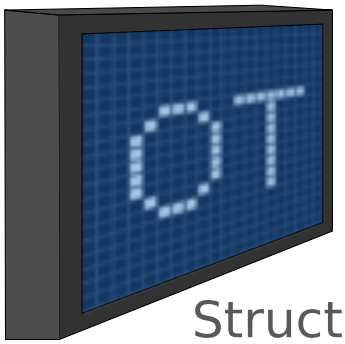

SLM

Toolbox for
Structured Light Methods

OTSLM Documentation

Isaac Lenton

Mar 14, 2020

Contents

CHAPTER 1

Introduction

OTSLM is a set of Matlab functions and graphical user interface for generating patterns for phase and amplitude spatial light modulators (SLMs) such as the digital micromirror device (DMD) and liquid crystal type device. The focus of this toolbox is on patterns for optical tweezers systems but the same functions can probably be used in other applications where amplitude or phase control of light is required.

In the initial release we include functions our group currently uses or is interested in using, but we hope that others will also contribute codes for patterns they use in research publications. If you would like to contribute patterns, we would love to hear from you, see the *Contributing* section.

This documentation provides an overview of the toolbox functions and classes, including examples, typical output, and function/class reference pages which can be used to extend the toolbox for your own needs. The documentation is split into three parts: a *Getting Started* section, *Examples* and *Packages* reference section. The examples section contains additional details about specific tasks the toolbox can be used for. Additional example code is provided as part of the toolbox in the `examples` directory. The packages section contains information about each of the packages. This includes function/class reference pages and example output. Most toolbox functions/classes are documented in the source code, and can be viewed by typing `help <function-name>` at the Matlab prompt. The documentation includes the rendered function/class help and additional content such as examples and typical output.

The toolbox is a work in progress. It is likely, at least in the early versions, the functions will move around, change names and behaviour. Some functions still lack documentation and might be a bit unstable. Comments and suggestions welcome.

To get started using the toolbox, take a look at the *Getting Started* section.

1.1 License

If you publish work using this toolbox, please cite it as

> I. C. D. Lenton, A. B. Stilgoe, T. A. Nieminen, H. Rubinsztein-Dunlop, "OTSLM toolbox for structured light methods", Computer Physics Communications, 2019.

This version of the code is licensed under the GNU GPLv3. Parts of the toolbox incorporate third party open source code, see the documentation, `thirdparty` folder and code for details about licensing of these parts. Further details

can be found in LICENSE.md. If you would like to use the toolbox for something not covered by the license, please contact us.

1.2 Contributing

If you would like to contribute a feature, report a bug or request we add something to the toolbox, the easiest way is by creating a new issue on the OTSLM GitHub page.

If you have code you would like to submit, fork the repository, add the code and open a new issue. This method is preferable to pasting the code in the issue or sending it to us via email since your contribution details will remain attached to the commit you send (tracking authorship).

1.3 Contact us

The best person to contact for inquiries about the toolbox or licensing is Isaac Lenton

CHAPTER 2

Getting Started

This page will guide you through getting started with OTSLM. This page is split into three sections: *installation*, *using the GUIs*, and *writing functions with the toolbox*.

2.1 Installation

To run OTSLM you need to download the toolbox files and have a recent version of Matlab installed (we tested OTSLM with Matlab 2018a). There are a couple of ways to get OTSLM. You can download one of the Matlab toolbox files (with the `.mltbx` extension), you can download a `.zip` archive containing the source code, or you can clone the GitHub repository. The advantage of cloning the GitHub repository is you can easily switch between different versions of the toolbox or download the most recent changes/improvements to the toolbox. There are a range of online tutorials for getting started with git and GitHub, for example https://product.hubspot.com/blog/git-and-github-tutorial-for-beginners.

2.1.1 Using a `.mltbx` file

You can download the latest stable release of OTSLM from either the GitHub release page or the Mathworks File Exchange (you can also access the File Exchange from within Matlab via the **Home > Add-ons > Get Add-ons** button). Simply download the appropriate `.mltbx` file for the relevant version. Once downloaded, execute the file and follow the instructions to install the toolbox.

To change/remove the toolbox, go to **Home > Add-ons > Manage Add-ons** and select the toolbox you would like to configure.

2.1.2 Using a `.zip` or cloning the repository

The latest version of OTSLM can be downloaded from the OTSLM GitHub page. Simply click the *Clone or Download* button and select your preferred method of download. If you are cloning the repository you can checkout different tags to select the desired release. Alternatively, for a specific version, navigate to the release page and select the `.zip` file for the desired release.

