Contents

1 User Guide 3
  1.1 Getting Started ....................................................... 3
  1.2 Loading Data ............................................................ 4
  1.3 Reconstructing Data (Numerical Propagation) ....................... 8
  1.4 Reconstructing Point Source Holograms .......................... 10
  1.5 Scattering Calculations .............................................. 13
  1.6 Scattering from Arbitrary Structures with DDA ................... 20
  1.7 Bayesian inference of Parameter Values .......................... 22
  1.8 Fitting Models to Data ............................................... 25
  1.9 Developer’s Guide ..................................................... 28
  1.10 HoloPy Tools ........................................................... 30
  1.11 Concepts ............................................................... 31

2 holoPy package 33
  2.1 Module contents ..................................................... 33
  2.2 Subpackages ........................................................... 33

3 References and credits 75

Bibliography 77

Python Module Index 79
**Release 3.2.1**

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* to keep up to date on releases, answer questions, and benefit from other users’ questions.
Skip to the *Loading Data* tutorial if you already have HoloPy installed and want to get started quickly.

## 1.1 Getting Started

### 1.1.1 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])
1.1.2 Using HoloPy

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

```python
from holopy import test_disp
test_disp()
```

You should see a window pop up with 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:

```python
%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().

4 Chapter 1. User Guide

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.

An additional option in jupyter is to use %matplotlib nbagg to use inline interactive plots.

1.2 Loading Data

HoloPy can work with any kind of image data, but we use it for digital holograms, so our tutorials will focus mostly on hologram data.

1.2.1 Loading and viewing a hologram

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

```python
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:

```python
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 microns. In general, you should specify `spacing` as the distance between adjacent pixel centres. You can also load an image without specifying a spacing value if you just want to look at it, but most holopy calculations will give erroneous results on such an image.

The final line simply 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.

### 1.2.2 Correcting Noisy Images

The raw hologram has some non-uniform illumination and an artifact in the upper right hand corner from dust somewhere in the optics. These types of things can be removed if you are able to 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()` can usually improve the image a lot.

```python
from holopy.core.process import bg_correct
bgpath = get_example_data_path('bg01.jpg')
```

(continues on next page)
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)

Some convenient tools for manipulating image data are included within HoloPy. See the HoloPy Tools page for details.
1.2.3 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 information about 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, which has a refractive index of 1.33. It was illuminated by a red laser with wavelength of 660 nm and polarization in the x-direction. We can tell HoloPy all of this information when loading the image:

```python
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:

```python
holo = hp.core.update_metadata(holo, medium_index=1.33, illum_wavelen=0.660, illum_polarization=(1,0))
```

**Note:** Spacing and wavelength must both be written in the same units - microns 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.

1.2.4 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:

```python
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:

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

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

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

Additional options of `save_image()` allow you to control how image intensity is scaled. Images saved as .tif (the default) will still contain metadata, which will be retrieved if you reload with `load()`, but not `load_image()`.

**Note:** Although HoloPy stores metadata even when writing to .tif image files, it is still recommended that holograms be saved in HDF5 format using `save()`. Floating point intensity values are rounded to 8-bit integers when using `save_image()`, resulting in information loss.
1.2.5 Non-Square Pixels

The holograms above make use of several default assumptions. When you load an image like:

```python
raw_holo = hp.load_image(imagepath, spacing = 0.0851)
```
you are making HoloPy assume a square array of evenly spaced grid points. If your pixels are not square, you can provide pixel spacing values in each direction:

```python
raw_holo = hp.load_image(imagepath, spacing = (0.0851, 0.0426))
```

Most displays will default to displaying square pixels but if you use HoloPy’s built-in `show()` function to display the image, your hologram will display with pixels of the correct aspect ratio.

1.3 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.

1.3.1 Example Reconstruction

```python
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:

```python
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:

```python
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()`:

```python
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:

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

### 1.3.2 Visualizing Reconstructions

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

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

Pressing the left and right arrow keys steps through volumes slices - propagation to different z-planes. (Don’t use the down arrow key: it will mess up the stepping due to a peculiarity of Matplotlib. If this happens, close your plot window and show it again. Sorry.). 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:

```python
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:

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

### 1.3.3 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, simply 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:

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

### 1.4 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 reconstructions 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. Currently, only square input images and propagation through media with a refractive index of 1 are supported.

### 1.4.1 Example Reconstruction

```python
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')
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 bsection of code in turn. The first block:

```python
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:

```python
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:

```python
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
```

(continues on next page)
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 \texttt{ps\_propagate()} and a cropped region of the result is dis-
played. See the Reconstructing Data (Numerical Propagation) page for details on visualizing the reconstruction
results.

1.4.2 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 \texttt{mag}, or when calling \texttt{ps\_propagate()} directly the desired pixel spacing in the reconstruction is specified
through the spacing of \texttt{out\_schema}. Note that the output spacing will not be the spacing of \texttt{out\_schema} exactly,
but should be within a few percent of it. We recommend calling \texttt{get\_spacing()} on \texttt{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 \texttt{npix\_out}, the entire reconstruction plane will be returned. However, if the
linear size of total reconstruction plane is larger than \texttt{npix\_out}, only the center region of the reconstruction plane
with linear size \texttt{npix\_out} is returned.

In the current version of the code, the amount of memory needed to perform a reconstruction scales with \texttt{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.

1.5 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.

1.5.1 A Simple Example

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

```python
import holopy as hp
from holopy.scattering import calc_holo, Sphere
```
We'll examine each section of code in turn. The first few lines:

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

sphere = Sphere(n = 1.59, r = .5, center = (4, 4, 5))
medium_index = 1.33
illum_wavelen = 0.660
illum_polarization = (1,0)
detector = hp.detector_grid(shape = 100, spacing = .1)

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

We will be scattering light off a Scatterer object, specifically a Sphere. 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 how we are illuminating our sphere, and how that light will be detected:

```python
medium_index = 1.33
illum_wavelen = 0.66
illum_polarization = (1,0)
```
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.

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

```python
detector = hp.detector_grid(shape = 100, spacing = .1)

holo = calc_holo(detector, sphere, medium_index, illum_wavelen, illum_polarization)
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.

```python
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.

### 1.5.2 More Complex Scatterers

**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).

```python
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)

1.5.3 Customizing Scattering Calculations

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().

```python
illum_dim = {'illumination':['red', 'green']}
n_dict = {'red':1.58, 'green':1.60}
w1_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=w1_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. By default, scattering from single spheres is calculated using Mie theory, which 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.

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 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 you are using multi-layered spheres (HoloPy’s implementation of the multisphere theory can’t currently handle coated spheres). You can specify Mie theory manually when calling the `calc_holo()` function:

```python
from holopy.scattering import Mie
holo = calc_holo(exp_img, collection, theory = Mie)
```

Similarly, HoloPy calculates scattering from cylindrical or spheroidal particles by using T-matrix code from Michael Mishchenko, but these scatterer types are not compatible with Mie 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. It is fairly easy to 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.
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 individual coordinate dimensions:

```python
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.

1.5.4 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.)

```python
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:

```python
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:

```python
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.

1.6 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:

```bash
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:

```bash
make seq
```

or:

```bash
make
```

or:

```bash
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:

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

or:

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

or:

```bash
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.
1.6.1 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:

```python
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`.

1.6.2 Multiple Materials: A Janus Sphere

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

```python
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:

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

Scattering calculations can inform us about the hologram produced by a specific scatterer, but they can’t tell us anything about what type of scatterer produced an experimentally measured hologram. For this inverse scattering problem, we turn to a Bayesian inference approach. We calculate the holograms produced by many similar scatterers, and we evaluate which ones are closest to our measured hologram. We can then use known information about the scatterers to determine the parameters of the scatterer (for example, its size and refractive index) that are most likely to have produced the observed hologram.

In the following example, we will infer the size, refractive index, and position of a spherical scatterer. The approach and formalism is described in more detail in [Dimiduk2016]. For more information on Bayesian inference in general, see [Gregory2005].

Here is the full example. We’ll go through it step-by-step afterward:

```python
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, calc_holo
from holopy.inference import prior, AlphaModel, TemperedStrategy

# 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)

# Set up the prior
s = Sphere(n=prior.Gaussian(1.5, .1), r=prior.BoundedGaussian(.5, .05, 0, np.inf), center=prior.make_center_priors(data_holo))

# Set up the noise model
model = AlphaModel(s, noise_sd=data_holo.noise_sd, alpha=1)
strat = TemperedStrategy()
result = strat.sample(model, data_holo)

fit_vals = result.values()
fit_holo = result.best_fit()
hp.save('example-sampling.h5', result)
```

The first few lines import the code needed to compute holograms and do parameter estimation.

```python
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, calc_holo
from holopy.inference import prior, AlphaModel, TemperedStrategy
```
1.7.1 Preparing Data

Next, we load a hologram from a file using the same steps as those in *Loading Data*

```python
# 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)
```

You will notice that the hologram data is localized to a region near the center of the image. We only want to compare calculated holograms to this region, so we will crop our image with `subimage()`. We also need to normalize the data so that its mean is 1, since calculations return a normalized result. Since our image is background divided, its mean is already very close to 1, but it is good to get in the habit of normalizing anyway.

