HoloPy uses them to store information about experimental conditions and to describe analysis procedures.

class_loader(*loader*, *node*)

class_representer(*dumper*, *data*)

complex_constructor(*loader*, *node*)

complex_representer(*dumper*, *data*)

ignore_aliases(*data*)

instancemethod_constructor(*loader*, *node*)

instancemethod_representer(*dumper*, *data*)

load(*inf*)

ndarray_representer(*dumper*, *data*)

numpy_float_representer(*dumper*, *data*)

numpy_int_representer(*dumper*, *data*)

numpy_ufunc_constructor(*loader*, *node*)

numpy_ufunc_representer(*dumper*, *data*)

save(*outf*, *obj*)

tuple_representer(*dumper*, *data*)

Prepare HoloPy objects for display on screen or write to file.

class Show2D(*im*)
Bases: `object`

click(*event*)

draw()

format_coord(*x*, *y*)

save(*filename*)
Saves the currently displayed Plot into a file.

Parameters **filename** (*str*) – Name for the file to save to.

save_all(*filenames*)
Saves the complete stack of images into separate files.

Parameters **filenames** (*list*) – Names of the files to save as a list. Has to have the same length as the number of images that are contained in this object.

exception VisualizationNotImplemented(*o*)
Bases: `Exception`

check_display()
Diagnostic test to check matplotlib backend.

You should see a white square inside a black square, with a colorbar. Pressing the left or right arrow keys should cycle through z. You should see:

Z = 0 : A white axes-aligned square Z = 1 : A white circle Z = 2 : A white diamond (square at 45 degrees)

display_image(*im*, *scaling='auto'*, *vert_axis='x'*, *horiz_axis='y'*, *depth_axis='z'*, *colour_axis='illumination'*)

save_plot(*filenames*, *data*, *scaling='auto'*, *vert_axis='x'*, *horiz_axis='y'*, *depth_axis='z'*, *colour_axis='illumination'*)

Saves a hologram or reconstruction to (a) file(s).

Parameters

- **filenames** (*list / str*) – Name(s) of the file(s). If there is only one image contained (e.g. hologram), the name will be used as a file name. If o contains more plottable images (e.g. reconstruction volume), it should be a list of filenames with the same length as objects.
- **data** (*xarray.DataArray or ndarray*) – Object to save.
- **scaling** (*(float, float), optional*) – (min, max) value to display in image, default is full range of o.
- **vert_axis** (*str, optional*) – axis to display vertically, default x.
- **horiz_axis** (*str, optional*) – axis to display horizontally, default y.
- **depth_axis** (*str, optional*) – axis to scroll with arrow keys, default 'z'.
- **colour_axis** (*str, optional*) – axis to display as RGB channels in colour image, default illumination.

Notes

Loads plotting library the first time it is required (so that we don't have to import all of matplotlib or mayavi just to load holopy)

show(*o*, *scaling='auto'*, *vert_axis='x'*, *horiz_axis='y'*, *depth_axis='z'*, *colour_axis='illumination'*)

Visualize a hologram or reconstruction

Parameters

- **o** (*xarray.DataArray or ndarray*) – Object to visualize
- **scaling** (*(float, float), optional*) – (min, max) value to display in image, default is full range of o.
- **vert_axis** (*str, optional*) – axis to display vertically, default x.
- **horiz_axis** (*str, optional*) – axis to display horizontally, default y.
- **depth_axis** (*str, optional*) – axis to scroll with arrow keys, default 'z'.
- **colour_axis** (*str, optional*) – axis to display as RGB channels in colour image, default illumination.

Notes

Loads plotting library the first time it is required (so that we don't have to import all of matplotlib or mayavi just to load holopy)

show_scatterer_slices(*scatterer*, *spacing*)

Show slices of a scatterer voxelation

scatterer [.Scatterer] scatterer to visualize

spacing [float or (float, float, float)] voxel spacing for the visualization

show_sphere_cluster(*s*, *color*)

This function to show a 3D rendering of a Spheres obj hasn't worked since HoloPy 3.0, due to Mayavi compatibility issues. We keep the code because we hope to re-implement this functionality eventually.

holopy.core.process package

Routines for image processing. Useful for pre-processing raw holograms prior to extracting final data or post-processing reconstructions.

Submodules

The centerfinder module is a group of functions for locating the centers of holographic ring patterns. The module can find the center of a single-sphere holographic pattern, a dimer holographic pattern, or the centers of multiple (well-separated: clearly separate ring patterns with separate centers) single spheres or dimers. The intended use is for determining an initial parameter guess for hologram fitting.

We thank the Grier Group at NYU for suggesting the use of the Hough transform. For their independent implementation of a Hough-based holographic feature detection algorithm, see: http://physics.nyu.edu/grierlab/software/circletransform.pro For a case study and further reading, see: F. C. Cheong, B. Sun, R. Dreyfus, J. Amato-Grill, K. Xiao, L. Dixon & D. G. Grier, Flow visualization and flow cytometry with holographic video microscopy, Optics Express 17, 13071-13079 (2009).

center_find(*image*, *centers=1*, *threshold=0.5*, *blursize=3.0*)

Finds the coordinates of the center of a holographic pattern. The coordinates returned are in pixels (row number, column number). Intended for finding the center of single particle or dimer holograms which basically show concentric circles. The optional threshold parameter (between 0 and 1) gives a bound on what magnitude of gradients to include in the calculation. For example, threshold=.75 means ignore any gradients that are less than 75% of the maximum gradient in the image. The optional blursize parameter sets the size of a Gaussian filter that is applied to the image. This step improves accuracy when small features in the image have large gradients (e.g. dust particles on the camera). Without blurring, these features may be incorrectly identified as the hologram center. For best results, blursize should be set to the radius of features to be ignored, but smaller than the distance between hologram fringes. To skip blurring, set blursize to 0.

Parameters

- **image** (`ndarray`) – image to find the center(s) in
- **centers** (`int`) – number of centers to find
- **threshold** (`float (optional)`) – fraction of the maximum gradient below which all other gradients will be ignored (range 0-.99)
- **blursize** (`float (optional)`) – radius (in pixels) of the Gaussian filter that is applied prior to Hough transform

Returns **res** – row(s) and column(s) of center(s)

Return type ndarray

Notes

When threshold is close to 1, the code will run quickly but may lack accuracy. When threshold is set to 0, the gradient at all pixels will contribute to finding the centers and the code will take a little bit longer.

hough(*col_deriv*, *row_deriv*, *centers=1*, *threshold=0.25*)

Following the approach of a Hough transform, finds the pixel which the most gradients point towards or away from. Uses only gradients with magnitudes greater than threshold*maximum gradient. Once the pixel is found, uses a brightness-weighted average around that pixel to refine the center location to return. After the first center is found, the sourrounding area is blocked out and another brightest pixel is searched for if more centers are required.

Parameters

- **`col_deriv`** (*`numpy.ndarray`*) – y-component of image intensity gradient
- **`row_deriv`** (*`numpy.ndarray`*) – x-component of image intensity gradient
- **`centers`** (*`int`*) – number of centers to find
- **`threshold`** (*`float (optional)`*) – fraction of the maximum gradient below which all other gradients will be ignored (range 0-.99)

Returns **res** – row and column of center or centers

Return type ndarray

`image_gradient`(*image*)

Uses the Sobel operator as a numerical approximation of a derivative to find the x and y components of the image's intensity gradient at each pixel.

Parameters **`image`** (*`ndarray`*) – image to find the gradient of

Returns

- **gradx** (*ndarray*) – x-components of intensity gradient
- **grady** (*ndarray*) – y-components of intensity gradient

Handles Fourier transforms of HoloPy images by using numpy's fft package. Tries to correctly interpret dimensions from xarray.

`fft`(*data*, *shift=True*)

More convenient Fast Fourier Transform

An easier to use fft function, it will pick the correct fft to do based on the shape of the array, and do the fftshift for you. This is intended for working with images, and thus for dimensions greater than 2 does slicewise transforms of each "image" in a multidimensional stack

Parameters

- **`data`** (*`ndarray or xarray`*) – The array to transform
- **`shift`** (*`bool`*) – Whether to preform an fftshift on the array to give low frequences near the center as you probably expect. Default is to do the fftshift.

Returns **fta** – The fourier transform of *a*

Return type ndarray

`ft_coord`(*c*)

`ft_coords`(*cs*)

`get_spacing`(*c*)

`ifft`(*data*, *shift=True*)

More convenient Inverse Fast Fourier Transform

An easier to use ifft function, it will pick the correct ifft to do based on the shape of the array, and do the fftshift for you. This is intended for working with images, and thus for dimensions greater than 2 does slicewise transforms of each "image" in a multidimensional stack

Parameters

- **data** (*ndarray or xarray*) – The array to transform
- **shift** (*bool*) – Whether to preform an fftshift on the array to give low frequences near the center as you probably expect. Default is to do the fftshift.

Returns The inverse fourier transform of *data*

Return type ndarray

ift_coord(*c*)

ift_coords(*cs*)

transform_metadata(*a*, *inverse*)

Image enhancement through background subtraction, contrast adjustment, or detrending

add_noise(*image*, *noise_mean=0.1*, *smoothing=0.01*, *poisson_lambda=1000*)

Add simulated noise to images. Intended for use with exact calculated images to make them look more like noisy 'real' measurements.

Real image noise usually has correlation, so we smooth the raw random variable. The noise_mean can be controlled independently of the poisson_lambda that controls the shape of the distribution. In general, you can stick with our default of a large poisson_lambda (ie for imaging conditions not near the shot noise limit).

Defaults are set to give noise vaguely similar to what we tend to see in our holographic imaging.

Parameters

- **image** (*xarray.DataArray*) – The image to add noise to.
- **smoothing** (*float*) – Fraction of the image size to smooth by. Should in general be << 1
- **poisson_lambda** (*float*) – Used to compute the shape of the noise distribution. You can generally leave this at its default value unless you are simulating shot noise limited imaging.

Returns **noisy_image** – A copy of the input image with noise added.

Return type xarray.DataArray

bg_correct(*raw*, *bg*, *df=None*)

Correct for noisy images by dividing by a background. The calculation used is (raw-df)/(bg-df).

Parameters

- **raw** (*xarray.DataArray*) – Image to be background divided.
- **bg** (*xarray.DataArray*) – background image recorded with the same optical setup.
- **df** (*xarray.DataArray*) – dark field image recorded without illumination.

Returns **corrected_image** – A copy of the background divided input image with None values of noise_sd updated to match bg.

Return type xarray.DataArray

detrend(*image*)

Remove linear trends from an image.

Performs a 2 axis linear detrend using scipy.signal.detrend

Parameters **image** (*xarray.DataArray*) – Image to process

Returns **image** – Image with linear trends removed

Return type xarray.DataArray

normalize(*image*)

Normalize an image by dividing by the pixel average. This gives the image a mean value of 1.

Parameters **image** (*xarray.DataArray*) – The array to normalize

Returns **normalized_image** – The normalized image

Return type xarray.DataArray

simulate_noise(*shape*, *mean=0.1*, *smoothing=0.01*, *poisson_lambda=1000*)

Create an array of correlated noise. The noise_mean can be controlled independently of the poisson_lambda that controls the shape of the distribution. In general, you can stick with our default of a large poisson_lambda (ie for imaging conditions not near the shot noise limit).

Defaults are set to give noise vaguely similar to what we tend to see in our holographic imaging.

Parameters

- **shape** (*int or array_like of ints*) – shape of noise array
- **smoothing** (*float*) – Fraction of the image size to smooth by. Should in general be << 1
- **poisson_lambda** (*float*) – Used to compute the shape of the noise distribution. You can generally leave this at its default value unless you are simulating shot noise limited imaging.

Returns **noisy_image** – A copy of the input image with noise added.

Return type ndarray

subimage(*arr*, *center*, *shape*)

Pick out a region of an image or other array

Parameters

- **arr** (*xarray.DataArray*) – The array to subimage
- **center** (*tuple of ints or floats*) – The desired center of the region, should have the same number of elements as the arr has dimensions. Floats will be rounded
- **shape** (*int or (int, int)*) – Desired shape of the region in x & y dimensions. If a single int is given it is applied along both axes. Shape values must be even.