To install OTSLM, download the latest version of the toolbox to your computer, if you downloaded a `.zip` file, extract the files to your computer.

Once downloaded, most of the toolbox functionality is ready to use. To start exploring the functionality of the toolbox immediately, you can run the `examples` or launch one of the GUIs in the `+otslm/+ui` directory. However, for writing your own code, you will probably want to add the toolbox to the Matlab path. To do this, simply run

```
addpath('/path/to/toolbox/otslm');
```

Replace the path with the path you placed the downloaded toolbox in. The folder must contain the `+otslm` directory and the `docs` directory. If you downloaded the latest toolbox from GitHub, the final part of the pathname will either be the repository path (if you used `git clone`) or something like `otslm-master` (if you downloaded a ZIP). The above line can be added to the start of each of your files or for a more permanent solution you can add it to the Matlab startup script.

2.1.3 Post installation

To check that `otslm` was found, run the following command and verify it displays the contents of the `+otslm/Contents.m` file

```
help otslm
```

If you have multiple versions of `otslm` downloaded, you may want to check which version is currently being used. The following command can be used to check which toolbox is being used

```
what otslm
```

OTSLM is implemented as a Matlab package, all the core functionality is contained within the `+otslm` directory and can be accessed by adding the folder containing `+otslm` to the path and prefixing the contained functions with `otslm.`. For example, to access the linear function in the `simple` sub-package, you would use

```
im = otslm.simple.linear([10, 10], 3);
```

Some functionality requires additional components. You can choose to install these now or later.

- Optical Tweezers Toolbox (1.5.1 or newer)
- Python (2.7 or newer)
  - numpy (tested on 1.13.3)
  - theano (tested on 0.9)
  - scipy (tested on 1.0)
  - pyfftw (optional, for Fourier transform)
- Red Tweezers
- Specific Matlab toolboxes:
  - Optimization Toolbox
  - Signal Processing Toolbox
  - Neural Network Toolbox
  - Symbolic Math Toolbox
  - Image Processing Toolbox
  - Instrument Control Toolbox

  - Parallel Computing Toolbox
  - Image Acquisition Toolbox
- Matlab MEX compatible C++ compiler

In some cases it is possible to re-write functions to avoid using specific Matlab toolboxes. If you encounter difficultly using a function because of a missing Matlab toolbox, let us know and we may be able to help.

2.2 Exploring the toolbox with the GUI

The toolbox includes a graphical user interface (GUI) for many of the core functions. The user interface allows you to explore the functionality of the toolbox without writing a single line of code. The GUIs can be accessed by running the OTSLM Launcher application. The launcher can be found in the **Apps** menu (if OTSLM was installed using a `.mltbx` file), or run from the file explorer by navigating to the `+otslm/+ui` directory and running `Launcher.mlapp`. If you have already added OTSLM to the path, you can also start the launcher by running the following command in the command window

```
otslm.ui.Launcher
```

If everything is installed correctly, the launcher should appear, as depicted in Fig. 2.1. The window is split into 4 sections: a description of the toolbox, a list of GUI categories, a list of applications, and a description about the selected application. Once you select an application, click Launch.