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

**Note:** It is often useful to test an unfamiliar technique on data for which you know the expected outcome. Instead of actual data, you could use a hologram calculated from `calc_holo()`, and modulated by random noise with `add_noise()`.

1.7.2 Defining a Probability Model

**Priors**

We know that the hologram was produced by a spherical scatterer, so we want to define a `Sphere` object like we did in the *Scattering Calculations* tutorial. However, in this case we don’t know what parameters to specify for the sphere (since that is what we’re trying to find out). Instead, we write down a probabilistic statement of our prior information about the sphere. In statistics, we call this a prior. For the case we are investigating here, we would probably have some best guess and uncertainty about the size and index of the particle, obtained from the supplier or from prior work with the particle. We will guess the radius to be 0.5 micrometers (with 50 nm error) and refractive index to be 1.5 (with 0.1 error). We also need to provide a prior for the position of the sphere. We can use a `hough()` transform to get a pretty good guess of where the particle is in x and y, but it is difficult to determine where it is in z.

**Note:** One trick to get a better estimate of z position is to numerically propagate the hologram backwards in space with `propagate()`, and look for where the interference fringes vanish.

Let’s turn our information about priors into code by defining our scatterer:

```python
s = Sphere(n=prior.Gaussian(1.5, .1), r=prior.BoundedGaussian(.5, .05, 0, np.inf),
           center=prior.make_center_priors(data_holo))
```

We use a Gaussian distribution as the prior because it is the most conservative distribution we can use if all we know is some expected value and some uncertainty about that expected value. For the radius we also know that it must be nonnegative, so we can bound the Gaussian at zero. The `make_center_priors()` function automates generating priors for a sphere center using `center_find()` (based on a hough transform). It assigns Gaussian priors for x and y, and picks a large uniform prior for z to represent our ignorance about how far the particle is from the imaging plane. In this case the center prior will be:
Next we need to define a model that tells HoloPy how probable it is that we would see the data we observed given some hypothetical scatterer position, size and index. In the language of statistics, this is referred to as a likelihood. In order to compute a likelihood, you need some estimate of how noisy your data is (so that you can figure out how likely it is that the differences between your model and data could be explained by noise). Here we use the `noise_sd` attribute of our hologram, which was automatically calculated from the deviation in background images when we ran `bg_correct()`.

```python
model = AlphaModel(s, noise_sd=data_holo.noise_sd, alpha=1)
```

**Note:** \(\alpha\) is a model parameter that scales the scattered beam intensity relative to the reference beam. It is often less than 1 for reasons that are poorly understood. If you aren’t sure what value it should take in your system, you can allow \(\alpha\) to vary by giving it a prior, as we did with the sphere parameters.

### 1.7.3 Sampling the Posterior

Finally, we can sample the posterior probability for this model. Essentially, a set of proposed scatterers are randomly generated according to the priors we specified. Each of these scatterers is then evaluated in terms of how well it matches the experimental hologram `data_holo`. A Monte Carlo algorithm iteratively produces and tests sets of scatterers to find the scatterer parameters that best reproduce the target hologram. We end up with a distribution of values for each parameter (the posterior) that represents our updated knowledge about the scatterer when accounting for the expected experimental hologram. We need to define a `TemperedStrategy` object that sets up the inference calculation, here using default settings. Then, we combine the model and data with the strategy to perform the actual sampling (ignoring any RuntimeWarnings about invalid values):

```python
strat = TemperedStrategy()
result = strat.sample(model, data_holo)
```

The above line of code may take a long time to run (it takes 10-15 mins on our 8-core machines). If you just want to quickly see what the results look like, try:

```python
strat = TemperedStrategy(nwalkers=10, max_pixels=100)
result = strat.sample(model, data_holo, nsamples=100)
```

This code should run very quickly, but its results cannot be trusted for any actual data. Nevertheless, it can give you an idea of what format the results will take. In our last line of code, we have adjusted three parameters to make the code run faster: `nwalkers` describes the number of scatterers produced in each generation. `samples` describes how many generations of scatterers to produce. Together, they define how many scattering calculations must be performed. For the values chosen in the fast code, a Monte Carlo steady state will not yet have been achieved, so the resulting posterior distribution is not very meaningful. `max_pixels` describes the maximum number of pixels compared between the experimental hologram and the test holograms. It turns out that holograms contain a lot of redundant information owing to their symmetry, so a subset of pixels can be analyzed without loss of accuracy. However, 100 pixels is probably too few to capture all of the relevant information in the hologram.

You can get a quick look at our obtained best fit values and the resulting hologram with:
fit_vals = result.values()
fit_holo = result.best_fit()

result.values() gives you the maximum a posteriori probability (MAP) value as well as one-sigma credibility intervals (or you can request any other sigma with an argument to the function). You can also look only at central measures:

result.MAP
result.mean
result.median

Since calculation of useful results takes a long time, you will usually want to save them to an hdf5 file:

hp.save('example-sampling.h5', result)

### 1.7.4 Tying Parameters

You may desire to use **tied priors**, in which several physical quantities that could be varied independently are constrained to have the same (but non-constant) value. A common example involves multi-particle holograms in which all of the particles are constrained to have the same refractive index, but the index is determined by the Bayesian inference method. This may be done by defining a prior and using it in multiple places:

```python
from holopy.scattering import Spheres
n_prior = prior.Gaussian(1.5, .1)
s1 = Sphere(n=n_prior, r=prior.BoundedGaussian(.5, .05, 0, np.inf),
            center=prior.make_center_priors(data_holo))
s2 = Sphere(n=n_prior, r=prior.BoundedGaussian(.5, .05, 0, np.inf),
            center=prior.make_center_priors(data_holo))
spheres = Spheres([s1, s2])
```

And then defining the model using:

```python
model = AlphaModel(spheres, noise_sd=data_holo.noise_sd, alpha=1)
```

### 1.7.5 References

### 1.8 Fitting Models to Data

In addition to Bayesian inference, HoloPy can also do simpler least-squares fits to determine the scatterer parameters that best match an experimentally measured hologram. The main advantage of this technique is that it can be much faster. The drawback is that good initial guesses of each parameter are required to obtain accurate results.

**Note:** The HoloPy fitting methods have been superseded by the Bayesian inference techniques described in the [Bayesian inference of Parameter Values](#) tutorial. We strongly recommend that approach unless you have a good reason that fitting is preferable in your particular situation.
1.8.1 A Simple Fit

We start by loading and processing data just as we did for the parameter inference in the previous tutorial.

```python
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, calc_holo

# 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)
```

Define a Model

The model specification is a little bit different from the inference case. First, we define a parameterized scatterer including initial guesses and absolute bounds using the `Parameter` class. Note that the bounds here are not uncertainty values as in the inference case, but instead represent the full allowed range of a parameter (like the `Uniform` prior). The center coordinates must be specified as \((x, y, z)\), in that order. Here, we will keep particle radius and refractive index fixed. Fitting works best when there are only a few uncertain parameters. You can find guesses for \(x\) and \(y\) coordinates with `center_find()`, and guess \(z\) with `propagate()`. In this image (uncropped version), the particle’s center is near \((24, 22, 15)\), with coordinates in microns.

```python
from holopy.fitting import fit, Model
from holopy.fitting import Parameter as par
par_s = Sphere(center = (par(guess = 24, limit = [15,30]),
                     par(22, [15, 30]), par(15, [10, 20])),
               r = .5, n = 1.58)

Then this parametrized scatterer, along with a desired scattering calculation, is used to define a model:

```python
model = Model(par_s, calc_holo, alpha = par(.6, [.1, 1]))
```

alpha is an additional fitting parameter first introduced in [Lee2007] (see `References and credits` for additional details).

To see how well the guess in your model lines up with the hologram you are fitting to, use:

```python
guess_holo = calc_holo(data_holo, par_s, scaling=model.alpha)
```

Run the Fit

Once you have all of that set up, running the fit is almost trivially simple:

```python
result = fit(model, data_holo)
```
We can see just the fit results with result.scatterer.center. The initial guess of the sphere’s position (24, 22, 15) was corrected by the fitter to (24.17, 21.84, 16.42). Notice that we have achieved sub-pixel position resolution!

From the fit, result.scatterer gives the scatterer that best matches the hologram. result.alpha is the alpha for the best fit. result.chisq and result.rsq are statistical measures of the the goodness of the fit.

You can also compute a hologram of the final fit result to compare to the data with:

```python
result_holo = calc_holo(data_holo, result.scatterer, scaling=result.alpha)
```

Finally, we save the result with:

```python
hp.save('result.h5', result)
```

### 1.8.2 Speeding up Fits with Random Subset Fitting

A hologram usually contains far more information than is needed to determine the number of parameters you are interested in. Because of this, you can often get a significantly faster fit with no little or no loss in accuracy by fitting to only a random fraction of the pixels in a hologram.