Returns **sub** – Subset of shape shape centered at center. DataArray coords will be set such that the upper left corner of the output has coordinates relative to the input.

Return type xarray.DataArray

zero_filter(*image*)

Search for and interpolate pixels equal to 0. This is to avoid NaN's when a hologram is divided by a BG with 0's. Interpolation fails if any of the four corner pixels are 0.

Parameters **image** (*xarray.DataArray*) – Image to process

Returns **image** – Image where pixels = 0 are instead given values equal to average of neighbors. dtype is the same as the input image

Return type xarray.DataArray

Submodules

Error classes used in holopy

exception BadImage
    Bases: `Exception`

exception CoordSysError
    Bases: `Exception`

exception DependencyMissing(*dependency*, *message=''*)
    Bases: `Exception`

exception DeprecationError
    Bases: `Exception`

exception LoadError(*filename*, *message*)
    Bases: `Exception`

exception NoMetadata
    Bases: `Exception`

exception PerformanceWarning
    Bases: `UserWarning`

raise_fitting_api_error(*correct*, *obselete*)

Root class for all of holopy. This class provides serialization to and from yaml text file for all holopy objects.

yaml files are structured text files designed to be easy for humans to read and write but also easy for computers to read. HoloPy uses them to store information about experimental conditions and to describe analysis procedures.

class HoloPyObject
    Bases: *holopy.core.holopy_object.Serializable*

    Ancestor class for all HoloPy classes.

    HoloPy object's purpose is to provide the machinery for saving to and loading from HoloPy yaml files

    classmethod from_yaml(*loader*, *node*)
        Convert a representation node to a Python object.

    classmethod to_yaml(*dumper*, *data*)
        Convert a Python object to a representation node.

class Serializable
    Bases: `yaml.YAMLObject`

    Base class for any object that wants a nice clean yaml output

    classmethod to_yaml(*dumper*, *data*)
        Convert a Python object to a representation node.

class SerializableMetaclass(*name*, *bases*, *kwds*)
    Bases: `yaml.YAMLObjectMetaclass`

class Mapper
    Bases: *holopy.core.holopy_object.HoloPyObject*

    Creates "maps" from objects containing priors that retain their hierarchical structure (including ties) but are easily serializable. The main entry point is through *convert_to_map*, which returns a map of the object and also updates the Mapper *parameter* and *parameter_names* attributes so they can be extracted for later use.

    add_parameter(*parameter*, *name*)

    check_for_ties(*parameter*)

    convert_to_map(*parameter*, *name=''*)

    get_parameter_index(*parameter*, *name*)

iterate_mapping(*prefix*, *pairs*)

map_dictionary(*parameter*, *name*)

map_transformed_prior(*parameter*, *name*)

map_xarray(*parameter*, *name*)

edit_map_indices(*map_entry*, *indices*)

Adjusts a map to account for ties between parameters

Parameters

- **map_entry** – map or subset of map created by model methods
- **indices** (`listlike`) – indices of parameters to be tied

make_xarray(*dim_name*, *keys*, *values*)

Packs values into xarray with new dim and coords (keys)

read_map(*map_entry*, *parameter_values*)

Reads a map to create an object

Parameters

- **map_entry** – map or subset of map created by model methods
- **parameter_values** (`listlike`) – values to replace map placeholders in final object

transformed_prior(*transformation*, *base_priors*)

cartesian_distance(*p1, p2=[0, 0, 0]*)

Return the Cartesian distance between points p1 and p2.

Parameters **p2** (`p1,`) – Coordinates of point 1 and point 2 in N-dimensions

Returns **dist** – Cartesian distance between points p1 and p2

Return type float64

chisq(*fit*, *data*)

Calculate per-point value of chi-squared comparing a best-fit model and data.

Parameters

- **fit** (`array_like`) – Values of best-fit model to be compared to data
- **data** (`array_like`) – Data values to be compared to model

Returns **chisq** – Chi-squared per point

Return type float

Notes

chi-squared is defined as

$$\chi^2 = \frac{1}{N}\sum_{\text{points}}(\text{fit} - \text{data})^2$$

where N is the number of data points.

find_transformation_function(*initial_coordinates*, *desired_coordinates*)

keep_in_same_coordinates(*coords*)

rotate_points (*points*, *theta*, *phi*, *psi*)

Rotate an array of points about Euler angles in a z, y, z convention.

Parameters

- **points** (*array-like (n, 3)*) – Set of points to be rotated
- **phi, psi** (*theta,*) – Euler rotation angles in z, y, z convention. These are *not* the same as angles in spherical coordinates.

Returns **rotated_points** – Points rotated by Euler angles

Return type array(n,3)

rotation_matrix (*alpha*, *beta*, *gamma*, *radians=True*)

Return a 3D rotation matrix

Parameters

- **beta, gamma** (*alpha,*) – Euler rotation angles in z, y, z convention
- **radians** (*boolean*) – Default True; treats input angles in radians

Returns **rot** – Rotation matrix. To rotate a column vector x, use np.dot(rot, x.)

Return type array(3,3)

Notes

The Euler angles rotate a vector (in the active picture) by alpha clockwise about the fixed lab z axis, beta clockwise about the lab y axis, and by gamma about the lab z axis. Clockwise is defined as viewed from the origin, looking in the positive direction along an axis.

This breaks compatability with previous conventions, which were adopted for compatability with the passive picture used by SCSMFO.

rsq (*fit*, *data*)

Calculate correlation coeffiction R-squared comparing a best-fit model and data.

Parameters

- **fit** (*array_like*) – Values of best-fit model to be compared to data
- **data** (*array_like*) – Data values to be compared to model

Returns **rsq** – Correlation coefficient R-squared.

Return type float

Notes

R-squared is defined as

$$R^2 = 1 - \frac{\sum_{\text{points}}(\text{data} - \text{fit})^2}{\sum_{\text{points}}(\text{data} - \bar{\text{data}})^2}$$

where $\bar{\text{data}}$ is the mean value of the data. If the model perfectly describes the data, $R^2 = 1$.

to_cartesian (*r*, *theta*, *phi*)

transform_cartesian_to_cylindrical (*x_y_z*)

transform_cartesian_to_spherical (*x_y_z*)

transform_cylindrical_to_cartesian(*rho_phi_z*)

transform_cylindrical_to_spherical(*rho_phi_z*)

transform_spherical_to_cartesian(*r_theta_phi*)

transform_spherical_to_cylindrical(*r_theta_phi*)

Classes for defining metadata about experimental or calculated results.

clean_concat(*arrays*, *dim*)

Concatenate a list of xr.DataArray objects along a specified dimension, keeping the metadata of the first array.

Parameters

- **arrays** (list of `xr.xarray`) –
- **dim** (*valid dimension (string)*) –

Returns

Return type xarray

copy_metadata(*old*, *data*, *do_coords=True*)

Create a new *xarray* with data from one input and metadata from another.

Parameters

- **old** (*xr.DataArray*) – The xarray to copy the metadata from.
- **data** (*xr.DataArray*) – The xarray to copy the data from.
- **do_coords** (*bool, optional*) – Whether or not to copy the coordinates. Default is True
- **Returns** –
- **xr.DataArray** –

data_grid(*arr*, *spacing=None*, *medium_index=None*, *illum_wavelen=None*, *illum_polarization=None*, *noise_sd=None*, *name=None*, *extra_dims=None*, *z=0*)

Create a set of detector points along with other experimental metadata.

Returns

Return type DataArray object

Notes

Use the higher-level detector_grid() and detector_points() functions. This should be viewed as a factory method.

detector_grid(*shape*, *spacing*, *name=None*, *extra_dims=None*)

Create a rectangular grid of pixels to represent a detector on which scattering calculations are to be performed.

Parameters

- **shape** (*int or list-like (2)*) – If int, detector is a square grid of shape x shape points. If array_like, detector has *shape*[0] rows and *shape*[1] columns.
- **spacing** (*int or list-like (2)*) – If int, distance between square detector pixels. If array_like, *spacing*[0] between adjacent rows and *spacing*[1] between adjacent columns.
- **name** (*string, optional*) –
- **extra_dims** (*dict, optional*) – extra dimension(s) to add to the empty detector grid as {dimname: [coords]}.

Returns **grid** – DataArray of zeros with coordinates calculated according to *shape* and *spacing*

Return type DataArray object

Notes

Typically used to define a set of points to represent the pixels of a digital camera in scattering calculations.

`detector_points` (*coords={}*, *x=None*, *y=None*, *z=None*, *r=None*, *theta=None*, *phi=None*, *name=None*)
Returns a one-dimensional set of detector coordinates at which scattering calculations are to be done.

Parameters

- **`coords`** (*`dict, optional`*) – Dictionary of detector coordinates. Default: empty dictionary. Typical usage should not pass this argument, giving other parameters (Cartesian *x*, *y*, and *z* or polar *r*, *theta*, and *phi* coordinates) instead.
- **`y`** (*`x,`*) – Cartesian x and y coordinates of detectors.
- **`z`** (*`int or array_like, optional`*) – Cartesian z coordinates of detectors. If not specified, assume *z* = 0.
- **`r`** (*`int or array_like, optional`*) – Spherical polar radial coordinates of detectors. If not specified, assume *r* = infinity (far-field).
- **`theta`** (*`int or array_like, optional`*) – Spherical polar coordinates (polar angle from z axis) of detectors.
- **`phi`** (*`int or array_like, optional`*) – Spherical polar azimuthal coodinates of detectors.
- **`name`** (*`string`*) –

Returns **grid** – DataArray of zeros with calculated coordinates.

Return type DataArray object

Notes

Specify either the Cartesian or the polar coordinates of your detector. This may be helpful for modeling static light scattering calculations. Use detector_grid() to specify coordinates of a grid of pixels (e.g., a digital camera.)

`dict_to_array` (*schema*, *inval*)

`flat` (*a*)

`from_flat` (*a*)

`get_extents` (*detector_grid*)
Find the x, y, z extent of a `` `detector_grid` ``, as a dict.

`get_spacing` (*detector_grid*)
Find the (x, y) spacing for a `` `detector_grid` ``.

`get_values` (*a*)

`make_coords` (*shape*, *spacing*, *z=0*)

`make_subset_data` (*data*, *pixels=None*, *return_selection=False*, *seed=None*)
Sub-sample a data for faster inference.

Parameters

- **`data`** (*xr.DataArray*) – The data to subsample

- **pixels** (*int, optional*) – The number of pixels to subsample. Defaults to the entire image.
- **return_selection** (*bool, optional*) – Whether to return the pixel indices which were sampled. Default is False
- **seed** (*int or None, optional*) – If not None, the seed to seed the random number generator with.

Returns

- **subset** (*xr.DataArray*)
- **[selection** (*np.ndarray, dtype int]*)

to_vector(*c*)

update_metadata(*a, medium_index=None, illum_wavelen=None, illum_polarization=None, noise_sd=None*)

Returns a copy of an image with updated metadata in its 'attrs' field.

Parameters

- **a** (*xarray.DataArray*) – image to update.
- **medium_index** (*float*) – Updated refractive index of the medium in the image.
- **illum_wavelen** (*float*) – Updated wavelength of illumination in the image.
- **illum_polarization** (*list-like*) – Updated polarization of illumination in the image.
- **noise_sd** (*float*) – standard deviation of Gaussian noise in the image.

Returns **b** – copy of input image with updated metadata.

Return type xarray.DataArray

class BoundedGaussian(*mu, sd, lower_bound=-inf, upper_bound=inf, name=None*)

Bases: *holopy.core.prior.Gaussian*

lnprob(*p*)

Note that this does not return the actual log-probability, but a value proportional to it.

prob(*p*)

Note that this does not return the actual probability, but a value proportional to it.

sample(*size=None*)

class ComplexPrior(*real, imag, name=None*)

Bases: *holopy.core.prior.TransformedPrior*

A complex free parameter

ComplexPrior has a real and imaginary part which can (potentially) vary separately.