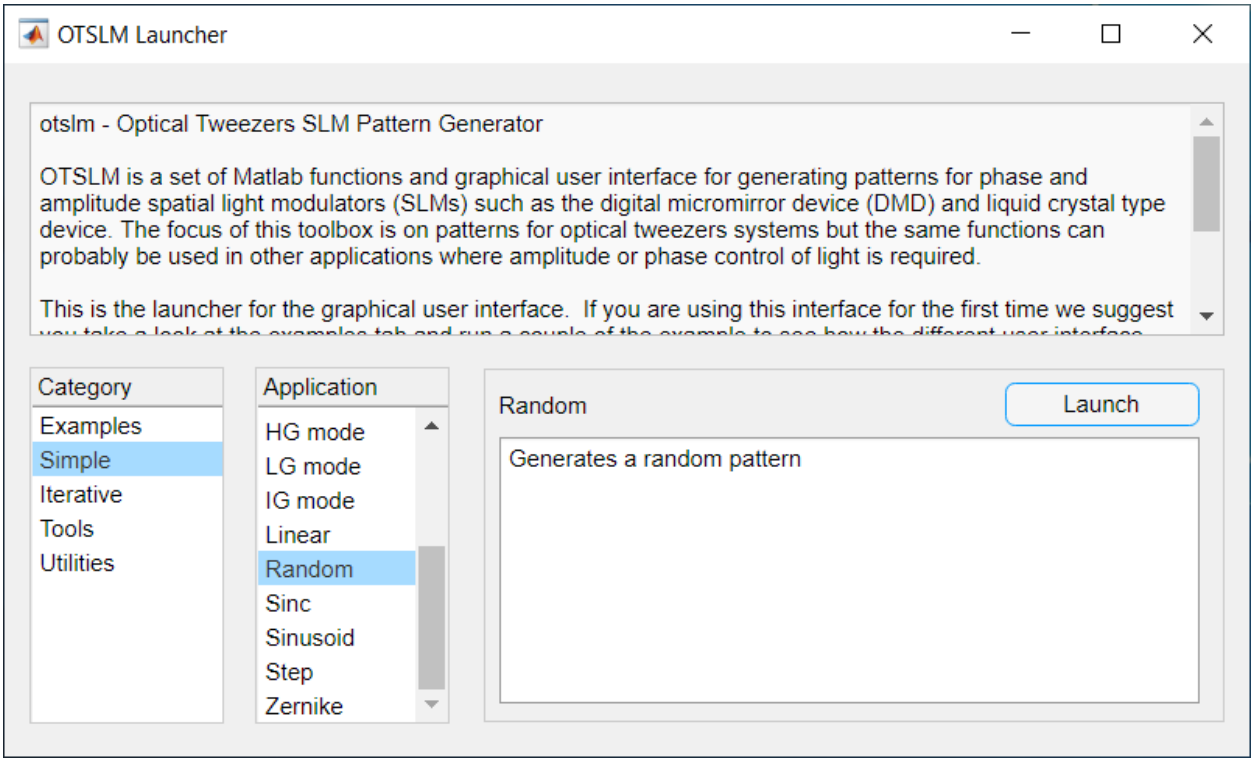

Fig. 2.1: Overview of the Launcher application.

The output from various applications can either be saved to the Matlab workspace or sent to a *`otslm.utils.Showable`* device (if one has already been configured). Applications which generate a pattern have an option to enter a Matlab variable name. When the pattern is generated, the image is saved to the current Matlab workspace.

Applications which take patterns as inputs (for example, combine and finalize) can use the patterns produced by another window by simply specifying the same variable name, for example see Fig. 2.2.