```python
result = fit(model, data_holo, random_subset=.01)
```

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 .1 or .01 with little effect on your results and gain a speedup of 10x or greater.

### 1.8.3 Advanced Parameter Specification

#### Complex Index of Refraction

You can specify a complex index with:

```python
from holopy.fitting import ComplexParameter
Sphere(n = ComplexParameter(real = par(1.58, step = 0.01), imag = 1e-4))
```

This will fit to the real part of index of refraction while holding the imaginary part fixed. You can fit to it as well by specifying imag = par(1e-4) instead of imag = 1e-4. In a case like this where we are providing a small imaginary part for numerical stability, you would not want to fit to it. However fitting to an imaginary index component could be useful for a metal particle. Setting the key word argument step = 0.01 specifies the the step size used in calculating the numerical derivatives of this parameter. Specifying a small step size is often necessary when fitting for an index of refraction.

#### Tying Parameters

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:

```python
from holopy.scattering import Spheres
n1 = par(1.59)
sc = Spheres([Sphere(n = n1, r = par(0.5e-6), center = [10., 10., 20.]),
              Sphere(n = n1, r = par(0.5e-6), center = [9., 11., 21.])])
```
1.9 Developer’s Guide

1.9.1 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 build_ext --inplace
```

This puts the extensions inside the source tree, so that you can work directly from /home/me/holopy. You will need to add /home/me/holopy to your python_path for python to find the module when you import it.

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.
1.9.2 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 Speeding up Fits with Random Subset Fitting), 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.

Bayesian inference of Parameter Values 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.

1.9.3 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:

```python
class YourTheory(ScatteringTheory):
    def _raw_fields(self, positions, scatterer, medium_wavevec, medium_index, illum_˓→polarization):
        # Your code here
    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 _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 because we want to provide a faster fields implementation for mie and multisphere (and you might want to for your theory).

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.
1.9.4 Adding a new inference model

To perform inference, you need a noise model. You can make a new noise model by inheriting from \texttt{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: \texttt{\_forward}. This function receives a dictionary of parameter values and a data shape schema (defined by \texttt{detector\_grid()}, for example) and needs to return simulated data of shape specified. See the \texttt{\_forward} function in \texttt{AlphaModel} for an example of how to do this.

If you want to use some other noise model, you may need to override \texttt{\_lnlike} and define the probability given your uncertainty. You can reference \texttt{\_lnlike} in \texttt{NoiseModel}.

1.9.5 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:

\begin{verbatim}
python run_nose.py
\end{verbatim}

1.10 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.

1.10.1 General Image Processing Tools

The tools described here are frequently used when analyzing holograms. They are available from the \texttt{holopy.core.\_process} namespace.

The \texttt{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 \texttt{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 \texttt{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 \texttt{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 \texttt{zero\_filter()}. This function replaces the dead pixel with the average of its neighbours, and fails if adjacent pixels have zero intensity.

The \texttt{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 \texttt{image\_gradient()}. To simply remove a planar intensity gradient from an image, use \texttt{detrend()}. Note that this gives a mean pixel value of zero.

Frequency space analysis provides a powerful tool for working with images. Use \texttt{fft()} and \texttt{ifft()} to perform fourier transforms and inverse fourier transforms, respectively. These make use of \texttt{scipy.fftpack} functions, but are wrapped to correctly interpret HoloPy objects. HoloPy also includes a Hough transform (\texttt{hough()}) to help identify lines and other features in your images.
1.10.2 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()`.

1.11 Concepts

1.11.1 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.

1.11.2 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.

1.11.3 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 $\alpha$, $\beta$, and $\gamma$ (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 $\alpha$ about the HoloPy $z$ axis.
2. Rotate the scatterer an angle $\beta$ about the HoloPy $y$ axis.
3. Rotate the scatterer an angle $\gamma$ 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:

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

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 experimental metadata, reconstructing holograms, calculating light scattering, fitting scattering models to data, and visualizing images and calculations.

2.2 Subpackages

2.2.1 holopy.core package

Subpackages

holopy.core.io package

Module contents

holopy.core.io.io module

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

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, normals=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.
- normals (list-like) – Orientation of detector. 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, normals=None, noise_sd=None, channel=None, name=None)

Load data or results

Parameters

- inf (single or list of basestring or files) – File to load. If the file is a yaml file, all other arguments are ignored. If inf is a list of image files or filenames they are all loaded as a a timeseries hologram
- 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)

Returns obj

Return type The object loaded, holopy.core.marray.Image, or as loaded from yaml

pack_attrs (a, do_spacing=False, scaling=None)

pad_channel (im, color_axis='illumination', padval=0)

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

- `im (ndarray or holopy.image.Image)` - image to save.
- `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
- `scaling ('auto', None, or (None|Int, None|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.

`unpack_attrs (a)`

### holopy.core.process package

**Module contents**

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

### holopy.core.process.centerfinder module

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](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 surrounding 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

**holopy.core.process.fourier module**

Handles Fourier transforms of HoloPy images by using scipy’s fftpack. Tries to correctly interpret dimensions from xarray.
**fft** *(a, overwrite=False, 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 Marray, 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**

- **a** *(ndarray)* – The array to transform
- **overwrite** *(bool)* – Allow this function to overwrite the Marray you pass in. This may improve performance slightly. Default is not to overwrite
- **shift** *(bool)* – Whether to perform an fftshift on the Marray to give low frequencies near the center as you probably expect. Default is to do the fftshift.

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

**Return type** ndarray

**ift_coord** *(c)*

**ift_coords** *(cs)*

**get_spacing** *(c)*

**iff** *(a, overwrite=False, 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 Marray, 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**

- **a** *(ndarray)* – The array to transform
- **overwrite** *(bool)* – Allow this function to overwrite the Marray you pass in. This may improve performance slightly. Default is not to overwrite
- **shift** *(bool)* – Whether to perform an fftshift on the Marray to give low frequencies near the center as you probably expect. Default is to do the fftshift.

**Returns** ifta – The inverse fourier transform of a

**Return type** ndarray

**ift_coord** *(c)*

**ift_coords** *(cs)*

**transform_metadata** *(a, inverse)*

**holopy.core.process.img_proc module**

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** *(ndarray or Image)* – 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** *ndarray*

**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** *(ndarray)* – Image to process

**Returns**

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

**Return type** *ndarray*

**normalize** *(image)*

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

**Parameters**

- **image** *(ndarray)* – The array to normalize

**Returns**

- **normalized_image** – The normalized image

**Return type** *ndarray*

**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** (*numpy.ndarray*) – 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 tuple of ints*) – Desired shape of the region. If a single int is given the region will be that dimension in along every axis. Shape should be even

**Returns** sub – Subset of shape shape centered at center. For marrays, marray.origin will be set such that the upper left corner of the output has coordinates relative to the input.

**Return type** numpy.ndarray or RegularGrid marray object

### 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.

**Parameters** image (*ndarray*) – 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** ndimage

---

**Module contents**

Loading, saving, and basic processing of data.

holopy.core contains code to load images and holopy yamls into marray objects. It also contains the machinery for saving all HoloPy objects as holopy yaml files. Finally, it provides some basic mathematical operations, mostly as higher level wrappers around numpy or scipy routines.

Main use cases are

1. Image or other data file + metadata => Image or other Marray object

2. Raw Image + processing => processed Image object

3. Any HoloPyObject from calculations or processing => achival yaml text or text/binary result

---

**holopy.core.holopy_object module**

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.

like_me (filter_none=True, **kwargs)

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

ordered_dump (dumper, tag, data)

holopy.core.math module

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.

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 compatibility with previous conventions, which were adopted for compatibility with the passive picture used by SCSMFO.

**rsq** (*fit, data*)

Calculate correlation coefficient 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*)

Returns Cartesian coordinates of a point given in spherical polar coordinates.

Parameters **theta, phi** (*r, *) – Spherical polar coordinates of point.

Returns **cartesian_coords** – Dictionary of Cartesian coordinates of point with keys ‘x’, ‘y’, and ‘z’.

Return type dict

2.2. Subpackages
to_spherical \((x, y, z)\)

Return the spherical polar coordinates of a point in Cartesian coordinates.

Parameters

\(y, z (x,)\) – Cartesian coordinates of point

Returns spherical_coords – Dictionary of spherical polar coordinates \((r, \theta, \phi)\) of point with keys ‘r’, ‘theta’, ‘phi’.

Return type \(\text{dict}\)

Notes

\(\theta\) is the polar angle measured from the \(z\) axis with range \((0, \pi)\). \(\phi\) is the azimuthal angle with range \((0, 2\pi)\).

holopy.core.metadata module

Classes for defining metadata about experimental or calculated results.

clean_concat \((	ext{arrays, dim})\)
copy_metadata \((	ext{old, data, do_coords=True})\)
data_grid \((	ext{arr, spacing=None, medium_index=None, illum_wavelen=None, illum_polarization=None, normals=None, noise_sd=None, name=None, extra_dims=None, z=0})\)

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

Returns

Return type \(\text{DataArray object}\)

Notes

Use of higher-level detector_grid() and detector_points() functions is recommended.

default_norms \((\text{coords, n})\)
detector_grid \((\text{shape, spacing, normals=None, name=None, extra_dims=None})\)

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

Parameters

\cdot \text{shape} (\text{int or list-like (2)}) – If int, detector is a square grid of shape \(x\) shape points. If array_like, detector has \(\text{shape[0]}\) rows and \(\text{shape[1]}\) columns.

\cdot \text{spacing} (\text{int or list-like (2)}) – If int, distance between square detector pixels. If array_like, \(\text{spacing[0]}\) between adjacent rows and \(\text{spacing[1]}\) between adjacent columns.

\cdot \text{normals} (\text{list-like or None}) – If list-like, detector orientation.

\cdot \text{name} (\text{string, optional}) –

\cdot \text{extra_dims} (\text{dict, optional}) – extra dimension(s) to add to the empty detector grid as \{\text{dimname:[coords]}\}

Returns grid – DataArray of zeros with coordinates calculated according to \(\text{shape}\) and \(\text{spacing}\)

Return type \(\text{DataArray object}\)
Notes

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

**detector_points**

```python
detector_points(coords={}, x=None, y=None, z=None, r=None, theta=None, phi=None, normals='auto', 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 coordinates of detectors.
- `normals (string, optional)` – Default behavior: normal in +z direction for Cartesian coordinates, -r direction for polar coordinates. Non-default behavior not currently implemented.
- `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.)
HoloPy Documentation, Release 3.2.1

update_metadata(a, medium_index=None, illum_wavenlen=None, illum_polarization=None, normals=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_wavenlen** (*float*) – Updated wavelength of illumination in the image.
- **illum_polarization** (*list-like*) – Updated polarization of illumination in the image.
- **normals** (*list-like*) – Updated detector orientation of the image.
- **noise_sd** (*float*) – standard deviation of Gaussian noise in the image.

