HoloPy Documentation

Release 3.0.0

Manoharan Lab, Harvard University

May 15, 2017

Contents

1 User Guide **3**
- 1.1 Getting Started . . . 3
- 1.2 Loading Data . . . 4
- 1.3 Reconstructing Data (Numerical Propagation) . . . 7
- 1.4 Reconstructing Point Source Holograms (Numerical Propagation) . . . 8
- 1.5 Scattering Calculations . . . 10
- 1.6 Scattering from Arbitrary Structures with DDA . . . 15
- 1.7 Bayesian inference of Parameter Values . . . 16
- 1.8 Fitting Models to Data . . . 20
- 1.9 Developer's Guide . . . 22
- 1.10 HoloPy Tools . . . 23
- 1.11 Concepts . . . 24

2 holopy package **27**
- 2.1 Subpackages . . . 27
- 2.2 Module contents . . . 30

3 References and credits **31**

Bibliography **33**

Release 3.0.0

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

- **Backward propagation of light from a digital hologram to *reconstruct* 3D volumes.**
  - This approach requires no prior knowledge about the scatterer
- **Forward propagation of light from a :ref: *scattering calculation <calc_tutorial>* of a predetermined scatterer.**
  - Comparison to a measured hologram with :ref: *Bayesian inference <infer_tutorial>* 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.

CHAPTER 1

User Guide

Skip to the tutorials if you already have HoloPy installed and want to get started quickly.

Getting Started

Installation

need conda use conda-forge channel to get holopy import holopy

If this line works, skip to usage before diving into the tutorials. .. _dependencies:

Dependencies

TODO: list all dependencies??

Optional dependencies for certain calculations:

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

Windows Support

HoloPy is not currently supported on Windows due to Fortran compiler issues. If this is an area you know something about, we welcome any assistance in getting things working. In the mean time, your best option is probably to run a linux virtual machine on your Windows computer.

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. Try running:

```
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:

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

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

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

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.

Loading and viewing a hologram

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

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

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

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

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

The spacing argument tells holopy about the scale of your image. Here, we had previously measured that each pixel is a square with side length 0.0851 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 usage for instructions.

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.

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

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

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

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

If you are worried about stray light in your optical train, you should also capture a dark-field image of your sample, recorded with no laser illumination.

```
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)
```

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:

```
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 holo, as in `holo.medium_index`. However, you must use a special function `update_metadata()` to edit them. If we forgot to specify metadata when loading the image, we can use `update_metadata()` to add it later:

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

Note: Spacing and wavelength must 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.

Saving and Reloading Holograms

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

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

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

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

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

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

Additional options of `save_image()` allow you to control how image intensity is scaled. 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.

Non-Square Pixels

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

```
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:

```
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.

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.

Example Reconstruction

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

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

zstack = np.linspace(1, 15, 8)
rec_vol = hp.propagate(holo, zstack)
hp.show(rec_vol)
```

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

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

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

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

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

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