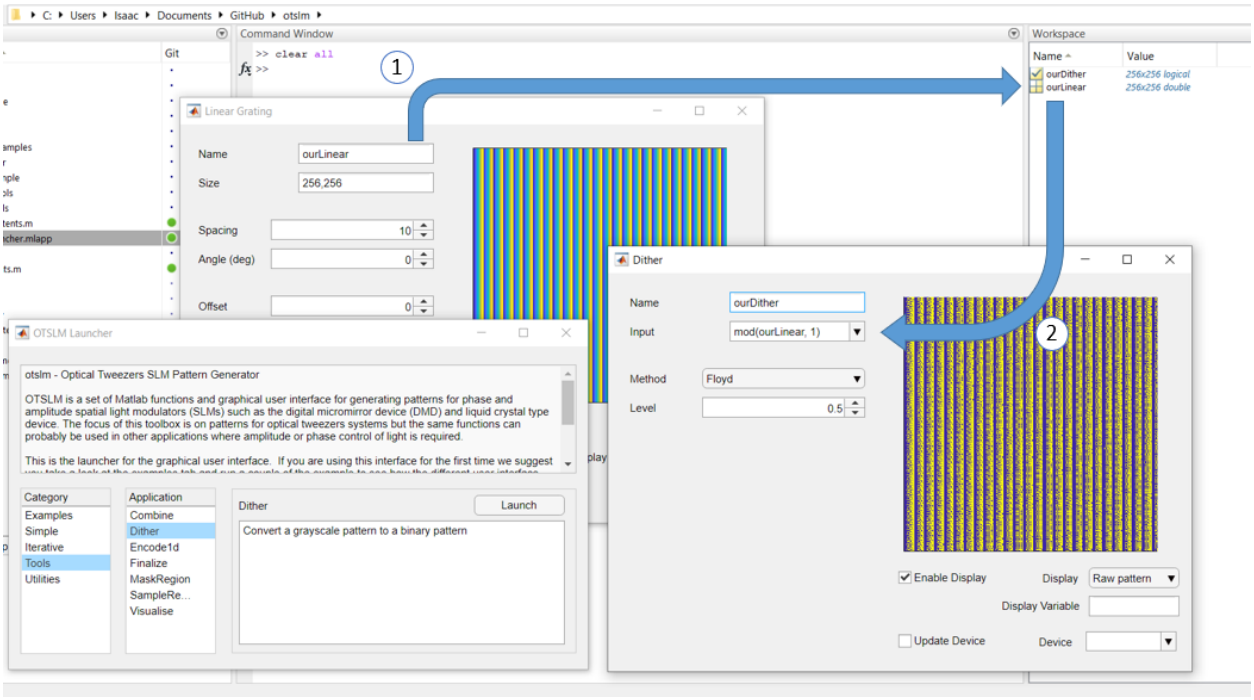

Fig. 2.2: Illustration showing dataflow between the GUI windows. A linear grating is generated with the name `outLinear`, when the pattern is ready it is saved to the Matlab workspace (1). This pattern can then be used by other interfaces, for example (2) shows the same variable name being used as an input to the Dither application.

If an app produces an error or warning, these will be displayed in the Matlab console.

The example applications show how the user interfaces can be combined to achieve a particular goal. To get started using the GUI, work through these examples. For additional information, see the *ui Package* documentation.

It is possible to customize these interfaces, however creating custom user interfaces in Matlab is rather time consuming and involves a lot of code duplication. Instead, we recommend using live scripts, see the *Gratings and Lens LiveScript* example. It is also possible to create a graphical user interfaces in LabVIEW, for details see *Accessing OTSLM from LabVIEW*.

2.3 Using the toolbox functions

The toolbox functions and classes are organised into four main packages: *simple Package*, *iter Package*, *tools Package* and *utils Package*. To use these functions, either prefix the function with `otslm` and the package name

```
im = otslm.simple.linear([10, 10], 3);
```

import a specific function

```
import otslm.simple.linear;
im = linear([10, 10], 3);
```

or import the entire package

```
import otslm.simple.*;
im1 = linear([10, 10], 3);
im2 = spherical([10, 10], 3);
```

Most of the toolbox functions produce/operate on 2-D matrices. The type of values in these matrices depends on the method, but values will typically be logical, double or complex. Complex matrices are typically used when the complex amplitude of the light field needs to be represented. Double matrices are used for both amplitude and phase patterns. Logicals are returned when the function could be used as a mask, for instance, *otslm.simple.aperture()* returns a logical array by default.

For phase patterns, there are three type of value ranges: `[0, 1)`, `[0, 2*pi)` and device specific colour range (after applying a lookup table to the pattern). Most of the *otslm.simple* functions return phase patterns between 0 and 1 or patterns which can be converted to this range using `mod(pattern, 1)`. To convert these patterns to the `[0, 2*pi)` range or apply a specific colour-map, you can use the *otslm.tools.finalize()* function.

To get started using the toolbox functions for beam shaping, take a look at the *Beams* and *Advanced Beams* examples. The `examples` directory provides examples of other toolbox functions and how they can be used.

To get help on toolbox functions or classes, type `help` followed by the OTSLM package, function, class or method name. For example, to get help on the *otslm.simple* package, type:

```
help otslm.simple
```

or to get help on the `run` method in the *otslm.iter.DirectSearch* class use

```
help otslm.iter.DirectSearch/run
```

For more extensive help, refer to this documentation.

CHAPTER 3

Examples

The toolbox has a range of examples in the `examples` directory. Additional information is provided here for some of these examples.

3.1 Simple Beams

This page describes the `examples.simple_beams` example. This example demonstrates some of the simpler hologram generation functions in the toolbox.

Contents

3.1.1 Initial setup

The example starts by adding the OTSLM toolbox to the path. The example script is in the `otslm-directory/examples/` directory, allowing us to specify the `otslm-directory` relative to the current directory with `../`

```
addpath('../');
```

We define a couple of properties for the patterns, starting with the size of the patterns `[512, 512]` and the incident illumination we will use for the simulation.

```
sz = [512, 512];

% incident = [];        % Incident beam (use default in visualize)
incident = otslm.simple.gaussian(sz, 150);  % Incident beam (gaussian)
% incident = ones(sz);  % Incident beam (use uniform illumination)
```

We use *otslm.simple.gaussian()* to create a Gaussian profile for the incident illumination. Alternatively we could just us the Matlab `ones()` function to create a uniform incident illumination or load a gray-scale image from a file.