Returns **b** – copy of input image with updated metadata. The ‘normals’ field is not allowed to be empty.

Return type *xarray.DataArray*

holopy.core.utils module

Misc utility functions to make coding more convenient

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):

- **d** (*dict*) – The dictionary to operate on
- **keys** (*list(string]*) – The keys to exclude

returns (*dict*) – A copy of dict without any of the specified keys

ensure_array(x)

ensure_listlike(x)

is_none(o)

Check if something is None.

This can’t be done with a simple is check anymore because numpy decided that array is None should do an element wise comparison.

Parameters **o** (*object*) – Anything you want to see if is None

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

Analogous 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
2.2.2 holopy.fitting package

Module contents

Fit models of scattering to data

Make precision measurements of a scattering system by fitting a model of it to data

The fitting module is used to:

1. Define Scattering Model -> Model object
2. Fit model to data -> FitResult object
3. Fit model to timeseries -> list of FitResult objects

holopy.fitting.fit module

Routines for fitting a hologram to an exact solution

class FitResult (parameters, scatterer, chisq, rsq, converged, time, model, minimizer, minimization_details)

Bases: holopy.core.holopy_object.HoloPyObject

The results of a fit.

You should not make objects of this class directly, they will be given to you by `fit()`

Parameters

- **parameters** *(array (float))* – The fitted values for each parameter
- **scatterer** *(Scatterer)* – The best fit scatterer
- **chisq** *(float)* – The $\chi^2$ goodness of fit
- **rsq** *(float)* – The $R^2$ goodness of fit
- **converged** *(bool)* – Did the minimizer converge
- **time** *(float)* – Time in seconds the fit took
- **minimizer** *(Minimizer)* – The minimizer used in the fit
- **minimization_details** *(object)* – Additional information returned by the minimizer about the minimization

alpha

fitted_holo *(schema)*

classmethod from_summary *(summary, scatterer_cls)*

Build a FitResult from a summary.

The returned FitResult will be incomplete because summaries do not contain all of the minimizer information that full results do

Parameters summary *(dict)* – A dict as from cls.summary containing information about a fit.

next_model ()

Construct a model to fit the next frame in a time series

niter ()
summary()
    Put just the essential components of a fit result in a dictionary

    summary_misc = ['rsq', 'chisq', 'time', 'converged', 'niter']

fit (model, data, minimizer=<class 'holopy.fitting.minimizer.Nmpfit'>, random_subset=None)
    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 (Marray object) – The data to fit
    • minimizer ((optional) Minimizer) – The minimizer to use to do the fit
    • random_subset (float (optional)) – Fit only a randomly selected fraction of the
data points in data

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

Return type FitResult

make_subset_data (data, random_subset=None, pixels=None, return_selection=False)

holopy.fitting.minimizer module

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

class Minimizer
    Bases: holopy.core.holopy_object.HoloPyObject

    Common interface to all minimizers holopy supports

    minimize (parameters, cost_func)
        Find the best solution to an optimization problem

Parameters
    • parameters (list of Parameter objects) – Parameters to vary in the model
    • cost_func (function) – A function taking parameters as arguments that returns the
residual for the minimization problem

pars_from_minimizer (parameters, values)

class Nmpfit (quiet=False, ftol=1e-10, xtol=1e-10, gtol=1e-10, damp=0, maxiter=100)
    Bases: holopy.fitting.minimizer.Minimizer

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

Parameters
    • quiet (Boolean) – If True, suppress output on minimizer convergence.
    • 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 co-
sine 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.

**minimize** *(parameters, cost_func, debug=False)*

Find the best solution to an optimization problem

**Parameters**

• **parameters** *(list of Parameter objects)* – Parameters to vary in the model

• **cost_func** *(function)* – A function taking parameters as arguments that returns the residual for the minimization problem

### holopy.fitting.model module

Classes for defining models of scattering for fitting

**class BaseModel** *(scatterer, medium_index=None, illum_wavelen=None, illum_polarization=None, theory='auto')*

Bases: holopy.core.holopy_object.HoloPyObject

**get_par**(name, pars, schema=None, default=None)

**get_pars**(names, pars, schema=None)

**par**(name, schema=None, default=None)

**parameters**

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

Bases: holopy.fitting.model.BaseModel

Representation of a model to fit to data

**Parameters**

• **alpha** *(float or Parameter)* – Extra scaling parameter, hopefully this will be removed by improvements in our theory soon.

• **constraints** *(function or list of functions)* – One or a list of constraint functions. A constraint function should take a scatterer as an argument and return False if you wish to disallow that scatterer (usually because it is un-physical for some reason)

**get_alpha**(pars=None)

**guess**

**guess_dict**

**residual**(pars, data)

---

2.2. Subpackages
class ParameterizedObject(obj)
    Bases: holopy.fitting.model.Parametrization
    Specify parameters for a fit by including them in an object
    Parameters are named automatically from their position in the object
        Parameters obj(scatterer) – Object containing parameters specifying any values vary in the
          fit. It can also include numbers for any fixed values

    guess
    make_from(parameters)

class Parametrization(make_scatterer, parameters)
    Bases: holopy.core.holopy_object.HoloPyObject
    Description of free parameters and how to make a scatterer from them
    Parameters
        • make_scatterer(function) – A function which should take the Parametrization pa-
          rameters by name as keyword arguments and return a scatterer
        • parameters(list) – The list of parameters for this Parametrization

    guess
    make_from(parameters)

limit_overlaps(fraction=0.1)
    Generator for constraint prohibiting overlaps beyond a certain tolerance
    Parameters fraction(float) – Fraction of the sphere diameter that the spheres should be al-
        lowed to overlap by
    Returns constraint – A function which tests scatterers to see if the exceed the specified tolerance
    Return type function (scatterer -> bool)

tied_name(name1, name2)

holopy.fitting.parameter module

Classes for describing free parameters in fitting models

class ComplexParameter(real, imag, name=None)
    Bases: holopy.fitting.parameter.Parameter
    A complex free parameter
    ComplexParameters have 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 name’s for the parameters; ComplexParameter will name them
        • name(string) – Short descriptive name of the ComplexParameter. Do not provide this
          if using a ParameterizedScatterer, a name will be assigned based its position within the
          scatterer.

    guess
class Parameter(guess=None, limit=None, name=None, **kwargs)
Bases: holopy.core.holopy_object.HoloPyObject

Describe a free parameter in a fitting model

Parameters

- **guess** *(optional) float –* Your initial guess for this parameter
- **limit** *(optional) float or (float, float) –* Describe how the minimizer should allow a parameter to vary. A single value here fixes the parameter to that value, a pair limits the parameter to vary between (min, max)
- **name** *(optional) string –* A short descriptive name of the parameter. Do not provide this if using the parameter in a ParameterizedScatterer, it will assign a name based on the Parameter’s position within the scatterer
- ****kwargs *(varies) –* Additional information made available to the minimizer. This can be used for things like step sizes.

fixed

scale *(physical)*
Scales parameters to approximately unity

Parameters physical *(np.array(dtype=float)) –*
Returns scaled
Return type np.array(dtype=float)

unscale *(scaled)*
Inverts scale’s transformation

Parameters scaled *(np.array(dtype=float)) –*
Returns physical
Return type np.array(dtype=float)

2.2.3 holopy.inference package

Module contents

holopy.inference.noise_model module

class AlphaModel(scatterer, noise_sd=None, alpha=1, medium_index=None, illum_wavelen=None, illum_polarization=None, theory='auto')
Bases: holopy.inference.noise_model.NoiseModel

class NoiseModel(scatterer, noise_sd, medium_index=None, illum_wavelen=None, illum_polarization=None, theory='auto')
Bases: holopy.fitting.model.BaseModel

Model probabilities of observing data

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

lnlike *(par_vals, data)*

lnposterior *(par_vals, data)*

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

Bases: holopy.inference.prior.Gaussian

lnprob(p)
sample(size=None)

class Gaussian(mu, sd, name=None)