Parameters

- **imag** (*real,*) – The real and imaginary parts of this parameter. Assign floats to fix that portion or parameters to allow it to vary. The parameters must be purely real. You should omit names for the parameters; ComplexPrior will name them
- **name** (*string*) – Short descriptive name of the ComplexPrior. Do not provide this if using a ParameterizedScatterer, a name will be assigned based its position within the scatterer.

imag

lnprob(*p*)

map_keys

prob(*p*)

real

class Gaussian(*mu*, *sd*, *name=None*)
Bases: *holopy.core.prior.Prior*

guess

lnprob(*p*)

prob(*p*)

sample(*size=None*)

variance

class Prior
Bases: *holopy.core.holopy_object.HoloPyObject*

Base class for Bayesian priors in holopy.

Prior subclasses should define at least the following methods:

- guess
- sample
- prob
- lnprob

renamed(*name*)

scale(*physical*)

unscale(*scaled*)

class TransformedPrior(*transformation*, *base_prior*, *name=None*)
Bases: *holopy.core.prior.Prior*

guess

lnprob(*p*)

map_keys

prob(*p*)

sample(*size=None*)

class Uniform(*lower_bound*, *upper_bound*, *guess=None*, *name=None*)
Bases: *holopy.core.prior.Prior*

interval

lnprob(*p*)

prob(*p*)

sample(*size=None*)

generate_guess(*parameters*, *nguess=1*, *scaling=1*, *seed=None*)

make_center_priors(*im*, *z_range_extents=5*, *xy_uncertainty_pixels=1*, *z_range_units=None*)
Make sensible default priors for the center of a sphere in a hologram

Parameters

- **im** (*xarray*) – The image you wish to make priors for
- **z_range_extents** (*float (optional)*) – What range to extend a uniform prior for z over, measured in multiples of the total extent of the image. The default is 5 times the extent of the image, a large range, but since tempering is quite good at refining this, it is safer to just choose a large range to be sure to include the correct value.
- **xy_uncertainty_pixels** (*float (optional)*) – The number of pixels of uncertainty to assume for the centerfinder. The default is 1 pixel, and this is probably correct for most images.
- **z_range_units** (*float*) – Specify the range of the z prior in your data units. If provided, z_range_extents is ignored.

updated(*prior*, *v*, *extra_uncertainty=0*)
Update a prior from a posterior

Parameters

- **v** (UncertainValue) – The new value, usually from an mcmc result
- **extra_uncertainty** (*float*) – provide a floor for uncertainty (sd) of the new parameter

Misc utility functions to make coding more convenient

class LnpostWrapper(*model*, *data*, *new_pixels=None*, *minus=False*)
Bases: *holopy.core.holopy_object.HoloPyObject*

We want to be able to define a specific model.lnposterior calculation that only takes parameter values as an argument for passing into optimizers. However, individual functions can't be pickled to distribute hologram calculations with python multiprocessing. This class solves both issues.

evaluate(*par_vals*)

class NonePool
Bases: object

close()

map(*function*, *arguments*)

class SuppressOutput(*suppress_output=True*)
Bases: object

choose_pool(*parallel*)
This is a remake of schwimmbad.choose_pool with a single argument.

dict_without(*d*, *keys*)
Exclude a list of keys from a dictionary.

Silently ignores any key in keys that is not in the dict (this is intended to be used to make sure a dict does not contain specific keys) :param d: The dictionary to operate on :type d: dict :param keys: The keys to exclude :type keys: list(string) :param returns: A copy of dict without any of the specified keys :type returns: d2

ensure_array(*x*)
if x is None, returns None. Otherwise, gives x in a form so that each of: *len(x)*, *x[0]*, *x+2* will not fail.

ensure_listlike(*x*)

ensure_scalar(*x*)

mkdir_p(*path*)
Equivalent to mkdir -p at the shell, this function makes a directory and its parents as needed, silently doing nothing if it exists.

repeat_sing_dims (*indict*, *keys='all'*)

updated (*d*, *update={}*, *filter_none=True*, ***kwargs*)

Return a dictionary updated with keys from update

Analgous to sorted, this is an equivalent of d.update as a non-modifying free function

Parameters

- **d** (*dict*) – The dict to update
- **update** (*dict*) – The dict to take updates from

4.1.2 holopy.inference package

Submodules

Stochastic fitting of models to data

class CmaStrategy (*npixels=None*, *popsize=None*, *resample_pixels=True*, *parent_fraction=0.25*, *weight_function=None*, *walker_initial_pos=None*, *tols={}*, *seed=None*, *parallel='auto'*)

Bases: `holopy.core.holopy_object.HoloPyObject`

Inference strategy defining a Covariance Matrix Adaptation Evolutionary Strategy using cma package

Parameters

- **npixels** (*int, optional*) – Number of pixels in the image to fit. default fits all.
- **resample_pixels** (*Boolean, optional*) – If true (default), new pixels are chosen for each call of posterior. Otherwise, a single pixel subset is used throughout calculation.
- **parent_fraction** (*float, optional*) – Fraction of each generation to use to construct the next generation. Takes symbol mu in cma literature
- **weight_function** (*function, optional*) – takes arguments (i, popsize), i in range(popsize); returns weight of i
- **tols** (*dict, optional*) – tolerance values to overwrite the cma defaults
- **seed** (*int, optional*) – random seed to use
- **parallel** (*optional*) – number of threads to use or pool object or one of {None, 'all', 'mpi'}. Default tries 'mpi' then 'all'.

fit (*model*, *data*)

run_cma (*obj_func*, *parameters*, *initial_population*, *weight_function*, *tols={}*, *seed=None*, *parallel='auto'*)

instantiate and run a CMAEvolutionStrategy object

Parameters

- **obj_func** (*Function*) – function to be minimized (not maximized like posterior)
- **parameters** (*list of Prior objects*) – parameters to fit
- **initial_population** (*array*) – starting population with shape = (popsize, len(parameters))
- **weight_function** (*function*) – takes arguments (i, popsize), i in range(popsize); returns weight of i
- **tols** (*dict, optional*) – tolerance values to overwrite the cma defaults

- **seed** (*int, optional*) – random seed to use
- **parallel** (*optional*) – number of threads to use or pool object or one of {None, 'all', 'mpi'}. Default tries 'mpi' then 'all'.

Sample posterior probabilities given model and data

class EmceeStrategy(*nwalkers=100*, *nsamples=None*, *npixels=None*, *walker_initial_pos=None*, *parallel='auto'*, *seed=None*)
Bases: `holopy.core.holopy_object.HoloPyObject`

sample(*model*, *data*)

class TemperedStrategy(*next_initial_dist=<function sample_one_sigma_gaussian>*, *nwalkers=100*, *nsamples=1000*, *min_pixels=None*, *npixels=1000*, *walker_initial_pos=None*, *parallel='auto'*, *stages=3*, *stage_len=30*, *seed=None*)
Bases: `holopy.inference.emcee.EmceeStrategy`

add_stage_strategy(*nsamples*, *npixels*)

sample(*model*, *data*)

emcee_lnprobs_DataArray(*sampler*)

emcee_samples_DataArray(*sampler*, *parameter_names*)

sample_emcee(*model*, *data*, *nwalkers*, *nsamples*, *walker_initial_pos*, *parallel='auto'*, *seed=None*)

sample_one_sigma_gaussian(*result*)

fit(*data*, *model*, *parameters=None*, *strategy=None*)

make_default_model(*base_scatterer*, *fitting_parameters=None*)

make_uniform(*guesses*, *key*)

parameterize_scatterer(*base_scatterer*, *fitting_parameters*)

replace_center(*parameters*, *key*)

sample(*data*, *model*, *strategy=None*)

validate_strategy(*strategy*, *operation*)

class AlphaModel(*scatterer*, *alpha=1*, *noise_sd=None*, *medium_index=None*, *illum_wavelen=None*, *illum_polarization=None*, *theory='auto'*, *constraints=[]*)
Bases: `holopy.inference.model.Model`

Model of hologram image formation with scaling parameter alpha.

alpha

class ExactModel(*scatterer*, *calc_func=<function calc_holo>*, *noise_sd=None*, *medium_index=None*, *illum_wavelen=None*, *illum_polarization=None*, *theory='auto'*, *constraints=[]*)
Bases: `holopy.inference.model.Model`

Model of arbitrary scattering function given by calc_func.

class LimitOverlaps(*fraction=0.1*)
Bases: `holopy.core.holopy_object.HoloPyObject`

Constraint prohibiting overlaps beyond a certain tolerance. fraction is the largest overlap allowed, in terms of sphere diameter.

check(*s*)

class Model(*scatterer*, *noise_sd=None*, *medium_index=None*, *illum_wavelen=None*, *illum_polarization=None*, *theory='auto'*, *constraints=[]*)

Bases: `holopy.core.holopy_object.HoloPyObject`

Model probabilites of observing data

Compute probabilities that observed data could be explained by a set of scatterer and observation parameters.

add_tie(*parameters_to_tie*, *new_name=None*)

Defines new ties between model parameters

Parameters

- **parameters_to_tie** (`listlike`) – names of parameters to tie, as given by keys in model.parameters
- **new_name** (`string, optional`) – the name for the new tied parameter

ensure_parameters_are_listlike(*pars*)

fit(*data*, *strategy=None*)

forward(*pars*, *detector*)

classmethod from_yaml(*loader*, *node*)

Convert a representation node to a Python object.

generate_guess(*n=1*, *scaling=1*, *seed=None*)

illum_polarization

illum_wavelen

initial_guess

dictionary of initial guess values for each parameter

initial_guess_scatterer

lnlike(*pars*, *data*)

Compute the log-likelihood for pars given data

Parameters

- **pars** (`dict or list`) – list - values for each parameter in the order of self._parameters dict - keys should match self.parameters
- **data** (`xarray`) – The data to compute likelihood against

Returns lnlike

Return type float

lnposterior(*pars*, *data*, *pixels=None*)

Compute the log-posterior probability of pars given data

Parameters

- **pars** (`dict or list`) – list - values for each parameter in the order of self._parameters dict - keys should match self.parameters
- **data** (`xarray`) – The data to compute posterior against
- **pixels** (`int (optional)`) – Specify to use a random subset of all pixels in data

Returns lnposterior

Return type float

lnprior(*pars*)
Compute the log-prior probability of pars

Parameters pars (*dict or list*) – list - values for each parameter in the order of self._parameters dict - keys should match self.parameters

Returns lnprior

Return type float

medium_index

noise_sd

parameters
dictionary of the model's parameters

sample(*data*, *strategy=None*)

scatterer

scatterer_from_parameters(*pars*)
Creates a scatterer by setting values for model parameters

Parameters pars (*dict or list*) – list - values for each parameter in the order of self._parameters dict - keys should match self.parameters

Returns

Return type scatterer

theory_from_parameters(*pars*)

Interfaces to minimizers. Classes here provide a common interface to a variety of third party minimizers.

class NmpfitStrategy(*npixels=None*, *quiet=True*, *ftol=1e-10*, *xtol=1e-10*, *gtol=1e-10*, *damp=0*, *maxiter=100*, *seed=None*)
Bases: `holopy.core.holopy_object.HoloPyObject`

Levenberg-Marquardt minimizer, from Numpy/Python translation of Craig Markwardt's mpfit.pro.

Parameters

- **npixels** (*None*) – Fit only a randomly selected fraction of the data points in data
- **quiet** (*Boolean*) – If False, print output on minimizer convergence. Default is True
- **ftol** (*float*) – Convergence criterion for minimizer: converges if actual and predicted relative reductions in chi squared <= ftol
- **xtol** (*float*) – Convergence criterion for minimizer: converges if relative error between two Levenberg-Marquardt iterations is <= xtol
- **gtol** (*float*) – Convergence criterion for minimizer: converges if absolute value of cosine of angle between vector of cost function evaluated at current solution for minimized parameters and any column of the Jacobian is <= gtol
- **damp** (*float*) – If nonzero, residuals larger than damp will be replaced by tanh. See nmpfit documentation.
- **maxiter** (*int*) – Maximum number of Levenberg-Marquardt iterations to be performed.