```
zstack = np.linspace(1, 15, 8)
rec_vol = hp.propagate(holo, zstack)
```

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

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

Visualizing Reconstructions

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

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

Pressing the left and right arrow keys steps through volumes slices - propagation to different z-planes. (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 usage for instructions.

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

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

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

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

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` variable into `propagate()` indicating how many times the hologram should be iteratively propagated. For example, to propagate in three steps over each distance, we write:

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

Reconstructing Point Source Holograms (Numerical Propagation)

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 than propagation calculations where a collimated light source (plane wave) is used. For recontructions using a plane wave see *Reconstructing Data (Numerical Propagation)*.

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

Example Reconstruction

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

imagepath = get_example_data_path('ps_image01.jpg')
bgpath = get_example_data_path('ps_bg01.jpg')
L = 0.0407 # distance from light source to screen
cam_spacing = 12e-6 # linear size of camera pixels
mag = 1.0 # magnification
npix_out = 1020 # linear size of output image (pixels)
zstack = np.arange(1.08e-3, 1.18e-3, 0.01e-3) # distances from camera to reconstruct

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

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

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

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

loads the relevant modules. The second block:

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

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

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

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

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

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

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 arbitraily. In order to magnify the reconstruction the spacing in the reconstruction plane should be smaller than spacing in the original hologram. In the code above, the magnification of the reconstruction can be set using the variable `mag`, or when calling `ps_propagate()` directly the desired pixel spacing in the reconstruction is specified through the spacing of `out_schema`. Note that the output spacing will not be the spacing of `out_schema` exactly, but should be within a few percent of it. We recommend calling `get_spacing(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 nunber of pixels, and reconstructions with small output spacings will have a large number of pixels. If the linear size (in pixels) of the total reconstruction plane is smaller than `npix_out`, the entire recontruction plane will be returned. However, if the linear size of total reconstruction plane is larger than `npix_out`, only the center region of the reconstruction plane with linear size `npix_out` is returned.

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

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.

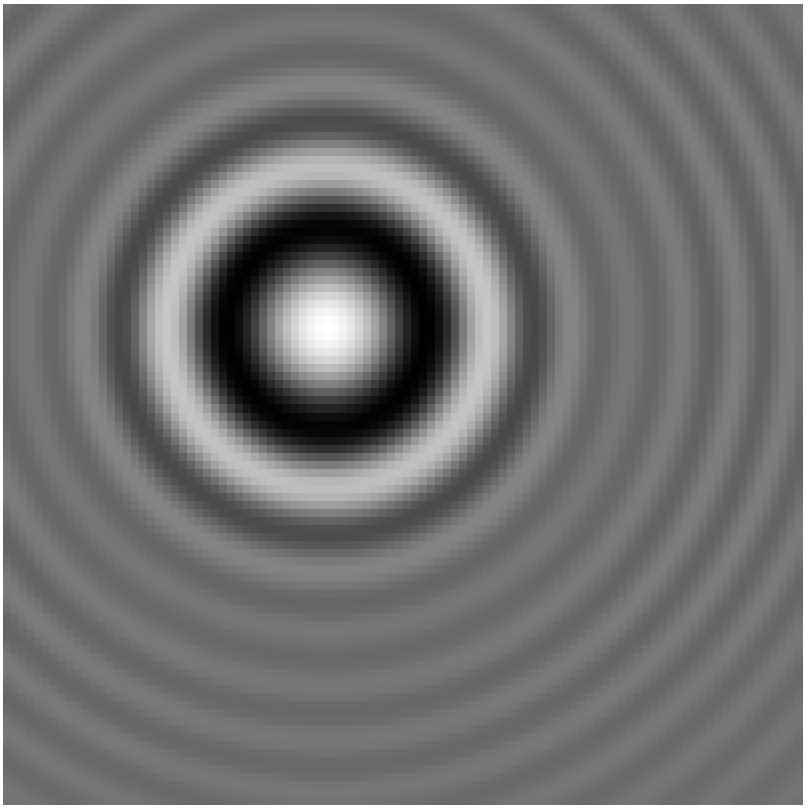
A Simple Example

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

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

sphere = Sphere(n = 1.59, r = .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)
```



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

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

load the relevant modules from HoloPy that we'll need for doing our calculation. The next lines describes the scatterer we would like to model:

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

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:

```
medium_index = 1.33
illum_wavelen = 0.66
```

```
illum_polarization = (1,0)
detector = hp.detector_grid(shape = 100, spacing = .1)
```

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

The scattered light will be collected at a detector, which is frequently a digital camera mounted onto a microscope. Our detector is defined 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 paramaters affect the absolute size of the detector.