Bases: holopy.inference.prior.Prior

guess
lnprob(p)
prob(p)
sample(size=None)

class Prior(guess=None, limit=None, name=None, **kwargs)

Bases: holopy.fitting.parameter.Parameter

guess
interval
lnprob(p)
sample(size=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 this is 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
holopy.inference.result module

Results of sampling

class SamplingResult (dataset, model, strategy)
    Bases: holopy.core.holopy_object.HoloPyObject

    MAP
    best_fit()
    data
    lnprobs
    mean
    median
    output_scatterer()
    samples
    sigma_intervals (sigmas=[-2, -1, 1, 2])
    values (sigma_interval=1)

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

    dataset
    model

class UncertainValue (value, plus, minus=None, n_sigma=1, kind='MAP')
    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

    autocorr_from_sentinal (d)
    autocorr_to_sentinal (d)
    get_stage_names (inf)

holopy.inference.sample module

Sample posterior probabilities given model and data

class EmceeStrategy (nwalkers=100, pixels=2000, threads='auto', cleanup_threads=True, seed=None)
    Bases: holopy.core.holopy_object.HoloPyObject

    make_guess (parameters)
    sample (model, data, nsamples=1000, walker_initial_pos=None)
class TemperedStrategy(next_initial_dist=<function sample_one_sigma_gaussian>, nwalkers=100, min_pixels=50, max_pixels=1000, threads='auto', stages=3, stage_len=30, seed=None)

Bases: holopy.inference.sample.EmceeStrategy

sample (model, data, nsamples=1000)

autothreads (threads='auto', quiet=False)

d钷ce_lnprobs_DataArray (sampler)

d钷ce_samples_DataArray (sampler, parameters)

get_acor (sampler)

sample_emcee (model, data, nwalkers, nsamples, walker_initial_pos, threads='auto', cleanup_threads=True, seed=None)

sample_emcee_autocorr (model, data, nwalkers, independent_samples, walker_initial_pos, estimated_autocorr, threads='auto')

sample_one sigma gaussian (result)

tempered_sample (model, data, nwalkers=100, min_pixels=50, max_pixels=2000, samples=600, next_initial_dist=<function sample_one_sigma_gaussian>, stages=3, stage_len=30, seed=None, threads='auto')

2.2.4 holopy.propagation package

Module contents

holopy.propagation.convolution_propagation module

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 (Image or VectorGrid) – Hologram to propagate

• d (float or list of floats) – Distance to propagate, in meters, 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**cfsp. 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

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

Return type Image or Volume

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.

---

\(^1\) Kreis, Handbook of Holographic Interferometry (Wiley, 2005), equation 3.79 (page 116)

\(^2\) Kreis, Optical Engineering 41(8):1829, section 5
Parameters

- **shape** ((int, int)) – maximum dimensions of the transfer function
- **spacing** ((float, float)) – the spacing between points is the grid to calculate
- **wavelen** (float) – the wavelength in the medium you are propagating through
- **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
- **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 *np.ndarray*

References

**holopy.propagation.point_source_propagate module**

**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. 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.


data is a holopy Xarray. It is the hologram to reconstruct. Must be square. The pixel spacing must also be square. d = distance from pinhole to reconstructed image, in meters (this is z in Jericho and Kreuzer). Can be a scalar or a 1D list or array. L = distance from screen to pinhole, in meters beam_c = [x,y] coordinates of beam center, in pixels out_schema = size of output image and pixel spacing, default is the schema of data.

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

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


data is a holopy Xarray. It is the hologram to reconstruct. Must be square. The pixel spacing must also be square. d = distance from pinhole to reconstructed image, in meters (this is z in Jericho and Kreuzer). Can be a scalar. L = distance from screen to pinhole, in meters beam_c = [x,y] coordinates of beam center, in pixels out_schema = size of output image and pixel spacing, default is the schema of data. if Ip == True, returns Ip to

---

2.2. Subpackages
be used on calculations in the stack if \( \text{Ip} == \text{False} \) compute reconstructed image as normal if \( \text{Ip} \) is an image, use this to speed up calculations
returns an image(volume) corresponding to the reconstruction at plane(s) \( d \).

### 2.2.5 holoPy.scattering package

**Subpackages**

**holoPy.scattering.scatterer package**

**Module contents**

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

**holoPy.scattering.scatterer.bisphere module**

Defines cylinder scatterers.

**class Bisphere**

\[
\text{class Bisphere}(n=None, h=None, d=None, center=None, rotation=(0, 0, 0))
\]

Based on: holoPy.scattering.scatterer.scatterer.CenteredScatterer

Scattering object representing bisphere scatterers

**Parameters**

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

**holoPy.scattering.scatterer.capsule module**

Defines capsule scatterers.

**class Capsule**

\[
\text{class Capsule}(n=None, h=None, d=None, center=None, rotation=(0, 0, 0))
\]

Based on: 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

### holopy.scattering.scatterer.composite module

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* – List of scatterers that make up this object

### Notes

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

**add**(scatterer)

**from_parameters**(parameters, update=False)

**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)

**parameters**

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

**select**(keys)

**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

---

2.2. Subpackages
holopy.scattering.scatterer.csg module

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)

holopy.scattering.scatterer.cylinder module

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)–
holopy.scattering.scatterer.ellipsoid module

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

Holopy documentation, Release 3.2.1

2.2. Subpackages

holopy.scattering.scatterer.janus module

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

holopy.scattering.scatterer.scatterer module

The abstract base class for all scattering objects

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

from_parameters(parameters, update=False)

Create a Scatterer from a dictionary of parameters

Parameters parameters (dict or list) – 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

**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

**select** *(keys)*

**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)*

**guess** ()

**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**

**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)
checkguess (par)

find_bounds (indicator)

    Finds the bounds needed to contain an indicator function

Notes

    Will probably determine incorrect bounds for functions which are not convex

holopy.scattering.scatterer.sphere module

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
        list of complex – Index of each each layer

t
        list of float – Thickness of each layer

center
    length 3 listlike – specifies coordinates of center of sphere

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
        complex or list of complex – index of refraction of each layer of the sphere

r
    float or list of float – radius of the sphere or outer radius of each sphere.

center
    length 3 listlike – specifies coordinates of center of sphere

guess ()

indicators
like_me(**overrides)
num_domains
rotated(alpha, beta, gamma)

**holopy.scattering.scatterer.spherecluster module**

Defines Spheres, a Scatterers scatterer consisting of Spheres

class Spheres(scatterers, warn=True)
    Bases: holopy.scattering.scatterer.composite.Scatterers
    
    Contains optical and geometrical properties of a cluster of spheres.
    
    spheres
        list of Spheres – Spheres which will make up the cluster

**Notes**

add(scatterer)
center
centers
guess()
largest_overlap()
n
n_imag
n_real
overlaps
r

**holopy.scattering.scatterer.spheroid module**

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
        complex – Index of refraction
    
    r
        (float, float) – length of xy and z semi-axes of the spheroid
rotation
  3-tuple, list or numpy array – specifies the Euler angles (alpha, beta, gamma) in radians

center
  3-tuple, list or numpy array – specifies coordinates of center of the scatterer

indicators

holopy.scattering.theory package

Subpackages

holopy.scattering.theory.mie_f package

Module contents

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

holopy.scattering.theory.mie_f.mie_specfuncs module

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:

\texttt{qratio} \((z1, z2, nstop, dns1=None, dns2=None, eps1=0.001, eps2=1e-16)\)
Calculate ratio of Riccati-Bessel functions defined in \cite{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.

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

Notes

See \cite{Mackowski1990} eqns. 65-66. Uses Lentz continued fraction algorithm for logarithmic derivatives.

\texttt{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)
• \( \text{nmx} \) (order from which downward recursion begins.)
• \( \text{nstop} \) (integer, maximum order)

Notes

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

\texttt{log\_der\_13}(z, \text{nstop}, \text{eps1}=0.001, \text{eps2}=1e-16)

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


Parameters

• \( z \) (complex number)
• \( \text{nstop} \) (maximum order of computation)
• \( \text{eps1} \) (underflow criterion for Lentz continued fraction for \( D_n \))
• \( \text{eps2} \) (convergence criterion for Lentz continued fraction for \( D_n \))

\texttt{riccati\_psi\_xi}(x, \text{nstop})

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

Parameters

• \( x \) (float) – Argument
• \( \text{nstop} \) (int) – Maximum order to calculate to

Returns \( \psi \) and \( \xi \)

Return type ndarray(2, nstop)

Notes

Uses upwards recursion.

\texttt{holopy.scattering.theory.mie\_f.miescatlib module}

MieScatLib.py

Library of code to do Mie scattering calculations.

\texttt{asymmetry\_parameter}(al, bl)

Calculate asymmetry parameter of scattered field.

Parameters \( \text{bn} \) (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 $(a_l, b_l)$
Calculates scattering and extinction cross sections given arrays of Mie scattering coefficients an and bn.

Parameters
- $b_n (a_n)$ – 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 $(m, x, n_{max}, \text{eps1}=0.001, \text{eps2}=1e-16)$
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 $(m, x, n_{stop}, \text{eps1}=0.001, \text{eps2}=1e-16)$
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 ($k_{med} * a$)
- $n_{stop}$ (int) – Maximum order of scattered field expansion
- $\text{eps1}$ (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].