Notes

See nmpfit documentation for further details. Not all functionalities of nmpfit are implemented here: in particular, we do not allow analytical derivatives of the residual function, which is impractical and/or impossible to calculate for holograms. If you want to weight the residuals, you need to supply a custom residual function.

`calc_residuals`(*par_vals*)

`cleanup_from_fit`()

`fit`(*model*, *data*)
fit a model to some data

Parameters

- **`model`** (`Model` object) – A model describing the scattering system which leads to your data and the parameters to vary to fit it to the data
- **`data`** (`xarray.DataArray`) – The data to fit

Returns **result** – an object containing the best fit parameters and information about the fit

Return type `FitResult`

`get_errors_from_minimizer`(*fitted_pars*)

`initialize_fit`(*model*, *data*)

`minimize`(*parameters*, *obj_func*)

`unscale_pars_from_minimizer`(*values*)

Results of sampling

class `FitResult`(*data*, *model*, *strategy*, *time*, *kwargs={}*)
Bases: `holopy.core.holopy_object.HoloPyObject`

`add_attr`(*kwargs*)

`best_fit`()

`forward`(*pars*)

`guess_hologram`

`guess_parameters`

`guess_scatterer`

`hologram`

`max_lnprob`

`parameters`

`scatterer`

class `SamplingResult`(*data*, *model*, *strategy*, *time*, *kwargs={}*)
Bases: `holopy.inference.result.FitResult`

`burn_in`(*sample_number*)

class `TemperedSamplingResult`(*end_result*, *stage_results*, *strategy*, *time*)
Bases: `holopy.inference.result.SamplingResult`

class UncertainValue(*guess*, *plus*, *minus=None*, *name=None*)
Bases: `holopy.core.holopy_object.HoloPyObject`

Represent an uncertain value

Parameters

- **value** (`float`) – The value
- **plus** (`float`) – The plus n_sigma uncertainty (or the uncertainty if it is symmetric)
- **minus** (`float or None`) – The minus n_sigma uncertainty, or None if the uncertainty is symmetric
- **n_sigma** (`int (or float)`) – The number of sigma the uncertainties represent

class LeastSquaresScipyStrategy(*ftol=1e-10*, *xtol=1e-10*, *gtol=1e-10*, *max_nfev=None*, *npixels=None*)
Bases: `holopy.core.holopy_object.HoloPyObject`

fit(*model*, *data*)
fit a model to some data

Parameters

- **model** (`Model` object) – A model describing the scattering system which leads to your data and the parameters to vary to fit it to the data
- **data** (`xarray.DataArray`) – The data to fit

Returns **result** – Contains the best fit parameters and information about the fit

Return type `FitResult`

minimize(*parameters*, *residuals_function*)

unscale_pars_from_minimizer(*parameters*, *values*)

4.1.3 holopy.propagation package

Submodules

Code to propagate objects/waves using scattering models.

propagate(*data*, *d*, *medium_index=None*, *illum_wavelen=None*, *cfsp=0*, *gradient_filter=False*)
Propagates a hologram along the optical axis

Parameters

- **data** (`xarray.DataArray`) – Hologram to propagate
- **d** (`float or list of floats`) – Distance to propagate or desired schema. A list tells to propagate to several distances and return the volume
- **cfsp** (`integer (optional)`) – Cascaded free-space propagation factor. If this is an integer > 0, the transfer function G will be calculated at d/csf and the value returned will be G**csf. This helps avoid artifacts related to the limited window of the transfer function
- **gradient_filter** (`float`) – For each distance, compute a second propagation a distance gradient_filter away and subtract. This enhances contrast of rapidly varying features. You may wish to use the number that is a multiple of the medium wavelength (illum_wavelen / medium_index)

Returns **data** – The hologram progagated to a distance d from its current location.

Return type xarray.DataArray

Notes

holopy is agnostic to units, and the propagation result will be correct as long as the distance and wavelength are in the same units.

trans_func (*schema*, *d*, *med_wavelen*, *cfsp=0*, *gradient_filter=0*)
Calculates the optical transfer function to use in reconstruction

This routine uses the analytical form of the transfer function found in in Kreis[1]. It can optionally do cascaded free-space propagation for greater accuracy[2], although the code will run slightly more slowly.

Parameters

- **schema** (*xarray.DataArray*) – Hologram to obtain the maximum dimensions of the transfer function
- **d** (*float or list of floats*) – Reconstruction distance. If list or array, this function will return an array of transfer functions, one for each distance
- **med_wavelen** (*float*) – The wavelength in the medium you are propagating through
- **cfsp** (*integer (optional)*) – Cascaded free-space propagation factor. If this is an integer > 0, the transfer function G will be calculated at d/csf and the value returned will be G**csf
- **gradient_filter** (*float (optional)*) – Subtract a second transfer function a distance gradient_filter from each z

Returns trans_func – The calculated transfer function. This will be at most as large as shape, but may be smaller if the frequencies outside that are zero

Return type xarray.DataArray

References

interpolate2D (*data*, *i*, *j*, *fill=None*)
Interpolates values from a 2D array (data) given non-integer indecies i and j. If [i,j] is outside of the shape of data, fill is returned. If fill=None, the value of the closest edge pixel to (i,j) is used.

ps_propagate (*data*, *d*, *L*, *beam_c*, *out_schema=None*)
Propagates light back through a hologram that was taken using a diverging reference beam.

Parameters

- **is a holopy Xarray. It is the hologram to reconstruct. Must be** (*data*) –
- **The pixel spacing must also be square.** (*square.*) –
- **= distance from pinhole to reconstructed image, in meters (this is** (*d*) –
- **in Jericho and Kreuzer) Can be a scalar or a 1D list or array.** (*z*) –
- **= distance from screen to pinhole, in meters** (*L*) –

[1] Kreis, Handbook of Holographic Interferometry (Wiley, 2005), equation 3.79 (page 116)
[2] Kreis, Optical Engineering 41(8):1829, section 5

- **= [x,y] coodinates of beam center, in pixels** (*beam_c*) –
- **= size of output image and pixel spacing, default is the schema** (*out_schema*) –
- **data.** (*of*) –

Returns

Return type an image(volume) corresponding to the reconstruction at plane(s) d.

Notes

Only propagation through media with refractive index 1 is supported. This is a wrapper function for ps_propagate_plane() This function can handle a single reconstruction plane or a volume.

Based on the algorithm described in Manfred H. Jericho and H. Jurgen Kreuzer, "Point Source Digital In-Line Holographic Microscopy," Chapter 1 of Coherent Light Microscopy, Springer, 2010. http://link.springer.com/chapter/10.1007%2F978-3-642-15813-1_1

ps_propagate_plane(*data*, *d*, *L*, *beam_c*, *out_schema=None*, *old_Ip=False*)

Propagates light back through a hologram that was taken using a diverging reference beam.

Parameters

- **is a holopy Xarray. It is the hologram to reconstruct. Must be square.** (*data*) –
- **pixel spacing must also be square.** (*The*) –
- **= distance from pinhole to reconstructed image, in meters (this is z in** (*d*) –
- **and Kreuzer) Must be a scalar.** (*Jericho*) –
- **= distance from screen to pinhole, in meters** (*L*) –
- **= [x,y] coodinates of beam center, in pixels** (*beam_c*) –
- **= size of output image and pixel spacing, default is the schema** (*out_schema*) –
- **data.** (*of*) –
- **Ip == True, returns Ip to be used on calculations in the stack** (*if*) –
- **Ip == False compute reconstructed image as normal** (*if*) –
- **Ip is an image, use this to speed up calculations** (*if*) –

Returns

Return type returns an image(volume) corresponding to the reconstruction at plane(s) d.

Notes

Propataion can be to one plane only. Only propagation through media with refractive index 1 is supported.

Based on the algorithm described in Manfred H. Jericho and H. Jurgen Kreuzer, "Point Source Digital In-Line Holographic Microscopy," Chapter 1 of Coherent Light Microscopy, Springer, 2010. http://link.springer.com/chapter/10.1007%2F978-3-642-15813-1_1

4.1.4 holopy.scattering package

Scattering calculations

The scattering package provides objects and methods to define scatterer geometries, and theories to compute scattering from specified geometries. Scattering depends on holopy.core, and certain scattering theories may require external scattering codes.

The HoloPy scattering module is used to:

1. Describe geometry as a `scatterer` object
2. Define the result you want as a xarray.DataArray xarray.DataArray
3. Calculate scattering quantities with an `theory` appropriate for your `scatterer`

Subpackages

holopy.scattering.scatterer package

Modules for defining different types of scatterers, including scattering primitives such as Spheres, and more complex objects such as Clusters.

Submodules

Defines cylinder scatterers.

class Bisphere(*n=None*, *h=None*, *d=None*, *center=None*, *rotation=(0, 0, 0)*)
Bases: `holopy.scattering.scatterer.scatterer.CenteredScatterer`

Scattering object representing bisphere scatterers

Parameters

- **n** (*complex*) – Index of refraction
- **h** (*distance between centers*) –
- **d** (*diameter*) –
- **center** (*3-tuple, list or numpy array*) – specifies coordinates of center of the scatterer
- **rotation** (*3-tuple, list or numpy.array*) – specifies the Euler angles (alpha, beta, gamma) in radians defined in a-dda manual section 8.1

Defines capsule scatterers.

class Capsule(*n=None*, *h=None*, *d=None*, *center=None*, *rotation=(0, 0, 0)*)
Bases: `holopy.scattering.scatterer.scatterer.CenteredScatterer`

A cylinder with semi-spherical caps.

A particle with no rotation has its long axis pointing along +z, specify other orientations by euler angle rotations from that reference.

Parameters

- **n** (*complex*) – Index of refraction
- **h** (*height of cylinder*) –

- **d** (*diameter*) –
- **center** (*3-tuple, list or numpy array*) – specifies coordinates of center of the scatterer
- **rotation** (*3-tuple, list or numpy.array*) – specifies the Euler angles (alpha, beta, gamma) in radians

indicators

Defines Scatterers, a scatterer that consists of other scatterers, including scattering primitives (e.g. Sphere) or other Scatterers scatterers (e.g. two trimers).

class Scatterers (*scatterers=None*)

Bases: *holopy.scattering.scatterer.scatterer.Scatterer*

Contains optical and geometrical properties of a a composite scatterer. A Scatterers can consist of multiple scattering primitives (e.g. Sphere) or other Scatterers scatterers.

scatterers

List of scatterers that make up this object

Type list

parameters [property]

Dictionary of composite's parameters

add (*scatterer*)

Adds a new scatterer to the composite.

from_parameters ()

translated ()

rotated ()

Notes

Stores information about components in a tree. This is the most generic container for a collection of scatterers.

add (*scatterer*)

from_parameters (*new_parameters*)

Makes a new object similar to self with values as given in parameters. This returns a physical object, so any priors are replaced with their guesses if not included in passed-in parameters.