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

```
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 sysetm we would like to model, we can use that as an argument in `calc_holo()` instead of our `detector` object created from `detector_grid()`. Holopy will calculate a hologram image with pixels at the same positions as the experimental image, and so we don't need to worry about making a `detector_grid()` with the correct `shape` and `spacing` arguments.

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

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

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

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 radii corresponding to the layers (starting from the core and working outwards).

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

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

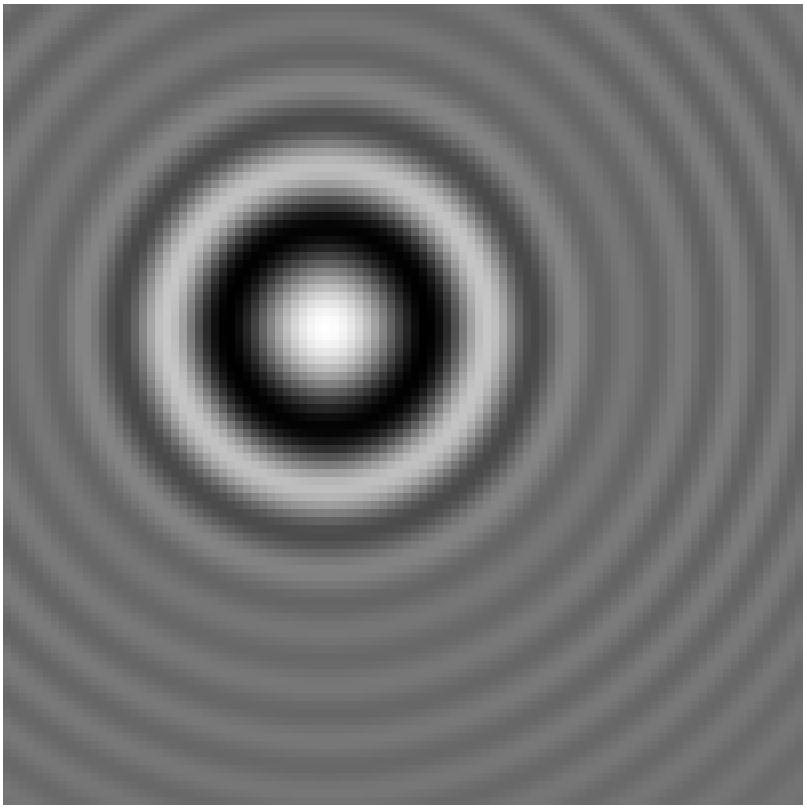
..testcode:

```
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.

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.

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, which gives 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:

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

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 scat_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 indvidual coordinate dimensions:

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

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

Static light scattering calculations

Scattering Matrices

In a static light scattering measurement you record the scattered intensity at a number of angles. In this kind of experiment you are usually not interested in the exact distance of the detector from the particles, and so it's most convenient to work with scattering matrices.

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

detector = hp.detector_points(theta = np.linspace(0, np.pi, 100), phi = 0)
```

```
distant_sphere = Sphere(r=0.5, n=1.59)
matr = calc_scat_matrix(detector, distant_sphere, medium_index, illum_wavelen)
```

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

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

Scattering Cross-Sections

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

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

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

Scattering from Arbitrary Structures with DDA

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

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

```
adda
```

at a terminal for HoloPy DDA calculations to succeed.

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.

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.

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

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

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

Mutiple Materials: A Janus Sphere

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

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

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

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

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

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 reverse problem, we turn to a Bayesian inference approach. We can calculate the holograms produced by many similar scatterers, and evaluate which ones are closest to our measured hologram. We can then use known information about the scatterers to determine which exact scatterer parameters were most likely to have produced the observed hologram.

In this example, we will infer the size, refractive index, and position of a spherical scatterer:

```
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, tempered_sample

# 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
noise_sd = data_holo.std()
model = AlphaModel(s, noise_sd=noise_sd, alpha=1)

result = tempered_sample(model, data_holo)

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

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

```
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, tempered_sample
```

Preparing Data

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

```
# 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.

```
# 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()`.