**holopy.scattering.theory.mie_f.multilayer_sphere_lib module**

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.


**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)

**Module contents**

Theories to compute scattering from objects.

All theories have a common interface defined by `holopy.scattering.theory.scatteringtheory.ScatteringTheory`. 
holopy.scattering.theory.dda module

Compute holograms using the discrete dipole approximation (DDA). Currently uses ADDA (http://code.google.com/p/a-dda/) to do DDA calculations. .. moduleauthor:: Thomas G. Dimiduk <tdimiduk@physics.harvard.edu>

class DDA(n_cpu=1, max_dpl_size=None, use_indicators=False, keep_raw_calculations=False, addacmd=[])

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 .. attribute:: n_cpu

int (optional) – Number of threads to use for the DDA calculation

max_dpl_size

float (optional) – 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.

use_indicators

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

keep_raw_calculations

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

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.

required_spacing (medium_wavelen, medium_index, n)

holopy.scattering.theory.mie module

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.
holopy.scattering.theory.multisphere module

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.

```python
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**

*integer (optional)* – maximum number of iterations to use in solving the interaction equations

**meth**

*integer (optional)* – method to use to solve interaction equations. Set to 0 for biconjugate gradient; 1 for order-of-scattering

**eps**

*float (optional)* – relative error tolerance in solution for interaction equations

**qeps1**

*float (optional)* – error tolerance used to determine at what order the single-sphere spherical harmonic expansion should be truncated

**qeps2**

*float (optional)* – error tolerance used to determine at what order the cluster spherical harmonic expansion should be truncated

**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 because overlapping spheres can be useful for getting fits to converge. The results to be sensile for small overlaps even though mathematically speaking they are not strictly 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, independently of the above scfodim.for parameters.

References

normalize_polarization(illum_polarization)

holopy.scattering.theory.scatteringtheory module

Base class for scattering theories. Implements python-based calc_intensity and calc_holo, based on subclass's calc_field. .. moduleauthor:: Jerome Fung <jerome.fung@post.harvard.edu> .. moduleauthor:: Vinothan N. Manoharan <vnm@seas.harvard.edu> .. moduleauthor:: Thomas G. Dimiduk <tdimiduk@physics.harvard.edu>

class ScatteringTheory
  Bases: holopy.core.holopy_object.HoloPyObject

  Defines common interface for all scattering theories. .. rubric:: Notes

  A subclasses that do the work of computing scattering should do it by implementing _raw_fields and/or _raw_scat_mats and (optionally) _raw_cross_sections. _raw_cross Sections is needed only for calc_cross_sections. Either of _raw_fields or _raw_scat_mats will give you calc_holo, calc_field, and calc_intensity. Obviously calc_scat_matrix will only work if you implement _raw_cross_sections. So the simplest thing is to just implement _raw_scat_mats. You only need to do _raw_fields there is a way to compute it more efficiently and you care about that speed, or if it is easier and you don’t care about matrices.

stack_spherical(a)

wavevec(a)

holopy.scattering.theory.tmatrix module

Compute holograms using Mishchenko’s T-matrix method for axisymmetric scatterers. Currently uses
class Tmatrix(delete=True)
  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.

  delete

    bool (optional) – If True (default), delete the temporary directory where we store the input and output file for the fortran executable

Notes

Does not handle near fields. This introduces ~5% error at 10 microns.
Module contents

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 Schema object
3. Calculate scattering quantities with an theory appropriate for your scatterer -> Marray object

holopy.scattering.calculations module

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 theta>.

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_intensity 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 (schema, scatterer, medium_index=None, illum_wavelen=None, illum_polarization=None, theory='auto')

Calculate hologram formed by interference between scattered fields and a reference wave

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_intensity 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(schema, 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**

- **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
- **scaling** (scaling value (alpha) for amplitude of reference wave) –

**Returns** `holo` – Calculated hologram from the given distribution of spheres

**Return type** `Image` object

calc_intensity(schema, scatterer, medium_index=None, illum_wavelen=None, illum_polarization=None, theory='auto')

Calculate intensity at a location or set of locations

**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_intensity will error out and ask you to specify a theory

**Returns** `inten` – scattered intensity

**Return type** `Image`

calc_scat_matrix(schema, scatterer, medium_index=None, illum_wavelen=None, theory='auto')

Compute farfield scattering matrices for scatterer

**Parameters**

- **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_intensity will error out and ask you to specify a theory.

Returns scat_matr – Scattering matrices at specified positions

Return type Marray

determine_theory (scatterer)

finalize (schema, result)

interpret_theory (scatterer, theory='auto')

prep_schema (schema, medium_index, illum_wavelen, illum_polarization)

scattered_field_to_hologram (scat, ref, normals)
Calculate a hologram from an E-field

Parameters

• scat (VectorGrid) – The scattered (object) field
• ref (xarray[vector]) – The reference field
• detector_normal ((float, float, float)) – Vector normal to the detector the hologram should be measured at (defaults to z hat, a detector in the x, y plane)

holopy.scattering.geometry module

Routines for common calculations and transformations of groups of spheres.

This code is in need of significant refactoring and simplification, refactoring which may break code that depends on it.

angles (cluster, degrees=True)

calculate the angles between one particle and every pair of other particles

Parameters

• cluster (holopy.scattering.scatterer.Scatterer) – A sphere cluster to determine the interparticle distances of.
• degrees (bool) – Whether to return angles in degrees (True) or in radians (False).

Notes

Angle abc is the acute angle formed by edges connecting points ab and bc. If a, b, and c are locations of particles (vertices), the returned 3D array has non-zero values for angles abc, zeros for angles aba, and NaN’s for “angles” aab.

distances (cluster, gaponly=False)

calculate the distances between each sphere in a cluster and each of the others

Parameters

• cluster (holopy.scattering.scatterer.Scatterer) – A sphere cluster to determine the interparticle distances of.
• gaponly (bool) – Whether to calculate the distances between particle centers or between particle surfaces (gap distances).
Notes

The returned array of distances includes redundant information. The identical distances between sphere 1 and sphere 2 and between sphere 2 and sphere 1 are both in the returned array. Calculating and returning the full array makes it easy for the user to access all the interparticle distances starting from any sphere of interest.

**make_cubecluster** *(index, radius, gap, xcom=0, ycom=0, zcom=0)*

Returns a sphere cluster of eight particles forming a cube centered on a given center of mass.

**Parameters**
- **index** – Index of refraction of particles.
- **radius** – Radius if particles.
- **gap** – Space to add between the particles.
- **xcom** – Center of mass x-coordinate
- **ycom** – Center of mass y-coordinate
- **zcom** – Center of mass z-coordinate

**make_octacluster** *(index, radius, gap, xcom=0, ycom=0, zcom=0)*

Returns a sphere cluster of six particles forming an octahedron centered on a given center of mass.

**Parameters**
- **index** – Index of refraction of particles.
- **radius** – Radius if particles.
- **gap** – Space to add between the particles.
- **xcom** – Center of mass x-coordinate
- **ycom** – Center of mass y-coordinate
- **zcom** – Center of mass z-coordinate

**make_polytetracluster** *(index, radius, gap, xcom=0, ycom=0, zcom=0)*

Returns a sphere cluster of six particles forming a polytetrahedron centered on a given center of mass of the middle tetrahedron.

**Parameters**
- **index** – Index of refraction of particles.
- **radius** – Radius if particles.
- **gap** – Space to add between the particles.
- **xcom** – Center of mass x-coordinate
- **ycom** – Center of mass y-coordinate
- **zcom** – Center of mass z-coordinate