The last part of the setup section defines a couple of functions for visualising the SLM patterns in the far-field.

```
o = 50;              % Region of interest size in output
padding = 500;        % Padding for FFT
zoom = @(im) im(round(size(im, 1)/2)+(-o:o), round(size(im, 2)/2)+(-o:o));
visualize = @(pattern) zoom(abs(otslm.tools.visualise(pattern, ...
    'method', 'fft', 'padding', padding, 'incident', incident)).^2);
```

This defines a function `visualize` which takes a pattern as input, uses the *otslm.tools.visualise()* method to simulate what the far-field looks like using the fast Fourier transform method, calculates the absolute value squared (converts from the complex output of *otslm.tools.visualise()* to an intensity image which we can plot with `imagesc()`) and zooms into a region of interest in the far-field image. This piece of code isn't really part of the example, it is only included to make the following sections more succinct. You could replace the use of the `visualize` function in the sections below with a single call the *otslm.tools.visualise()* and manually zoom into the resulting image.

3.1.2 Exploring different simple beams

The remainder of the example explores different beams. This section describes each beam phase pattern and shows the expected output.

Zero phase pattern

When a constant (or zero) phase pattern is placed on the SLM, the resulting beam is unmodified (except for a constant phase factor which doesn't affect the resulting intensity). When we visualise this beam, we should see the Fourier transform of the incident beam. If our incident beam is a Gaussian, we should see a Gaussian-like spot in the far-field, as shown in Fig. 3.1.

```
pattern = zeros(sz);
pattern = otslm.tools.finalize(pattern);
subplot(1, 2, 1), imagesc(pattern);
subplot(1, 2, 2), imagesc(visualize(pattern));
```

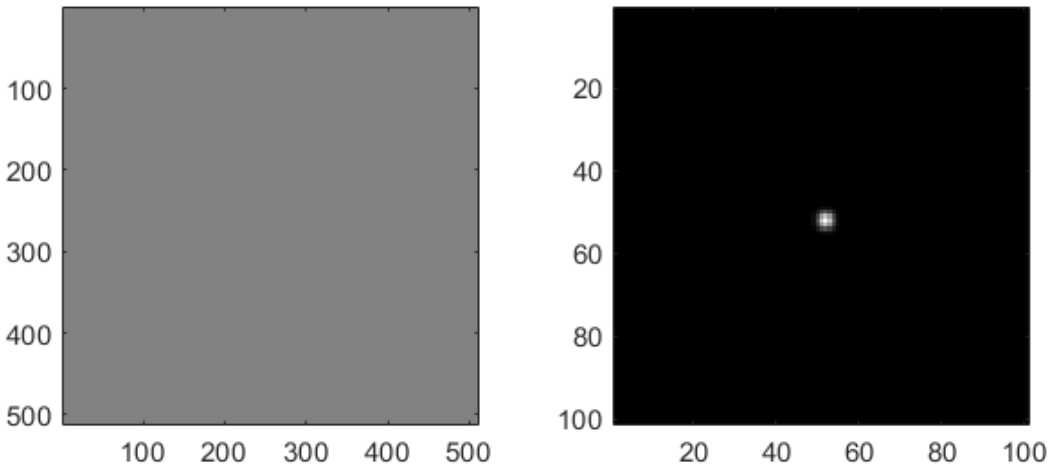

Fig. 3.1: A phase pattern with zero phase shift (left) and the resulting unchanged far-field intensity (right).

This example includes a call to *otslm.tools.finalize()*, for the zero phase pattern this call is redundant. If you changed the pattern to a constant uniform phase shift, for example `10.5*ones(sz)`, *otslm.tools.finalize()* would apply `mod(pattern, 1)*2*pi` to the pattern to ensure the pattern is between 0 and 2pi.

Linear grating

The linear grating can be used for shifting the focus of a beam in the far-field. The linear grating acts like a tilted mirror, on the side of the mirror where the path length is reduced the relative phase is less than zero, on the side of the mirror where the path length is increased the phase difference is larger. To create a linear grating you can use the *otslm.simple.linear()* function. This function has two required arguments, the pattern size and the grating spacing. The grating spacing is proportional to the distance the beam is displaced in the far-field and inversely proportional to the gradient of the pattern. Fig. 3.2 shows a typical output.

```
pattern = otslm.simple.linear(sz, 40, 'angle_deg', 45);
pattern = otslm.tools.finalize(pattern);
subplot(1, 2, 1), imagesc(pattern);
subplot(1, 2, 2), imagesc(visualize(pattern));
```

The *otslm.simple.linear()* function outputs a non-modulated pattern, as shown in Fig. 3.3. This makes it easier to combine the pattern with other patterns without introducing artefacts from applying `mod(pattern, 1)`. Passing the pattern to *otslm.tools.finalize()* applies the modulo to the pattern producing the recognisable blazed grating pattern.