Parameters

- **parameters** (*dict*) – dictionary of parameters to use in the new object. Keys should match those of self.parameters.
- **overwrite** (*bool (optional)*) – if True, constant values are replaced by those in parameters

get_component_list ()

in_domain (*points*)

Tell which domain of a scatterer points are in

Parameters points (*np.ndarray (Nx3)*) – Point or list of points to evaluate

Returns domain – The domain of each point. Domain 0 means not in the particle

Return type np.ndarray (N)

`index_at`(*point*)

`rotated`(*ang1*, *ang2=None*, *ang3=None*)

`translated`(*coord1*, *coord2=None*, *coord3=None*)

Make a copy of this scatterer translated to a new location

Parameters **`y, z`** (*`x,`*) – Value of the translation along each axis

Returns **translated** – A copy of this scatterer translated to a new location

Return type *Scatterer*

Do Constructive Solid Geometry (CSG) with scatterers. Currently only useful with the DDA th

class `CsgScatterer`(*s1*, *s2*)

Bases: *`holopy.scattering.scatterer.scatterer.Scatterer`*

`bounds`

`rotated`(*alpha*, *beta*, *gamma*)

class `Difference`(*s1*, *s2*)

Bases: *`holopy.scattering.scatterer.csg.CsgScatterer`*

`bounds`

`in_domain`(*points*)

Tell which domain of a scatterer points are in

Parameters **`points`** (*`np.ndarray (Nx3)`*) – Point or list of points to evaluate

Returns **domain** – The domain of each point. Domain 0 means not in the particle

Return type np.ndarray (N)

class `Intersection`(*s1*, *s2*)

Bases: *`holopy.scattering.scatterer.csg.CsgScatterer`*

`in_domain`(*points*)

Tell which domain of a scatterer points are in

Parameters **`points`** (*`np.ndarray (Nx3)`*) – Point or list of points to evaluate

Returns **domain** – The domain of each point. Domain 0 means not in the particle

Return type np.ndarray (N)

class `Union`(*s1*, *s2*)

Bases: *`holopy.scattering.scatterer.csg.CsgScatterer`*

`in_domain`(*points*)

Tell which domain of a scatterer points are in

Parameters **`points`** (*`np.ndarray (Nx3)`*) – Point or list of points to evaluate

Returns **domain** – The domain of each point. Domain 0 means not in the particle

Return type np.ndarray (N)

Defines cylinder scatterers.

class `Cylinder`(*n=None*, *h=None*, *d=None*, *center=None*, *rotation=(0, 0, 0)*)

Bases: *`holopy.scattering.scatterer.scatterer.CenteredScatterer`*

Scattering object representing cylinder scatterers

Parameters

- **n** (*complex*) – Index of refraction
- **h** (*height of cylinder*) –
- **d** (*diameter*) –
- **center** (*3-tuple, list or numpy array*) – specifies coordinates of center of the scatterer
- **rotation** (*3-tuple, list or numpy.array*) – specifies the Euler angles (alpha, beta, gamma) in radians defined in a-dda manual section 8.1

Defines ellipsoidal scatterers.

class Ellipsoid (*n=None, r=None, center=None, rotation=(0, 0, 0)*)
Bases: *holopy.scattering.scatterer.scatterer.CenteredScatterer*

Scattering object representing ellipsoidal scatterers

Parameters

- **n** (*complex*) – Index of refraction
- **r** (*float or (float, float, float)*) – x, y, z semi-axes of the ellipsoid
- **center** (*3-tuple, list or numpy array*) – specifies coordinates of center of the scatterer
- **rotation** (*3-tuple, list or numpy.array*) – specifies the Euler angles (alpha, beta, gamma) in radians defined in a-dda manual section 8.1

indicators
Ellipsoid indicators does not currently apply rotations

Type NOTE

Defines two types of Janus (two faced) Spheres as scattering primitives.

class JanusSphere_Tapered (*n=None, r=None, rotation=(0, 0), center=None*)
Bases: *holopy.scattering.scatterer.scatterer.CenteredScatterer*

indicators

class JanusSphere_Uniform (*n=None, r=None, rotation=(0, 0, 0), center=None*)
Bases: *holopy.scattering.scatterer.scatterer.CenteredScatterer*

indicators

The abstract base class for all scattering objects

class CenteredScatterer (*center=None*)
Bases: *holopy.scattering.scatterer.scatterer.Scatterer*

class Indicators (*functions, bound=None*)
Bases: *holopy.core.holopy_object.HoloPyObject*

Class holding functions describing a scatterer

One or more functions (one per domain) that take Nx3 arrays of points and return a boolean array of membership in each domain. More than one indicator is allowed to return true for a given point, in that case the point is considered a member of the first domain with a true value.

class Scatterer (*indicators, n, center*)
Bases: *holopy.core.holopy_object.HoloPyObject*

Base class for scatterers

bounds

contains(*points*)

from_parameters(*parameters*)
Create a Scatterer from a dictionary of parameters

Parameters **parameters** (*dict*) – Parameters for a scatterer. This should be of the form returned by Scatterer.parameters.

Returns **scatterer** – A scatterer with the given parameter values

Return type Scatterer class

in_domain(*points*)
Tell which domain of a scatterer points are in

Parameters **points** (*np.ndarray (Nx3)*) – Point or list of points to evaluate

Returns **domain** – The domain of each point. Domain 0 means not in the particle

Return type np.ndarray (N)

index_at(*points*, *background=0*)

num_domains

parameters
Get a dictionary of this scatterer's parameters

Parameters **None** –

Returns **parameters** – A dictionary of this scatterer's parameters. This dict can be passed to Scatterer.from_parameters to make a copy of this scatterer

Return type dict

translated(*coord1*, *coord2=None*, *coord3=None*)
Make a copy of this scatterer translated to a new location

Parameters **y, z** (*x,*) – Value of the translation along each axis

Returns **translated** – A copy of this scatterer translated to a new location

Return type *Scatterer*

voxelate(*spacing*, *medium_index=0*)
Represent a scatterer by discretizing into voxels

Parameters

- **spacing** (*float*) – The spacing between voxels in the returned voxelation
- **medium_index** (*float*) – The background index of refraction to fill in at regions where the scatterer is not present

Returns **voxelation** – An array with refractive index at every pixel

Return type np.ndarray

voxelate_domains(*spacing*)

x

y

z

bound_union(*d1*, *d2*)

find_bounds(*indicator*)

Finds the bounds needed to contain an indicator function

Notes

Will probably determine incorrect bounds for functions which are not convex

Defines Sphere, a scattering primitive

class LayeredSphere(*n=None*, *t=None*, *center=None*)

Bases: `holopy.scattering.scatterer.sphere.Sphere`

Alternative description of a sphere where you specify layer thicknesses instead of radii

n

Index of each each layer

Type list of complex

t

Thickness of each layer

Type list of float

center

specifies coordinates of center of sphere

Type length 3 listlike

r

class Sphere(*n=None*, *r=0.5*, *center=None*)

Bases: `holopy.scattering.scatterer.scatterer.CenteredScatterer`

Contains optical and geometrical properties of a sphere, a scattering primitive.

This can be a multiple layered sphere by making r and n lists.

n

index of refraction of each layer of the sphere

Type complex or list of complex

r

radius of the sphere or outer radius of each sphere.

Type float or list of float

center

specifies coordinates of center of sphere

Type length 3 listlike

indicators

num_domains

rotated(*alpha*, *beta*, *gamma*)

Defines Spheres, a Scatterers scatterer consisting of Spheres

class RigidCluster(*spheres*, *translation=(0, 0, 0)*, *rotation=(0, 0, 0)*)

Bases: `holopy.scattering.scatterer.spherecluster.Spheres`

from_parameters(*parameters*)

Makes a new object similar to self with values as given in parameters. This returns a physical object, so any priors are replaced with their guesses if not included in passed-in parameters.

Parameters

- **parameters** (*dict*) – dictionary of parameters to use in the new object. Keys should match those of self.parameters.
- **overwrite** (*bool (optional)*) – if True, constant values are replaced by those in parameters

scatterers

class Spheres(*scatterers*, *warn=True*)

Bases: *holopy.scattering.scatterer.composite.Scatterers*

Contains optical and geometrical properties of a cluster of spheres.

spheres

Spheres which will make up the cluster

Type list of Spheres

warn

if True, overlapping spheres raise warnings.

Type bool

Notes

add(*scatterer*)

center

centers

largest_overlap()

n

n_imag

n_real

overlaps

r

x

y

z

Defines spheroidal scatterers.

class Spheroid(*n=None*, *r=None*, *rotation=(0, 0, 0)*, *center=None*)

Bases: *holopy.scattering.scatterer.scatterer.CenteredScatterer*

Scattering object representing spheroidal scatterers

n

Index of refraction

Type complex

r
length of xy and z semi-axes of the spheroid

Type (float, float)

rotation
specifies the Euler angles (alpha, beta, gamma) in radians

Type 3-tuple, list or numpy array

center
specifies coordinates of center of the scatterer

Type 3-tuple, list or numpy array

indicators

holopy.scattering.theory package

Theories to compute scattering from objects.

All theories have a common interface defined by *holopy.scattering.theory.scatteringtheory.ScatteringTheory*.

Subpackages

holopy.scattering.theory.mie_f package

Fortran extension module for calculating cluster holograms using tmatrix scattering theory.

Submodules

Compute special functions needed for the computation of scattering coefficients in the Lorenz-Mie scattering solution and related problems such as layered spheres.

These functions are not to be used for calculations at each field point. Rather, they should be used once for the calculation of scattering coefficients, which then get passed to faster Fortran code for field calculations.

Papers referenced herein:

D. W. Mackowski, R. A. Altenkirch, and M. P. Menguc, "Internal absorption cross sections in a stratified sphere," Applied Optics 29, 1551-1559, (1990).

Yang, "Improved recursive algorithm for light scattering by a multilayered sphere," Applied Optics 42, 1710-1720, (1993).

Qratio(*z1*, *z2*, *nstop*, *dns1=None*, *dns2=None*, *eps1=0.001*, *eps2=1e-16*)
Calculate ratio of Riccati-Bessel functions defined in [Yang2003] eq. 23 by up recursion.

Notes

Logarithmic derivatives calculated automatically if not specified. Lentz continued fraction algorithm used to start downward recursion for logarithmic derivatives.

R_psi(*z1*, *z2*, *nmax*, *eps1=0.001*, *eps2=1e-16*)
Calculate ratio of Riccati-Bessel function psi: psi(z1)/psi(z2).

Notes

See [Mackowski1990] eqns. 65-66. Uses Lentz continued fraction algorithm for logarithmic derivatives.

log_der_1(*z*, *nmx*, *nstop*)

Computes logarithmic derivative of Riccati-Bessel function psi_n(z) by downward recursion as in BHMIE.

Parameters

- **z** (*complex argument*) –
- **nmx** (*order from which downward recursion begins.*) –
- **nstop** (*integer, maximum order*) –

Notes

psi_n(z) is related to the spherical Bessel function j_n(z). Consider implementing Lentz's continued fraction method.

log_der_13(*z*, *nstop*, *eps1=0.001*, *eps2=1e-16*)

Calculate logarithmic derivatives of Riccati-Bessel functions psi and xi for complex arguments. Riccati-Bessel conventions follow Bohren & Huffman.

See Mackowski et al., Applied Optics 29, 1555 (1990).

Parameters

- **z** (*complex number*) –
- **nstop** (*maximum order of computation*) –
- **eps1** (*underflow criterion for Lentz continued fraction for Dn1*) –
- **eps2** (*convergence criterion for Lentz continued fraction for Dn1*) –

riccati_psi_xi(*x*, *nstop*)

Calculate Riccati-Bessel functions psi and xi for real argument.

Parameters

- **x** (*float*) – Argument
- **nstop** (*int*) – Maximum order to calculate to

Returns psi and xi

Return type ndarray(2, nstop)

Notes

Uses upwards recursion.

MieScatLib.py

Library of code to do Mie scattering calculations.

asymmetry_parameter(*al*, *bl*)

Calculate asymmetry parameter of scattered field.

Parameters **bn** (*an,*) – coefficient arrays from Mie solution

Returns

Return type float

Notes

See discussion on Bohren & Huffman p. 120. The output of this function omits the prefactor of 4/(x^2 Q_sca).

`cross_sections`(*al*, *bl*)

Calculates scattering and extinction cross sections given arrays of Mie scattering coefficients an and bn.

Parameters **`bn`** (*`an,`*) – coefficient arrays from Mie solution

Returns Scattering, extinction, and radar backscattering cross sections

Return type ndarray(3)

Notes

See Bohren & Huffman eqns. 4.61 and 4.62. The output omits a scaling prefactor of 2 * pi / k^2.

`internal_coeffs`(*m*, *x*, *n_max*, *eps1=0.001*, *eps2=1e-16*)

Calculate internal Mie coefficients c_n and d_n given relative index, size parameter, and maximum order of expansion.

Parameters **`docstring for scatcoeffs`** (*`See`*) –

Returns Internal coefficients c_n and d_n

Return type ndarray(2,n) complex

Notes

Follow Bohren & Huffman's convention. Note that van de Hulst and Kerker have different conventions (labeling of c_n and d_n and factors of m) for their internal coefficients.

`nstop`(*x*)

Calculate maximum expansion order of Lorenz-Mie solution.

Parameters **`x`** (*`float`*) – Particle size parameter

Returns **nstop**

Return type int

Notes

Criterion taken from [Wiscombe1980].