**make_sqcluster** *(index, radius, gap, xcom=0, ycom=0, zcom=0)*

Returns a sphere cluster of four particles forming a square centered on a given center of mass.

**Parameters**
- **index** – Index of refraction of particles.
- **radius** – Radius if particles.
- **gap** – Space to add between the particles.
• \texttt{xcom} – Center of mass x-coordinate
• \texttt{ycom} – Center of mass y-coordinate
• \texttt{zcom} – Center of mass z-coordinate

\texttt{make_tetracluster}(\texttt{index}, \texttt{radius}, \texttt{gap}, \texttt{xcom}=0, \texttt{ycom}=0, \texttt{zcom}=0)
Returns a sphere cluster of four particles forming a tetrahedron centered on a given center of mass.

Parameters
• \texttt{index} – Index of refraction of particles.
• \texttt{radius} – Radius if particles.
• \texttt{gap} – Space to add between the particles.
• \texttt{xcom} – Center of mass x-coordinate
• \texttt{ycom} – Center of mass y-coordinate
• \texttt{zcom} – Center of mass z-coordinate

\texttt{make_tribipyrccluster}(\texttt{index}, \texttt{radius}, \texttt{gap}, \texttt{xcom}=0, \texttt{ycom}=0, \texttt{zcom}=0)
Returns a sphere cluster of five particles forming a triagonal bipyramid centered on a given center of mass.

Parameters
• \texttt{index} – Index of refraction of particles.
• \texttt{radius} – Radius if particles.
• \texttt{gap} – Space to add between the particles.
• \texttt{xcom} – Center of mass x-coordinate
• \texttt{ycom} – Center of mass y-coordinate
• \texttt{zcom} – Center of mass z-coordinate

\texttt{make_tricluster}(\texttt{index}, \texttt{radius}, \texttt{gap}, \texttt{xcom}=0, \texttt{ycom}=0, \texttt{zcom}=0)
Returns a sphere cluster of three particles forming an equilateral triangle centered on a given center of mass.

Parameters
• \texttt{index} – Index of refraction of particles.
• \texttt{radius} – Radius if particles.
• \texttt{gap} – Space to add between the particles.
• \texttt{xcom} – Center of mass x-coordinate
• \texttt{ycom} – Center of mass y-coordinate
• \texttt{zcom} – Center of mass z-coordinate

\subsection{2.2.6 \texttt{holopy.vis} package}

\textbf{Module contents}

Visualize HoloPy objects

Uses Matplotlib and Mayavi to visualize holopy objects.

1. \texttt{Image}, \texttt{Volume}, or \texttt{Spheres} object => plot or rendering
This module does not import plotting packages until they are actually needed so that holopy does not have a hard dependency on mayavi or matplotlib. Because of this you may see a small lag on your first plot.

holopy.vis.show module

A general show method that can display most holopy and scatterpy objects in a sensible way.

exception VisualizationNotImplemented (o)

    Bases: Exception

show (o, color=(0.5, 0.5, 0.5))

    Visualize a scatterer, hologram, or reconstruction

    Parameters o (Image, Volume or Spheres) – Object to visualize

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)

test_disp ()

holopy.vis.vis2d module

New custom display functions for holograms and reconstructions.

class plotter (im, plane_axes, slice_axis, starting_index, color_axis)

    Bases: object

    click (event)

    draw ()

    location (x, y)

    pixel (x, y)

show2d (im, plane_axes=None, slice_axis=None, starting_index=0, color_axis=None, phase=False)

    Display a hologram or reconstruction

    Allows scrolling through multidimensional holograms or reconstructions. Defaults to showing magnitude of complex images

    Parameters

        • im (ndarray) – Image to be shown
        • z0 (int) – slice along z dimension to show first.
        • t (int) – slice along time to show for reconstructions.

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
holopy.vis.vis3d module

Show sphere clusters using mayavi

import_mayavi()

show_scatterer(scatterer, spacing=None)

show_sphere_cluster(s, color)

volume_contour(d)
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.

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.


Python Module Index

holopy, 33
holopy.core, 39
holopy.core.holopy_object, 39
holopy.core.io, 33
holopy.core.math, 40
holopy.core.metadata, 42
holopy.core.process, 35
holopy.core.process.centerfinder, 35
holopy.core.process.fourier, 36
holopy.core.process.img_proc, 37
holopy.core.utils, 44
holopy.fitting, 45
holopy.fitting.fit, 45
holopy.fitting.minimizer, 46
holopy.fitting.model, 47
holopy.fitting.parameter, 48
holopy.inference, 49
holopy.inference.noise_model, 49
holopy.inference.prior, 50
holopy.inference.result, 51
holopy.inference.sample, 51
holopy.propagation, 52
holopy.propagation.convolution_propagation, 52
holopy.propagation.point_source_propagation, 53
holopy.scattering, 68
holopy.scattering.calculations, 68
holopy.scattering.geometry, 70
holopy.scattering.scatterer, 54
holopy.scattering.scatterer.bisphere, 54
holopy.scattering.scatterer.capsule, 54
holopy.scattering.scatterer.composite, 55
holopy.scattering.scatterer.csg, 56
holopy.scattering.scatterer.cylinder, 56
holopy.scattering.scatterer.ellipsoid, 57
holopy.scattering.scatterer.janus, 57
holopy.scattering.scatterer.scatterer, 57
holopy.scattering.scatterer.sphere, 59
holopy.scattering.scatterer.spherecluster, 60
holopy.scattering.scatterer.spheroid, 60
holopy.scattering.theory, 64
holopy.scattering.theory.dda, 65
holopy.scattering.theory.mie, 65
holopy.scattering.theory.mie_f, 61
holopy.scattering.theory.mie_f.mie_specfuncs, 61
holopy.scattering.theory.mie_f.miescatlib, 62
holopy.scattering.theory.mie_f.multilayer_sphere_lib, 64
holopy.scattering.theory.multisphere, 66
holopy.scattering.theory.scatteringtheory, 67
holopy.scattering.theory.tmatrix, 67
holopy.vis, 72
holopy.vis.show, 73
holopy.vis.vis2d, 73
holopy.vis.vis3d, 74
A
add() (Scatterers method), 55
add() (Spheres method), 60
add_noise() (in module holopy.core.process.img_proc), 37
all_numbers() (in module holopy.scattering.scatterer.ellipsoid), 57
alpha (FitResult attribute), 45
AlphaModel (class in holopy.inference.noise_model), 49
angles() (in module holopy.scattering.geometry), 70
asymmetry_parameter() (in module holopy.scattering.theory.mie_f.miescatlib), 62
autocorr_from_sentinal() (in module holopy.inference.result), 51
autocorr_to_sentinal() (in module holopy.inference.result), 51
autothreads() (in module holopy.inference.sample), 52

calc_cross_sections() (in module holopy.scattering.calculations), 68
calc_field() (in module holopy.scattering.calculations), 68
calc_holo() (in module holopy.scattering.calculations), 69
calc_intensity() (in module holopy.scattering.calculations), 69
calc_scat_matrix() (in module holopy.scattering.calculations), 69
Capsule (class in holopy.scattering.scatterer.capsule), 54
cartesian_distance() (in module holopy.core.math), 40
center (LayeredSphere attribute), 59
center (Sphere attribute), 59
center (Spheres attribute), 60
center (Spheroid attribute), 61
center_find() (in module holopy.core.process.centerfinder), 35
CenteredScatterer (class in holopy.scattering.scatterer.scatterer), 57
centers (Spheres attribute), 60
checkguess() (in module holopy.scattering.scatterer.scatterer), 59
chisq() (in module holopy.core.math), 40
clean_concat() (in module holopy.core.metadata), 42
click() (plotter method), 73
ComplexParameter (class in holopy.fitting.parameter), 48
cross_sections() (in module holopy.scattering.theory.mie_f.miescatlib), 63
CsgScatterer (class in holopy.scattering.scatterer.csg), 56
Cylinder (class in holopy.scattering.scatterer.cylinder), 56

data (SamplingResult attribute), 51
data_grid() (in module holopy.core.metadata), 42
dataset (TemperedSamplingResult attribute), 51
DDA (class in holopy.scattering.theory.dda), 65
default_extension() (in module holopy.core.io(io), 33
default_norms() (in module holopy.core.metadata), 42
delete (Tmatrix attribute), 67
detector_grid() (in module holopy.core.metadata), 42
detector_points() (in module holopy.core.metadata), 43
determine_theory() (in module holopy.scattering.calculations), 70
detrend() (in module holopy.core.process.img_proc), 38
dict_to_array() (in module holopy.core.metadata), 43
dict_without() (in module holopy.core.utils), 44
Difference (class in holopy.scattering.scatterer.csg), 56
distances() (in module holopy.scattering.geometry), 70
draw() (plotter method), 73

E
Ellipsoid (class in holopy.scattering.scatterer.ellipsoid), 57
emcee_lnprobs_DataArray() (in module holopy.inference.sample), 52
emcee_samples_DataArray() (in module holopy.inference.sample), 52
EmceeStrategy (class in holopy.inference.sample), 51
ensure_array() (in module holopy.core.utils), 44
ensure_listlike() (in module holopy.core.utils), 44
eps (Multisphere attribute), 66

F
fft() (in module holopy.core.process.fourier), 36
finalize() (in module holopy.scattering.calculations), 70
find_bounds() (in module holopy.scattering.scatterer.scatterer), 59
fit() (in module holopy.fitting.fit), 46
FitResult (class in holopy.fitting.fit), 45
fitted_holo() (FitResult method), 45
fixed (Parameter attribute), 49
flat() (in module holopy.core.metadata), 43
from_flat() (in module holopy.core.metadata), 44
from_parameters() (CenteredScatterer method), 57
from_parameters() (Scatterers method), 55
from_summary() (holopy.fitting.fit.FitResult class method), 45
from_yaml() (holopy.core.holopy_object.HoloPyObject class method), 40