Spherical grating

To shift the beam focus along the axial direction we can use a lens function. The toolbox includes a couple of simple *Lens functions*, here we use *otslm.simple.spherical()*. This function takes two required arguments: the pattern size and lens radius. Values outside the lens radius are invalid, we can choose how these values are represented using the `background` optional argument, in this case we choose to replace these values with a `checkerboard` pattern. The checkerboard pattern diffracts light to high angles (outside the range of the cropping in the `visualize` method).

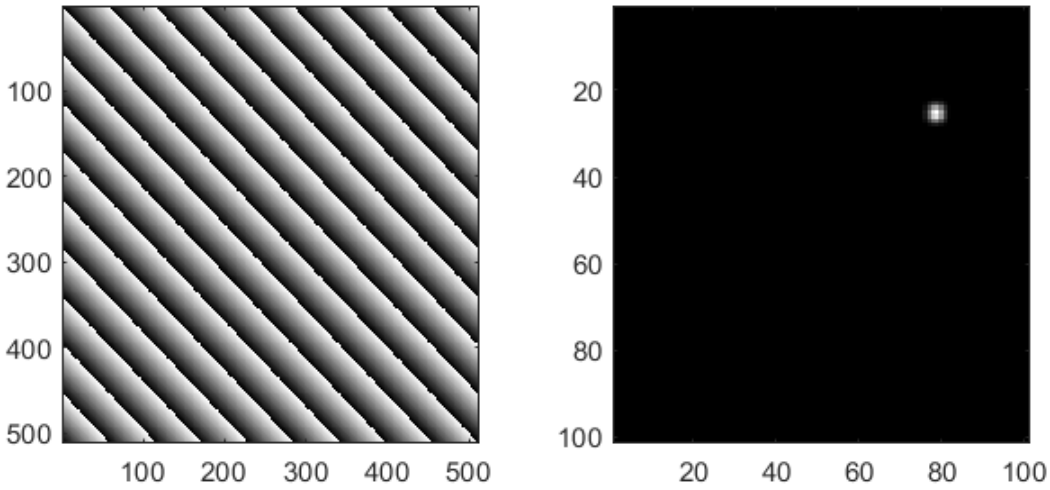

Fig. 3.2: A blazed grating generated using `otslm.simple.linear()` and and the resulting far-field intensity pattern.

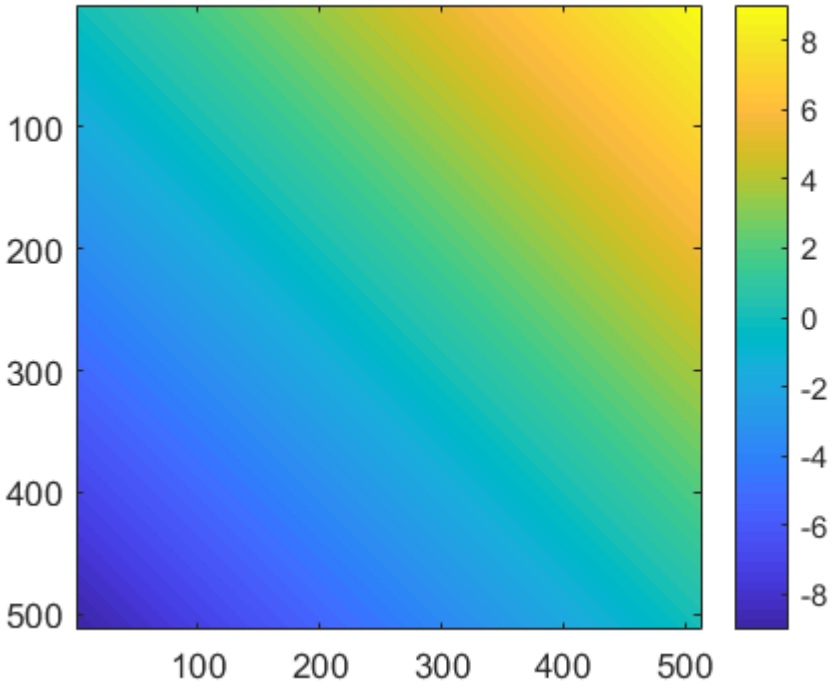

Fig. 3.3: Un-modulated output from `otslm.simple.linear()`.