`scatcoeffs`(*m*, *x*, *nstop*, *eps1=0.001*, *eps2=1e-16*)

Calculate expansion coefficients for scattered field in Lorenz-Mie solution.

Parameters

- **`m`** (*`complex`*) – Sphere relative refractive index (n_sphere / n_medium)
- **`x`** (*`float`*) – Sphere size parameter (k_med * a)
- **`nstop`** (*`int`*) – Maximum order of scattered field expansion

- **eps1** (*float, optional*) – In Lentz continued fraction algorithm for logarithmic derivative D_n(z), value of continued fraction numerator or denominator triggering ill-conditioning workaround.
- **eps2** (*float, optional*) – Convergence criterion for Lentz continued fraction algorithm

Returns Scattering coefficients a_n and b_n

Return type array(2, nstop), complex

Notes

Uses formula for scattering coefficients based on logarithmic derivative D_n(z) of spherical Bessel function psi_n(z). See [Bohren1983] eq. 4.88.

Following BHMIE, calculates D_n for complex argument using downward recursion, and Riccati-Bessel functions psi and xi for real argument using upward recursion.

Initializes downward recursion for D_n using Lentz continued fraction algorithm [Lentz1976].

multilayer_sphere_lib.py

Author: Jerome Fung (fung@physics.harvard.edu)

Functions to calculate the scattering from a spherically symmetric particle with an arbitrary number of layers with different refractive indices.

Key reference for multilayer algorithm: Yang, "Improved recursive algorithm for light scattering by a multilayered sphere," Applied Optics 42, 1710-1720, (1993).

scatcoeffs_multi(*marray*, *xarray*, *eps1=0.001*, *eps2=1e-16*)

Calculate scattered field expansion coefficients (in the Mie formalism) for a particle with an arbitrary number of spherically symmetric layers.

Parameters

- **marray** (*array_like, complex128*) – array of layer indices, innermost first
- **xarray** (*array_like, real*) – array of layer size parameters (k * outer radius), innermost first
- **eps1** (*float, optional*) – underflow criterion for Lentz continued fraction for Dn1
- **eps2** (*float, optional*) – convergence criterion for Lentz continued fraction for Dn1

Returns **scat_coeffs** – Scattering coefficients

Return type ndarray (complex)

Extensions for T-Matrix scattering calculations (in fortran77 and fortran90); numpy.distutils should automatically use f2py to compile these, and f2py should detect your fortran compiler.

The code works with gcc, but has not been tested with other compilers. Note that f2py by default compiles with optimization flags.

Ignore compiler warnings of unused variables, unused dummy arguments, and variables being used uninitialized from compiling scsmfo_min. The former is relics of how scsmfo was written which I am not touching. The latter is likely due to some GOTO statements that could cause a variable to be referenced before it's initialized. Under normal usage I wouldn't worry about it.

configuration(*parent_package=''*, *top_path=None*)

holopy.scattering.theory.tmatrix_f package

Submodules

configuration(*parent_package=", top_path=None*)

Submodules

Compute holograms using the discrete dipole approximation (DDA). Currently uses ADDA (https://github.com/adda-team/adda) to do DDA calculations. .. moduleauthor:: Thomas G. Dimiduk <tdimiduk@physics.harvard.edu>

class DDA(*n_cpu=1, use_gpu=False, gpu_id=None, max_dpl_size=None, use_indicators=True, keep_raw_calculations=False, addacmd=[], suppress_C_output=True*)

Bases: `holopy.scattering.theory.scatteringtheory.ScatteringTheory`

Computes scattering using the the Discrete Dipole Approximation (DDA). It can (in principle) calculate scattering from any arbitrary scatterer. The DDA uses a numerical method that represents arbitrary scatterers as an array of point dipoles and then self-consistently solves Maxwell's equations to determine the scattered field. In practice, this model can be extremely computationally intensive, particularly if the size of the scatterer is larger than the wavelength of light. This model requires an external scattering code: a-dda

n_cpu

Number of threads to use for the DDA calculation

Type int (optional)

max_dpl_size

Force a maximum dipole size. This is useful for forcing extra dipoles if necessary to resolve features in an object. This may make dda calculations take much longer.

Type float (optional)

use_indicators

If true, a scatterer's indicators method will be used instead of its built-in adda definition

Type bool

keep_raw_calculations

If true, do not delete the temporary file we run ADDA in, instead print its path so you can inspect its raw results

Type bool

Notes

Does not handle near fields. This introduces ~5% error at 10 microns. This can in principle handle any scatterer, but in practice it will need excessive memory or computation time for particularly large scatterers.

classmethod can_handle(*scatterer*)

Given a scatterer, returns a bool

raw_scat_matrs(*scatterer, pos, medium_wavevec, medium_index*)

Given a (3, N) array *pos* etc, returns an (N, 2, 2) array

required_spacing(*bounds, medium_wavelen, medium_index, n*)

class Lens(*lens_angle, theory, quad_npts_theta=100, quad_npts_phi=100, use_numexpr=True*)

Bases: `holopy.scattering.theory.scatteringtheory.ScatteringTheory`

Wraps a ScatteringTheory and overrides the raw_fields to include the effect of an objective lens.

`can_handle`(*scatterer*)
Given a scatterer, returns a bool

`desired_coordinate_system = 'cylindrical'`

`numexpr_integrand_prefactor1 = 'exp(1j * krho_p * sintheta * cos(phi_relative))'`

`numexpr_integrand_prefactor2 = 'exp(1j * kz_p * (1 - costheta))'`

`numexpr_integrand_prefactor3 = 'sqrt(costheta) * sintheta * phi_wts * theta_wts'`

`numexpr_integrandl = 'prefactor * (cosphi * (cosphi * S2 + sinphi * S3) + sinphi * (co`

`numexpr_integrandr = 'prefactor * (sinphi * (cosphi * S2 + sinphi * S3) - cosphi * (co`

`parameter_names = ('lens_angle',)`

`raw_fields`(*positions*, *scatterer*, *medium_wavevec*, *medium_index*, *illum_polarization*)
Given a (3, N) array *pos*, etc, returns a (3, N) array

`gauss_legendre_pts_wts`(*a*, *b*, *npts=100*)
Quadrature points for integration on interval [a, b]

`pts_wts_for_phi_integrals`(*npts*)
Quadrature points for integration on the periodic interval [0, pi]

Since this interval is periodic, we use equally-spaced points with equal weights.

Calculates holograms of spheres using Fortran implementation of Mie theory. Uses superposition to calculate scattering from multiple spheres. Uses full radial dependence of spherical Hankel functions for scattered field.

`class Mie`(*compute_escat_radial=True*, *full_radial_dependence=True*, *eps1=0.01*, *eps2=1e-16*)
Bases: `holopy.scattering.theory.scatteringtheory.ScatteringTheory`

Compute scattering using the Lorenz-Mie solution.

This theory calculates exact scattering for single spheres and approximate results for groups of spheres. It does not account for multiple scattering, hence the approximation in the case of multiple spheres. Neglecting multiple scattering is a good approximation if the particles are sufficiently separated.

This model can also calculate the exact scattered field from a spherically symmetric particle with an arbitrary number of layers with differing refractive indices, using Yang's recursive algorithm ([Yang2003]).

By default, calculates radial component of scattered electric fields, which is nonradiative.

Currently, in calculating the Lorenz-Mie scattering coefficients, the maximum size parameter x = ka is limited to 1000.

`can_handle`(*scatterer*)
Given a scatterer, returns a bool

`raw_cross_sections`(*scatterer*, *medium_wavevec*, *medium_index*, *illum_polarization*)
Calculate scattering, absorption, and extinction cross sections, and asymmetry parameter for spherically symmetric scatterers.

Parameters **`scatterer`** (`scatterpy.scatterer` object) – spherically symmetric scatterer to compute for (Calculation would need to be implemented in a radically different way, via numerical quadrature, for sphere clusters)

Returns **cross_sections** – Dimensional scattering, absorption, and extinction cross sections, and <cos heta>

Return type array (4)

Notes

The radiation pressure cross section C_pr is given by C_pr = C_ext - <cos heta> C_sca.

The radiation pressure force on a sphere is

F = (n_med I_0 C_pr) / c

where I_0 is the incident intensity. See van de Hulst, p. 14.

raw_fields (*positions*, *scatterer*, *medium_wavevec*, *medium_index*, *illum_polarization*)
Given a (3, N) array *pos*, etc, returns a (3, N) array

raw_scat_matrs (*scatterer*, *pos*, *medium_wavevec*, *medium_index*)
Returns far-field amplitude scattering matrices (with theta and phi dependence only) – assume spherical wave asymptotic r dependence

class AberratedMieLens (*spherical_aberration=0.0*, *lens_angle=1.0*, *calculator_accuracy_kwargs={}*)
Bases: `holopy.scattering.theory.mielens.MieLens`

parameter_names = ('lens_angle', 'spherical_aberration')

class MieLens (*lens_angle=1.0*, *calculator_accuracy_kwargs={}*)
Bases: `holopy.scattering.theory.scatteringtheory.ScatteringTheory`

Exact scattering from a sphere imaged through a perfect lens.

Calculates holograms of spheres using an analytical solution of the Mie scattered field imaged by a perfect lens (see [Leahy2020]). Can use superposition to calculate scattering from multiple spheres.

See also:

`mielensfunctions.MieLensCalculator`

can_handle (*scatterer*)
Given a scatterer, returns a bool

desired_coordinate_system = 'cylindrical'

parameter_names = ('lens_angle',)

raw_fields (*positions*, *scatterer*, *medium_wavevec*, *medium_index*, *illum_polarization*)

Parameters

- **positions** (*(3, N) numpy.ndarray*) – The (k * rho, phi, z) coordinates, relative to the sphere, of the points to calculate the fields. Note that the radial coordinate is rescaled by the wavevector.
- **scatterer** (`scatterer.Sphere` object) –
- **medium_wavevec** (*float*) –
- **medium_index** (*float*) –
- **illum_polarization** (*2-element tuple*) – The (x, y) field polarizations.

class AberratedMieLensCalculator (*spherical_aberration=None*, ***kwargs*)
Bases: `holopy.scattering.theory.mielensfunctions.MieLensCalculator`

must_be_specified = ['particle_kz', 'index_ratio', 'size_parameter', 'lens_angle', 'sp

class AlBlFunctions
Bases: `object`

Group of functions for calculating the Mie scattering coefficients, used for expressing the scattered field in terms of vector spherical harmonics.

The coefficients *a_l*, *b_l* are defined as

..math:

```
a_l =
```

rac{psi_l(x) psi_l'(nx) - n psi_l(nx) psi_l'(x)}

{xi_l(x) psi_l'(nx) - n psi_l(nx) xi_l'(x)},

b_l =

rac{psi_l(nx) psi_l'(x) - n psi_l(x) psi_l'(nx)}

{psi_l(nx) xi_l'(x) - n xi_l(x) psi_l'(nx)},

where ψ_l and ξ_l are the Riccati-Bessel functions of the first and third kinds, respectively. The definitions used here follow those of van de Hulst **[1]_**, which differ from those used in Bohren and Huffman **[2]_**.

static calculate_al_bl(*index_ratio*, *size_parameter*, *l*)

Returns *a_l* and *b_l*; see class docstring.

Parameters

- **index_ratio** (*float*) – relative index of refraction
- **size_paramter** (*float*) – Size parameter
- **l** (*int, array-like*) – Order of scattering coefficient

Returns a_l, b_l

Return type numpy.ndarray

static riccati_psin(*n*, *z*, *derivative=False*)

Riccati-Bessel function of the first kind or its derivative.

$$\psi_n(z) = z\, j_n(z),$$

where $j_n(z)$ is the spherical Bessel function of the first kind.

Parameters

n [int, array_like] Order of the Bessel function (n >= 0).

z [complex or float, array_like] Argument of the Bessel function.

derivative [bool, optional] If True, the value of the derivative (rather than the function itself) is returned.

Returns psin

Return type ndarray

static riccati_xin(*order*, *z*, *derivative=False*)

Riccati-Bessel function of the third kind or its derivative.

$$\xi_n(z) = z\, h_n^{(1)}(z),$$

where $h_n^{(1)}(z)$ is the first spherical Hankel function.