ft_coord() (in module holopy.core.process.fourier), 37
ft_coords() (in module holopy.core.process.fourier), 37

G
Gaussian (class in holopy.inference.prior), 50
get_acor() (in module holopy.inference.sample), 52
get_alpha() (Model method), 47
get_component_list() (Scatterers method), 55
get_example_data() (in module holopy.core.io.io), 33
get_example_data_path() (in module holopy.core.io.io), 33
get_extents() (in module holopy.core.metadata), 43
get_par() (BaseModel method), 47
get_pars() (BaseModel method), 47
guess() (Scatterer method), 58
guess() (Spheres method), 60
guess_dict (Model attribute), 47

H
holopy (module), 33
holopy.core (module), 39
holopy.core.holopy_object (module), 39
holopy.core.io (module), 33
holopy.core.io.io (module), 33
holopy.core.math (module), 40
holopy.core.metadata (module), 42
holopy.core.process (module), 35
holopy.core.process.centerfinder (module), 35
holopy.core.process.fourier (module), 36
holopy.core.process.img_proc (module), 37
holopy.core.utils (module), 44
holopy.fitting (module), 45
holopy.fitting.fit (module), 45
holopy.fitting.minimizer (module), 46
holopy.fitting.model (module), 47
holopy.fitting.parameter (module), 48
holopy.inference (module), 49
holopy.inference.noise_model (module), 49
holopy.inference.prior (module), 50
holopy.inference.result (module), 51
holopy.inference.sample (module), 51
holopy.propagation (module), 52
holopy.propagation.convolution_propagation (module), 52
holopy.propagation.point_source_propagate (module), 53
holopy.scattering (module), 68
holopy.scattering.calculations (module), 68
holopy.scattering.geometry (module), 70
holopy.scattering.scatterer (module), 54
holopy.scattering.scatterer.bisphere (module), 54
holopy.scattering.scatterer.capsule (module), 54
holopy.scattering.scatterer.composite (module), 55
holopy.scattering.scatterer.csg (module), 56
holopy.scattering.scatterer.cylinder (module), 56
holopy.scattering.scatterer.ellipsoid (module), 57
holopy.scattering.scatterer.janus (module), 57
holopy.scattering.scatterer.scatterer (module), 57
holopy.scattering.scatterer.sphere (module), 59
make_tetracluster() (in module holopy.scattering.geometry), 72
make_tribipyrccluster() (in module holopy.scattering.geometry), 72
make_tricluster() (in module holopy.scattering.geometry), 72
MAP (SamplingResult attribute), 51
max_dpl_size (DDA attribute), 65
mean (SamplingResult attribute), 51
median (SamplingResult attribute), 51
meth (Multisphere attribute), 66
Mie (class in holopy.scattering.theory.mie), 65
minimize() (Minimizer method), 46
minimize() (Nmpfit method), 47
Minimizer (class in holopy.fitting.minimizer), 46
mkdir_p() (in module holopy.core.utils), 44
Model (class in holopy.fitting.model), 47
model (TemperedSamplingResult attribute), 51
Multisphere (class in holopy.scattering.theory.multisphere), 66

n (LayeredSphere attribute), 59
n (Sphere attribute), 59
n (Spheres attribute), 60
n (Spheroid attribute), 60
n_imag (Spheres attribute), 60
n_real (Spheres attribute), 60
next_model() (FitResult method), 45
niter (Multisphere attribute), 66
niter() (FitResult method), 45
Nmpfit (class in holopy.fitting.minimizer), 46
NoiseModel (class in holopy.inference.noise_model), 49
normalize() (in module holopy.core.process.img_proc), 38
normalize_polarization() (in module holopy.scattering.theory.multisphere), 67
nstop() (in module holopy.scattering.theory.mie_f.miescatlib), 63
num_domains (Scatterer attribute), 58
num_domains (Sphere attribute), 60

ordered_dump() (in module holopy.core.holopy_object), 40
output_scatterer() (SamplingResult method), 51
overlaps (Spheres attribute), 60

pack_attrs() (in module holopy.core.io.io), 34
pad_channel() (in module holopy.core.io.io), 34
par() (BaseModel method), 47
Parameter (class in holopy.fitting.parameter), 48
ParameterizedObject (class in holopy.fitting.model), 47
parameters (BaseModel attribute), 47
parameters (CenteredScatterer attribute), 58
parameters (Scatters attribute), 55
Parametrization (class in holopy.fitting.model), 48
pars_from_minimizer() (Minimizer method), 46
pixel() (plotter method), 73
plotter (class in holopy.vis.vis2d), 73
prep_schema() (in module holopy.scattering.calculations), 70
primdim() (in module holopy.core.metadata), 43
Prior (class in holopy.inference.prior), 50
prob() (Gaussian method), 50
propagate() (in module holopy.propagation.convolution_propagation), 52
ps_propagate() (in module holopy.propagation.point_source_propagate), 53
ps_propagate_plane() (in module holopy.propagation.point_source_propagate), 53

qeps1 (Multisphere attribute), 66
qeps2 (Multisphere attribute), 66
Qratio() (in module holopy.scattering.theory.mie_f.mie_specfuncs), 61

r (LayeredSphere attribute), 59
r (Sphere attribute), 59
r (Spheres attribute), 60
r (Spheroid attribute), 60
R_psi() (in module holopy.scattering.theory.mie_f.mie_specfuncs), 61
repeat_sing_dims() (in module holopy.core.utils), 44
required_spacing() (DDA method), 65
residual() (Model method), 47
riccati_psi_xi() (in module holopy.scattering.theory.mie_f.mie_specfuncs), 62
rotate_points() (in module holopy.core.math), 40
rotated() (CsgScatterer method), 56
rotated() (Scatters method), 55
rotated() (Sphere method), 60
rotation (Spheroid attribute), 60
rotation_matrix() (in module holopy.core.math), 41
rsq() (in module holopy.core.math), 41

sample() (BoundedGaussian method), 50
sample() (EmceeStrategy method), 51
sample() (Gaussian method), 50
sample() (TemperedStrategy method), 52
HoloPy Documentation, Release 3.2.1

Index

sample() (Uniform method), 50
sample_emcee() (in module holopy.inference.sample), 52
sample_emcee_autocorr() (in module holopy.inference.sample), 52
sample_one_sigma_gaussian() (in module holopy.inference.sample), 52
samples (SamplingResult attribute), 51
SamplingResult (class in holopy.inference.result), 51
save() (in module holopy.core.io.io), 34
save_image() (in module holopy.core.io.io), 35
scale() (Parameter method), 49
scatcoeffs() (in module holopy.scattering.theory.mie_f.miescatlib), 63
scatcoeffs-multi() (in module holopy.scattering.theory.mie_f.multilayer_sphere_lib), 64
scattered_field_to_hologram() (in module holopy.scattering.calculations), 70
Scatterer (class in holopy.scattering.scatterer.scatterer), 58
Scatterers (class in holopy.scattering.scatterer.composite), 55
scatterers (Scatterers attribute), 55
ScatteringTheory (class in holopy.scattering.theory.scatteringtheory), 67
select() (CenteredScatterer method), 58
select() (Scatterers method), 55
Serializable (class in holopy.core.holopy_object), 40
SerializableMetaClass (class in holopy.core.holopy_object), 40
show() (in module holopy.vis.show), 73
show2d() (in module holopy.vis.vis2d), 73
show_scatterer() (in module holopy.vis.vis3d), 74
show_scatterer_slices() (in module holopy.vis.vis2d), 73
show_sphere_cluster() (in module holopy.vis.vis3d), 74
sigma_intervals() (SamplingResult method), 51
simulate_noise() (in module holopy.core.process.img_proc), 38
Sphere (class in holopy.scattering.scatterer.sphere), 59
sphere_coords() (in module holopy.core.metadata), 43
Spheres (class in holopy.scattering.scatterer.spherecluster), 60
spheres (Spheres attribute), 60
Spheroid (class in holopy.scattering.scatterer.spheroid), 60
stack_spherical() (in module holopy.scattering.theory.scatteringtheory), 67
subimage() (in module holopy.core.process.img_proc), 39
summary() (FitResult method), 45
summary_misc (FitResult attribute), 46
Temp (LayeredSphere attribute), 59
tempered_sample() (in module holopy.inference.sample), 52
TemperedSamplingResult (class in holopy.inference.result), 51
TemperedStrategy (class in holopy.inference.sample), 51
test_disp() (in module holopy.vis.show), 73
tied_name() (in module holopy.fitting.model), 48
Tmatrix (class in holopy.scattering.theory.tmatrix), 67
to_cartesian() (in module holopy.core.math), 41
to_spherical() (in module holopy.core.math), 42
to_vector() (in module holopy.core澹data), 43
to_yaml() (holopy.core.holopy_object.HoloPyObject class method), 40
to_yaml() (holopy.core.holopy_object.Serializable class method), 40
transform() (in module holopy.propagation.convolution_propagation), 52
transform_metadata() (in module holopy.core.process.fourier), 37
translated() (Scatterer method), 58
translated() (Scatterers method), 55
UncertainValue (class in holopy.inference.result), 51
Uniform (class in holopy.inference.prior), 50
Union (class in holopy.scattering.scatterer.csg), 56
unpack_attrs() (in module holopy.core.io.io), 35
unscale() (Parameter method), 49
update_metadata() (in module holopy.core.metadata), 43
updated() (in module holopy.core.dtypes), 44
updated() (in module holopy.inference.prior), 50
use_indicators (DDA attribute), 65
values() (SamplingResult method), 51
VisualizationNotImplemented, 73
volume_contour() (in module holopy.vis.vis3d), 74
voxelate() (Scatterer method), 58
voxelate_domains() (Scatterer method), 59
W
wavevec() (in module holopy.scattering.theory.scatteringtheory), 67
X
x (Scatterer attribute), 59
x (Spheres attribute), 60
y (Scatterer attribute), 59
Y
y (Spheres attribute), 60

Z

z (Scatterer attribute), 59
z (Spheres attribute), 60
zero_filter() (in module holopy.core.process.img_proc), 39