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, you would probably have some best guess and uncertainty about the size and index of your particle, obtained from the supplier or from prior work with the particle. We will guess radius to be 0.5 microns (with 50 nm error) and refractive index to be 1.5 (with 0.1 error). We also need to provided 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 difficul 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:

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

The Gaussian distribution is the prior used to describe a value for which all we know is some expected value and some uncertainty on 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_finder()` (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:

```
[Gaussian(mu=11.4215, sd=0.0851),
Gaussian(mu=9.0945, sd=0.0851),
Uniform(lower_bound=0, upper_bound=170.2)]
```

Likelihood

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 standard deviation of the data, which is an overestimate of the true noise, since it also includes variaion due to our signal.

```
noise_sd = data_holo.std()
model = AlphaModel(s, noise_sd=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 like the sphere parameters.

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. To do the actual sampling, we use `tempered_sample()` (ignoring any RuntimeWarnings about invalid values):

result = tempered_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 results look like, try:

```
result = tempered_sample(model, data_holo, nwalkers=10, samples=100, max_pixels=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 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 scatterering calculations must be performed. For the values chosen inthe 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 holgoram and the test holograms. It turns out that holograms contain a lot of redundant information (e.g. radial 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 values with:

```
..  testcode::
```

result.values()

result.values() gives you the maximum a posteriori probability (MAP) value as well as 1 sigma (or you can request any other sigma with an argument to the function) credibility intervals. 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:

```
..  testcode::
```

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

References

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 intial 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 .infer_tutorial tutorial. We strongly recommend that approach unless you have a good reason that fitting is preferable in your particular situation.

A Simple Fit

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

```
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 `par()` function. 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, and 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_finder()`, and guess z with `propagate()`. In this image (uncropped version), the particle's is near (24, 22, 15), with coordinates in microns.

```
from holopy.fitting import par, fit, Model
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:

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

`alpha` is an additional fitting parameter first introduced by Lee et al. 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 :

```
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:

```
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:

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

Finally, we save the result with:

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

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.

```
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.

Advanced Parameter Specification

Complex Index of Refraction

You can specify a complex index with:

```
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 :

```
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.])])
```