Parameters

- **n** (*int, array_like*) – Order of the Bessel function (n >= 0).
- **z** (*complex or float, array_like*) – Argument of the Bessel function.
- **derivative** (*bool, optional*) – If True, the value of the derivative (rather than the function itself) is returned.

Returns xin

Return type ndarray

class MieLensCalculator(*particle_kz=None, index_ratio=None, size_parameter=None, lens_angle=None, quad_npts=100, interpolate_integrals='check', interpolator_window_size=30.0, interpolator_degree=32*)

Bases: object

calculate_incident_field()

This is here so (i) Any corrections in the theory to the scattered field

have an easy place to enter, and

(ii) Other modules can consistently use the same scattered field as this module.

calculate_scattered_field(*krho, phi*)

Calculates the field from a Mie scatterer imaged through a high-NA lens and excited with an electric field of unit strength directed along the optical axis.

ec{E}_{sc} = A left[I_{12} sin(2phi) hat{y} +

-I_{10} hat{x} + I_{12} cos(2phi) hat{x} +

-I_{20} hat{x} + -I_{22} cos(2phi) hat{x} + -I_{22} sin(2phi) hat{y}

ight]

krho, phi [numpy.ndarray] The position of the particle relative to the focal point of the lens, in (i) cylindrical coordinates and (ii) dimensionless wavevectur units. Must all be the same shape.

field_xcomp, field_ycomp [numpy.ndarray] The (x, y) components of the electric field at the detector, where the initial field is polarized in the x-direction. Same shape as krho, phi

This will have problems for large rho, z, because of the quadrature points. Empirically this problem happens for rho >~ 4 * quad_npts. Could be adaptive if needed….

calculate_total_field(*krho, phi*)

The total (incident + scattered) field at the detector

calculate_total_intensity(*krho, phi*)

must_be_specified = ['particle_kz', 'index_ratio', 'size_parameter', 'lens_angle']

class MieScatteringMatrix(*parallel_or_perpendicular='perpendicular', index_ratio=None, size_parameter=None, max_l=None*)

Bases: object

class PiecewiseChebyshevApproximant(*function, degree, window_breakpoints, *args*)

Bases: object

calculate_al_bl(*index_ratio, size_parameter, l*)

calculate_pil_taul(*theta, max_order*)

The 1st through Nth order angle dependent functions for Mie scattering, evaluated at theta. The functions :math'pi(heta)' and :math' au(heta) are defined as:

..math:

```
\pi_n(      heta) =
```

rac{1}{sin heta} P_n^1(cos heta)

au_n(heta) =

rac{mathrm{d}}{mathrm{d} heta} P_n^1(cos heta)

where P_n^m is the associated Legendre function. The functions are computed by upward recurrence using the relations

..math:

```
\pi_n =
```

rac{2n-1}{n-1}cos heta , pi_{n-1} - rac{n}{n-1}pi_{n-2}

au_n = n , cos heta , pi_n - (n+1)pi_{n-1}

beginning with $pi_0 = 0$ and $pi_1 = 1$

theta [array_like] angles (in radians) at which to evaluate the angular functions

max_order [int > 0] Order at which to halt iteration. Must be > 0

pi, tau [ndarray] 2D arrays with shape (len(theta), max_order) containing the values of the angular functions evaluated at theta up to order *max_order*

gauss_legendre_pts_wts(*a*, *b*, *npts=100*)
Quadrature points for integration on interval [a, b]

j2(*x*)
A fast J_2(x) defined in terms of other special functions

spherical_h1n(*n*, *z*, *derivative=False*)
Spherical Hankel function H_n(z) or its derivative

spherical_h2n(*n*, *z*, *derivative=False*)
Spherical Hankel function H_n(z) or its derivative

Defines Multisphere theory class, which calculates scattering for multiple spheres using the (exact) superposition method implemented in modified version of Daniel Mackowski's SCSMFO1B.FOR. Uses full radial dependence of spherical Hankel functions for the scattered field.

class Multisphere(*niter=200*, *eps=1e-06*, *meth=1*, *qeps1=1e-05*, *qeps2=1e-08*, *compute_escat_radial=False*, *suppress_fortran_output=True*)
Bases: `holopy.scattering.theory.scatteringtheory.ScatteringTheory`

Exact scattering from a cluster of spheres.

Calculate the scattered field of a collection of spheres through a numerical method that accounts for multiple scattering and near-field effects (see [Fung2011], [Mackowski1996]). This approach is much more accurate than Mie superposition, but it is also more computationally intensive. The Multisphere code can handle any number of spheres; see notes below for details.

niter
maximum number of iterations to use in solving the interaction equations

Type integer (optional)

meth
method to use to solve interaction equations. Set to 0 for biconjugate gradient; 1 for order-of-scattering

Type integer (optional)

eps
relative error tolerance in solution for interaction equations

Type float (optional)

qeps1
error tolerance used to determine at what order the single-sphere spherical harmonic expansion should be truncated

Type float (optional)

qeps2
error tolerance used to determine at what order the cluster spherical harmonic expansion should be truncated

Type float (optional)

Notes

According to Mackowski's manual for SCSMFO1B.FOR [1]_ and later papers [2]_, the biconjugate gradient is generally the most efficient method for solving the interaction equations, especially for dense arrays of identical spheres. Order-of-scattering may converge better for non-identical spheres.

Multisphere does not check for overlaps becaue overlapping spheres can be useful for getting fits to converge. The results to be sensible for small overlaps even though mathemtically speaking they are not xstrictly valid.

Currently, Multisphere does not calculate the radial component of scattered electric fields. This is a good approximation for large kr, since the radial component falls off as 1/kr^2.

scfodim.for contains three parameters, all integers:

- nod: Maximum number of spheres
- **nod: Maximum order of individual sphere expansions. Will depend on** size of largest sphere in cluster.
- **notd: Maximum order of cluster-centered expansion. Will depend on** overall size of cluster.

Changing these values will require recompiling Fortran extensions.

The maximum size parameter of each individual sphere in a cluster is currently limited to 1000, indepdently of the above scfodim.for parameters.

References

can_handle(*scatterer*)
Given a scatterer, returns a bool

raw_cross_sections(*scatterer*, *medium_wavevec*, *medium_index*, *illum_polarization*)
Calculate scattering, absorption, and extinction cross sections, and asymmetry parameter for sphere clusters with polarized incident light.

The extinction cross section is calculated from the optical theorem. The scattering cross section is calculated by numerical quadrature of the scattered field, and the absorption cross section is calculated from the difference of the extinction cross section and the scattering cross section.

Parameters **scatterer** (`scatterpy.scatterer` object) – sphere cluster to compute for

Returns **cross_sections** – Dimensional scattering, absorption, and extinction cross sections, and <cos heta>

Return type array (4)

raw_fields (*positions*, *scatterer*, *medium_wavevec*, *medium_index*, *illum_polarization*)
Given a (3, N) array *pos*, etc, returns a (3, N) array

raw_scat_matrs (*scatterer*, *pos*, *medium_wavevec*, *medium_index*)
Calculate far-field amplitude scattering matrices at multiple positions

normalize_polarization (*illum_polarization*)

class ScatteringTheory
Bases: `holopy.core.holopy_object.HoloPyObject`

Defines common interface for all scattering theories.

Subclasses must implement: * can_handle * raw_fields or raw_scat_matrs or both. * (optionally) raw_cross_sections,

Notes

Subclasses should implement the following methods to create a scatteringtheory that works for the following user-facing methods: * *calc_holo*: *raw_fields* or *raw_scat_matrs* * *calc_intensity*: *raw_fields* or *raw_scat_matrs* * *calc_field*: *raw_fields* or *raw_scat_matrs* * *calc_scat_matrix*: *raw_scat_matrs* * *calc_cross_sections*: *raw_cross_sections*

By default, ScatteringTheories computer *raw_fields* from the *raw_scat_matrs*; over-ride the *raw_fields* method to compute the fields in a different way.

can_handle (*scatterer*)
Given a scatterer, returns a bool

desired_coordinate_system = 'spherical'

from_parameters (*parameters*)
Creates a ScatteringTheory like the current one, but with different parameters. Used for fitting

Parameters **dict** – keys should be valid *self.parameter_names* fields, values should be the corresponding kwargs

Returns

Return type ScatteringTheory instance, of the same class as *self*

parameter_names = ()

parameters

raw_cross_sections (*scatterer*, *medium_wavevec*, *medium_index*, *illum_polarization*)
Returns cross-sections, as an array [cscat, cabs, cext, asym]

raw_fields (*pos*, *scatterer*, *medium_wavevec*, *medium_index*, *illum_polarization*)
Given a (3, N) array *pos*, etc, returns a (3, N) array

raw_scat_matrs (*scatterer*, *pos*, *medium_wavevec*, *medium_index*)
Given a (3, N) array *pos* etc, returns an (N, 2, 2) array

Compute holograms using Mishchenko's T-matrix method for axisymmetric scatterers. Currently uses

class Tmatrix
Bases: `holopy.scattering.theory.scatteringtheory.ScatteringTheory`

Computes scattering using the axisymmetric T-matrix solution by Mishchenko with extended precision.

It can calculate scattering from axisymmetric scatterers such as cylinders and spheroids. Calculations for particles that are very large or have high aspect ratios may not converge.

Notes

Does not handle near fields. This introduces ~5% error at 10 microns.

`can_handle` (*scatterer*)
Given a scatterer, returns a bool

`raw_fields` (*pos*, *scatterer*, *medium_wavevec*, *medium_index*, *illum_polarization*)
Given a (3, N) array *pos*, etc, returns a (3, N) array

`raw_scat_matrs` (*scatterer*, *pos*, *medium_wavevec*, *medium_index*)
Given a (3, N) array *pos* etc, returns an (N, 2, 2) array

Submodules

Exceptions used in scatterpy module. These are separated out from the other exceptions in other parts of HoloPy to keep things modular.

`exception AutoTheoryFailed` (*scatterer*)
Bases: `Exception`

`exception InvalidScatterer` (*scatterer*, *message*)
Bases: `Exception`

`exception MissingParameter` (*parameter_name*)
Bases: `Exception`

`exception MultisphereFailure`
Bases: `Exception`

`exception OverlapWarning` (*scatterer*, *overlaps*)
Bases: `UserWarning`

`exception ParameterSpecificationError`
Bases: `Exception`

`exception TheoryNotCompatibleError` (*theory*, *scatterer*, *reason=None*)
Bases: `Exception`

`exception TmatrixFailure` (*logfilestr*)
Bases: `Exception`

`class ImageFormation` (*scattering_theory*)
Bases: `holopy.core.holopy_object.HoloPyObject`

Calculates fields, holograms, intensities, etc.

`calculate_cross_sections` (*scatterer*, *medium_wavevec*, *medium_index*, *illum_polarization*)

`calculate_scattered_field` (*scatterer*, *schema*)

Parameters **`scatterer`** (`scatterer` object) – (possibly composite) scatterer for which to compute scattering

Returns **e_field** – scattered electric field

Return type `VectorGrid`

calculate_scattering_matrix(*scatterer*, *schema*)
Compute scattering matrices for scatterer

Parameters **scatterer** (`holopy.scattering.scatterer` object) – (possibly composite) scatterer for which to compute scattering

Returns **scat_matr** – Scattering matrices at specified positions

Return type `Marray`

get_wavevec_from(*schema*)

select_scatterer_by_illumination(*scatterer*, *illum*)

Base class for scattering theories. Implements python-based calc_intensity and calc_holo, based on subclass's calc_field

calc_cross_sections(*scatterer*, *medium_index=None*, *illum_wavelen=None*, *illum_polarization=None*, *theory='auto'*)
Calculate scattering, absorption, and extinction cross sections, and asymmetry parameter <cos heta>.