By default, the spherical lens has a height of 1. We can scale the height by multiplying the output by the desired scale, this will scale the lens and the background pattern. To avoid applying the scaling to the background pattern we can use the `scale` optional argument. Typical output is shown in Fig. 3.4.

```
pattern = otslm.simple.spherical(sz, 200, 'scale', 5, ...
    'background', 'checkerboard');
pattern = otslm.tools.finalize(pattern);
subplot(1, 2, 1), imagesc(pattern);
subplot(1, 2, 2), imagesc(visualize(pattern));
```

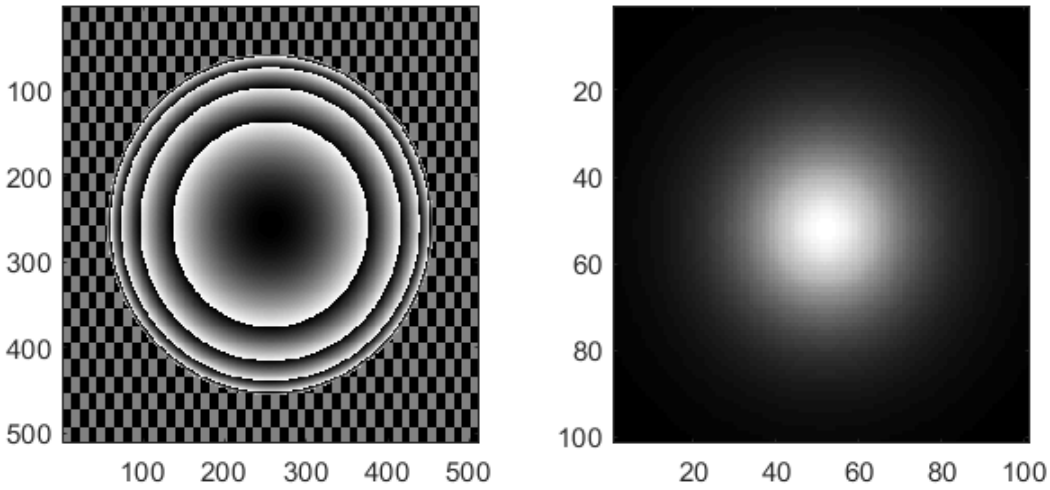

Fig. 3.4: Typical output and far-field intensity from the *otslm.simple.spherical()* function. The output has been modulated to produce the recognisable Fresnel-style lens pattern.

The output of *otslm.simple.spherical()* is non-modulated, similar to *otslm.simple.linear()* described above. Only when `otslm.tools.finalize`()` is applied does the pattern look like a Fresnel lens pattern.

LG Beam

The toolbox provides methods for generating the amplitude and phase patterns for LG beams. To calculate the phase profile for an LG beam, we can use *otslm.simple.lgmode()*. This function takes as inputs the pattern size, azimuthal and radial modes and an optional scaling factor for the radius of the pattern. Typical output is shown in Fig. 3.5.

```
amode = 3;  % Azimuthal mode
rmode = 2;  % Radial mode
pattern = otslm.simple.lgmode(sz, amode, rmode, 'radius', 50);
pattern = otslm.tools.finalize(pattern);
subplot(1, 2, 1), imagesc(pattern);
subplot(1, 2, 2), imagesc(visualize(pattern));
```

In order to generate a pure LG beam we need to be able to control both the amplitude and phase of the light. This can be achieved using separate devices for the amplitude and phase modulator or by mixing the amplitude pattern into the phase, as is described in the *Advanced Beams* example.