Developer's Guide

How HoloPy Stores Data

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

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

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

-TODO: how inference results are saved

Adding a new scattering theory

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

```
class YourTheory(ScatteringTheory):
  def _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 matricies you will need to implement _raw_scat_matrs, and if you want cross sections, you will need to implement _raw_cross_sections. We seperate 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.

Adding a new inference model

TODO by Tom Also need to refer to this somewhere in the inference tutorial.

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.

General Image Processing Tools

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

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

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

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

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

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

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

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

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()`.

If you want want to convert between spherical and cartesian coordinates, use `to_cartesian()` and `to_spherical()`.

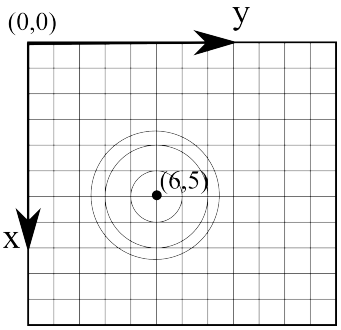
Concepts

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.

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.

Rotations of Scatterers

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

The Euler rotations are performed in the following way:

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

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

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

CHAPTER 2

holopy package

Subpackages

holopy.core package

Subpackages

holopy.core.io package

Submodules

holopy.core.io.io module

holopy.core.io.serialize module

Module contents

holopy.core.process package

Submodules

holopy.core.process.centerfinder module

holopy.core.process.fourier module

holopy.core.process.img_proc module

Module contents

Submodules

holopy.core.errors module

holopy.core.holopy_object module

holopy.core.math module

holopy.core.metadata module

holopy.core.utils module

Module contents

holopy.fitting package

Subpackages

Submodules

holopy.fitting.errors module

holopy.fitting.fit module

holopy.fitting.minimizer module

holopy.fitting.model module

holopy.fitting.parameter module

Module contents

holopy.inference package

Submodules

holopy.inference.noise_model module

holopy.inference.prior module

holopy.inference.result module

holopy.inference.sample module

Module contents

holopy.propagation package

Subpackages

Submodules

holopy.propagation.convolution_propagation module

Module contents

holopy.scattering package

Subpackages

holopy.scattering.scatterer package

Submodules

holopy.scattering.scatterer.bisphere module

holopy.scattering.scatterer.capsule module

holopy.scattering.scatterer.composite module

holopy.scattering.scatterer.csg module

holopy.scattering.scatterer.cylinder module

holopy.scattering.scatterer.ellipsoid module

holopy.scattering.scatterer.janus module

holopy.scattering.scatterer.scatterer module

holopy.scattering.scatterer.sphere module

holopy.scattering.scatterer.sphere_builtin module

holopy.scattering.scatterer.spherecluster module

Module contents

holopy.scattering.theory package

Subpackages

Submodules

holopy.scattering.theory.dda module

holopy.scattering.theory.mie module

holopy.scattering.theory.multisphere module

holopy.scattering.theory.scatteringtheory module

Module contents

Submodules

holopy.scattering.calculations module

holopy.scattering.errors module

holopy.scattering.geometry module

Module contents

holopy.vis package

Submodules

holopy.vis.show module

holopy.vis.vis2d module

holopy.vis.vis3d module

Module contents

Module contents

CHAPTER 3

References and credits

Please see the following references:

If you use HoloPy, we ask that you cite the articles above that are relevant to your application.

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 No. CBET-0747625 and performed in the Manoharan Lab at Harvard University

Bibliography

[Dimiduk2016] Dimiduk, T. G., Manoharan, V. N. (2016) Bayesian approach to analyzing holograms of colloidal particles. Optics Express

[Gregory2005] Gregory, P. (2005) Bayesian Logical Data Analysis. Cambridge University Press

[Fung2012] Fung, J., Perry, R. W., Dimiduk, T. G., & Manoharan, V. N. (2012). Imaging multiple colloidal particles by fitting electromagnetic scattering solutions to digital holograms. Journal of Quantitative Spectroscopy and Radiative Transfer, (0). doi:10.1016/j.jqsrt.2012.06.007

[Perry2012] Perry, R. W., Meng, G., Dimiduk, T. G., Fung, J., & Manoharan, V. N. (2012). Real-space studies of the structure and dynamics of self-assembled colloidal clusters. Faraday Discussions. doi:10.1039/c2fd20061a

[Fung2011] J. Fung *et al.*, "Measuring translational, rotational, and vibrational dynamics in colloids with digital holographic microscopy," *Optics Express* **19**, 8051-8065, (2011).

[Lee2007] S. H. Lee *et al.*, "Characterizing and tracking single colloidal particles with video holographic microscopy," *Optics Express* **15**, 18275-18282, (2007).

[Lentz1976] W. J. Lentz, "Generating Bessel functions in Mie scattering calculations using continued fractions," *Applied Optics* **15**, 668-671, (1976).

[Mackowski1996] D. W. Mackowski and M. I. Mishchenko, "Calculation of the T matrix and the scattering matrix for ensembles of spheres," *J. Opt. Soc. Am. A.* **13**, 2266-2278, (1996).

[Wiscombe1996] W. J. Wiscombe, "Mie Scattering Calculations: Advances in Technique and Fast, Vector-Speed Computer Codes," *NCAR Technical Report*, http://diogenes.iwt.uni-bremen.de/vt/laser/codes/NCARMieReport-revised%20August%201996.pdf

[Yang2003] W. Yang, "Improved recursive algorithm for light scattering by a multilayered sphere," *Applied Optics* **42**, 1710-1720, (2003).

[Yurkin2011] M. A. Yurkin and A. G. Hoekstra, "The discrete-dipole-approximation code ADDA: Capabilities and known limitations," *J. Quant. Spectrosc. Radiat. Transfer* **112**, 2234-2247 (2011).

[Bohren1983] C. F. Bohren and D. R. Huffman, *Absorption and Scattering of Light by Small Particles*, Wiley (1983).