Parameters

- **scatterer** (`scatterer` object) – (possibly composite) scatterer for which to compute scattering
- **medium_index** (*float or complex*) – Refractive index of the medium in which the scatter is imbedded
- **illum_wavelen** (*float or ndarray(float)*) – Wavelength of illumination light. If illum_wavelen is an array result will add a dimension and have all wavelengths
- **theory** (`theory` object (optional)) – Scattering theory object to use for the calculation. This is optional if there is a clear choice of theory for your scatterer. If there is not a clear choice, *calc_cross_sections* will error out and ask you to specify a theory

Returns **cross_sections** – Dimensional scattering, absorption, and extinction cross sections, and <cos theta>

Return type array (4)

calc_field(*detector*, *scatterer*, *medium_index=None*, *illum_wavelen=None*, *illum_polarization=None*, *theory='auto'*)
Calculate the scattered fields from a scatterer illuminated by a reference wave.

Parameters

- **detector** (*xarray object*) – The detector points and calculation metadata used to calculate the scattered fields.
- **scatterer** (`scatterer` object) – (possibly composite) scatterer for which to compute scattering
- **medium_index** (*float or complex*) – Refractive index of the medium in which the scatter is imbedded
- **illum_wavelen** (*float or ndarray(float)*) – Wavelength of illumination light. If illum_wavelen is an array result will add a dimension and have all wavelengths
- **theory** (`theory` object (optional)) – Scattering theory object to use for the calculation. This is optional if there is a clear choice of theory for your scatterer. If there is not a clear choice, *calc_field* will error out and ask you to specify a theory

Returns **e_field** – Calculated hologram from the given distribution of spheres

Return type `Vector` object

calc_holo(*detector*, *scatterer*, *medium_index=None*, *illum_wavelen=None*, *illum_polarization=None*, *theory='auto'*, *scaling=1.0*)

Calculate hologram formed by interference between scattered fields and a reference wave

Parameters

- **detector** (*xarray object*) – The detector points and calculation metadata used to calculate the hologram.
- **scatterer** (scatterer object) – (possibly composite) scatterer for which to compute scattering
- **medium_index** (*float or complex*) – Refractive index of the medium in which the scatter is imbedded
- **illum_wavelen** (*float or ndarray(float)*) – Wavelength of illumination light. If illum_wavelen is an array result will add a dimension and have all wavelengths
- **theory** (theory object (optional)) – Scattering theory object to use for the calculation. This is optional if there is a clear choice of theory for your scatterer. If there is not a clear choice, *calc_holo* will error out and ask you to specify a theory
- **scaling** (*scaling value (alpha) for amplitude of reference wave*) –

Returns **holo** – Calculated hologram from the given distribution of spheres

Return type xarray.DataArray

calc_intensity(*detector*, *scatterer*, *medium_index=None*, *illum_wavelen=None*, *illum_polarization=None*, *theory='auto'*)

Calculate intensity from the scattered field at a set of locations

Parameters

- **detector** (*xarray object*) – The detector points and calculation metadata used to calculate the intensity.
- **scatterer** (scatterer object) – (possibly composite) scatterer for which to compute scattering
- **medium_index** (*float or complex*) – Refractive index of the medium in which the scatter is imbedded
- **illum_wavelen** (*float or ndarray(float)*) – Wavelength of illumination light. If illum_wavelen is an array result will add a dimension and have all wavelengths
- **theory** (theory object (optional)) – Scattering theory object to use for the calculation. This is optional if there is a clear choice of theory for your scatterer. If there is not a clear choice, calc_intensity will error out and ask you to specify a theory

Returns **inten** – scattered intensity

Return type xarray.DataArray

calc_scat_matrix(*detector*, *scatterer*, *medium_index=None*, *illum_wavelen=None*, *theory='auto'*)

Compute farfield scattering matrices for scatterer

Parameters

- **detector** (*xarray object*) – The detector points and calculation metadata used to calculate the scattering matrices.
- **scatterer** (*holopy.scattering.scatterer* object) – (possibly composite) scatterer for which to compute scattering

- **medium_index** (*float or complex*) – Refractive index of the medium in which the scatter is imbedded
- **illum_wavelen** (*float or ndarray(float)*) – Wavelength of illumination light. If illum_wavelen is an array result will add a dimension and have all wavelengths
- **theory** (`theory` object (optional)) – Scattering theory object to use for the calculation. This is optional if there is a clear choice of theory for your scatterer. If there is not a clear choice, *calc_scat_matrix* will error out and ask you to specify a theory

Returns **scat_matr** – Scattering matrices at specified positions

Return type `Marray`

determine_default_theory_for(*scatterer*)

finalize(*detector*, *result*)

interpret_theory(*scatterer*, *theory='auto'*)

prep_schema(*detector*, *medium_index*, *illum_wavelen*, *illum_polarization*)

scattered_field_to_hologram(*scat*, *ref*)
Calculate a hologram from an E-field

Parameters

- **scat** (`VectorGrid`) – The scattered (object) field
- **ref** (*xarray[vector]]*) – The reference field

validate_scatterer(*scatterer*)

CHAPTER 5

References and credits

The following references describe applications of HoloPy and technical advances. If you use HoloPy, we ask that you cite the articles that are relevant to your application.

Ovryn and Izen and Lee and coworkers were the first to develop methods to fit scattering models to digital holograms:

The following papers describe different methods for calculating scattering and various algorithms that HoloPy uses in its calculations:

For scattering calculations and formalism, we draw heavily on the treatise of Bohren & Huffman. We generally follow their conventions except where noted.

For an introduction to Bayesian analysis of experimental data, we recommend

The package includes code from several sources. We thank Daniel Mackowski for allowing us to include his T-Matrix code, which computes scattering from clusters of spheres: SCSMFO1B.

We also make use of a modified version of the Python version of mpfit, originally developed by Craig Markwardt. The modified version we use is drawn from the stsci_python package.

We thank A. Ross Barnett for permitting us to use his routine SBESJY.FOR, which computes spherical Bessel functions.

HoloPy is based upon work supported by the National Science Foundation under grant numbers CBET-0747625, DMR-0820484, DMR-1306410, and DMR-1420570.

Bibliography

[Dimiduk2016] Dimiduk, Thomas G., and Vinothan N. Manoharan. "Bayesian Approach to Analyzing Holograms of Colloidal Particles." Optics Express 24, no. 21 (October 17, 2016): 24045–60. doi:10.1364/OE.24.024045.

[Wang2016] Wang, Anna, Rees F. Garmann, and Vinothan N. Manoharan. "Tracking E. Coli Runs and Tumbles with Scattering Solutions and Digital Holographic Microscopy." Optics Express 24, no. 21 (October 17, 2016): 23719–25. doi:10.1364/OE.24.023719.

[Dimiduk2014] Dimiduk, Thomas G., Rebecca W. Perry, Jerome Fung, and Vinothan N. Manoharan. "Random-Subset Fitting of Digital Holograms for Fast Three-Dimensional Particle Tracking." Applied Optics 53, no. 27 (September 20, 2014): G177–83. doi:10.1364/AO.53.00G177.

[Wang2014] Wang, Anna, Thomas G. Dimiduk, Jerome Fung, Sepideh Razavi, Ilona Kretzschmar, Kundan Chaudhary, and Vinothan N. Manoharan. "Using the Discrete Dipole Approximation and Holographic Microscopy to Measure Rotational Dynamics of Non-Spherical Colloidal Particles." Journal of Quantitative Spectroscopy and Radiative Transfer 146 (October 2014): 499–509. doi:10.1016/j.jqsrt.2013.12.019.

[Fung2013] Fung, Jerome, and Vinothan N. Manoharan. "Holographic Measurements of Anisotropic Three-Dimensional Diffusion of Colloidal Clusters." Physical Review E 88, no. 2 (August 30, 2013): 020302. doi:10.1103/PhysRevE.88.020302.

[Fung2012] Fung, Jerome, Rebecca W. Perry, Thomas G. Dimiduk, and Vinothan N. Manoharan. "Imaging Multiple Colloidal Particles by Fitting Electromagnetic Scattering Solutions to Digital Holograms." Journal of Quantitative Spectroscopy and Radiative Transfer 113, no. 18 (December 2012): 2482–89. doi:10.1016/j.jqsrt.2012.06.007.

[Kaz2012] Kaz, David M., Ryan McGorty, Madhav Mani, Michael P. Brenner, and Vinothan N. Manoharan. "Physical Ageing of the Contact Line on Colloidal Particles at Liquid Interfaces." Nature Materials 11, no. 2 (February 2012): 138–42. doi:10.1038/nmat3190.

[Perry2012] Perry, Rebecca W., Guangnan Meng, Thomas G. Dimiduk, Jerome Fung, and Vinothan N. Manoharan. "Real-Space Studies of the Structure and Dynamics of Self-Assembled Colloidal Clusters." Faraday Discussions 159, no. 1 (June 7, 2012): 211–34. doi:10.1039/C2FD20061A.

[Fung2011] Fung, Jerome, K. Eric Martin, Rebecca W. Perry, David M. Kaz, Ryan McGorty, and Vinothan N. Manoharan. "Measuring Translational, Rotational, and Vibrational Dynamics in Colloids with Digital Holographic Microscopy." Optics Express 19, no. 9 (April 25, 2011): 8051–65. doi:10.1364/OE.19.008051.

[Leahy2020] Leahy, Brian, Ronald Alexander, Caroline Martin, Solomon Barkley, and Vinothan N. Manoharan. "Large depth-of-field tracking of colloidal spheres in holographic microscopy by modeling the objective lens." Optics Express 28, no. 2 (2020): 1061-1075. doi:10.1364/OE.382159

[Ovryn2000] Ovryn, Ben, and Steven H. Izen. "Imaging of Transparent Spheres through a Planar Interface Using a High-Numerical-Aperture Optical Microscope." Journal of the Optical Society of America A 17, no. 7 (July 1, 2000): 1202–13. doi:10.1364/JOSAA.17.001202.

[Lee2007] Lee, Sang-Hyuk, Yohai Roichman, Gi-Ra Yi, Shin-Hyun Kim, Seung-Man Yang, Alfons van Blaaderen, Peter van Oostrum, and David G. Grier. "Characterizing and Tracking Single Colloidal Particles with Video Holographic Microscopy." Optics Express 15, no. 26 (December 24, 2007): 18275–82. doi:10.1364/OE.15.018275.

[Yurkin2011] Yurkin, Maxim A., and Alfons G. Hoekstra. "The Discrete-Dipole-Approximation Code ADDA: Capabilities and Known Limitations." Journal of Quantitative Spectroscopy and Radiative Transfer 112, no. 13 (September 2011): 2234–47. doi:10.1016/j.jqsrt.2011.01.031.

[Mackowski1990] Mackowski, D. W., Altenkirch, R. A., and Menguc, M. P. "Internal absorption cross sections in a stratified sphere." Applied Optics 29, no. 10 (1990). do:10.1364/AO.29.001551.

[Mackowski1996] Mackowski, Daniel W., and Michael I. Mishchenko. "Calculation of the T Matrix and the Scattering Matrix for Ensembles of Spheres." Journal of the Optical Society of America A 13, no. 11 (November 1, 1996): 2266. doi:10.1364/JOSAA.13.002266.

[Wiscombe1996] Wiscombe, J. "Mie Scattering Calculations: Advances in Technique and Fast, Vector-Speed Computer Codes," 1979. doi:10.5065/D6ZP4414.

[Wiscombe1980] Wiscombe, W. J. "Improved Mie scattering algorithms." Applied Optics 19, no. 9 (May 1, 1980): 1505. doi:10.1364/AO.19.001505.

[Yang2003] Yang, Wen. "Improved Recursive Algorithm for Light Scattering by a Multilayered Sphere." Applied Optics 42, no. 9 (March 20, 2003): 1710–20. doi:10.1364/AO.42.001710.

[Lentz1976] Lentz, William J. "Generating Bessel Functions in Mie Scattering Calculations Using Continued Fractions." Applied Optics 15, no. 3 (March 1, 1976): 668–71. doi:10.1364/AO.15.000668.

[Bohren1983] C. F. Bohren and D. R. Huffman, *Absorption and Scattering of Light by Small Particles*, Wiley (1983).

[Gregory2010] P. Gregory, *Bayesian Logical Data Analysis for the Physical Sciences*, Cambridge University Press (2010)

Python Module Index

Index