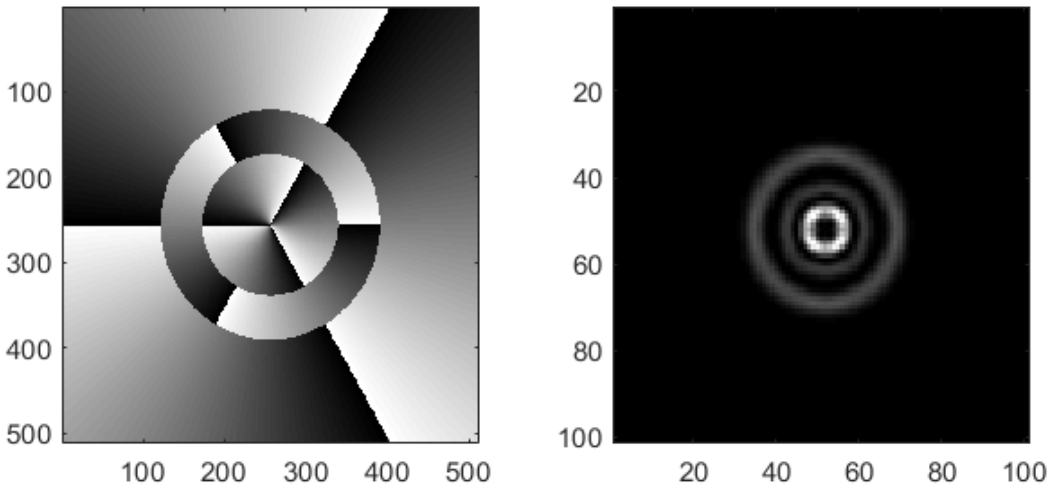

Fig. 3.5: Example output from *otslm.simple.lgmode()*.

HG Beam

Amplitude and phase patterns can be calculated using the *otslm.simple.hgmode()* function. The output from this function is shown in Fig. 3.6. This function takes as input the pattern size and the two mode indices. There is also an optional `scale` parameter for scaling the pattern. As with LG beams, generation of pure HG beams requires control of both the phase and amplitude of the light, see the *Advanced Beams* example for more details.

```
pattern = otslm.simple.hgmode(sz, 3, 2, 'scale', 70);
pattern = otslm.tools.finalize(pattern);
subplot(1, 2, 1), imagesc(pattern);
subplot(1, 2, 2), imagesc(visualize(pattern));
```

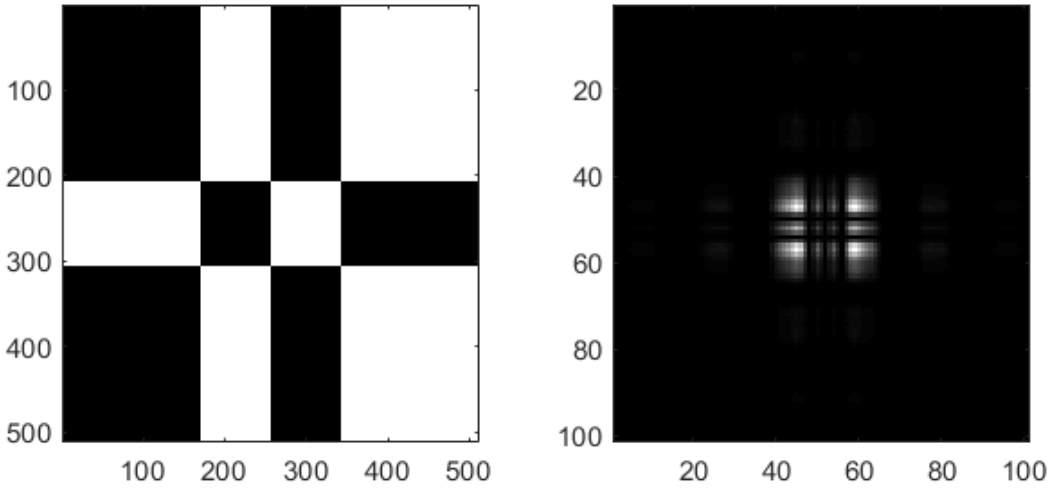

Fig. 3.6: Phase pattern (left) generated using the *otslm.simple.hgmode()* method and the corresponding simulated far-field (right). The simulated far-field doesn't have any amplitude correction, leading to a non-pure HG beam output.

Sinc pattern

A sinc amplitude pattern can be used to generate a line-shaped focal spot in the far-field. For phase-only SLMs, we need to *encode* the amplitude in the phase pattern, this can be achieved by mixing the pattern with a second phase pattern (as described in *Advanced Beams*), or for 1-D patterns we can encode the amplitude into the second dimension of the SLM (similar to Roichman and Grier, Opt. Lett. 31, 1675-1677 (2006)). In this example, we show the latter.

First we create the `sinc` profile using the *`otslm.simple.sinc()`* function. This function takes two required arguments, pattern size and the sinc radius. The function can generate both 1-dimensional and 2-dimensional sinc patterns, but for the 1-D encoding method we need a 1-dimensional pattern, as shown in Fig. 3.7.

```
radius = 50;
sinc = otslm.simple.sinc(sz, 50, 'type', '1d');
```

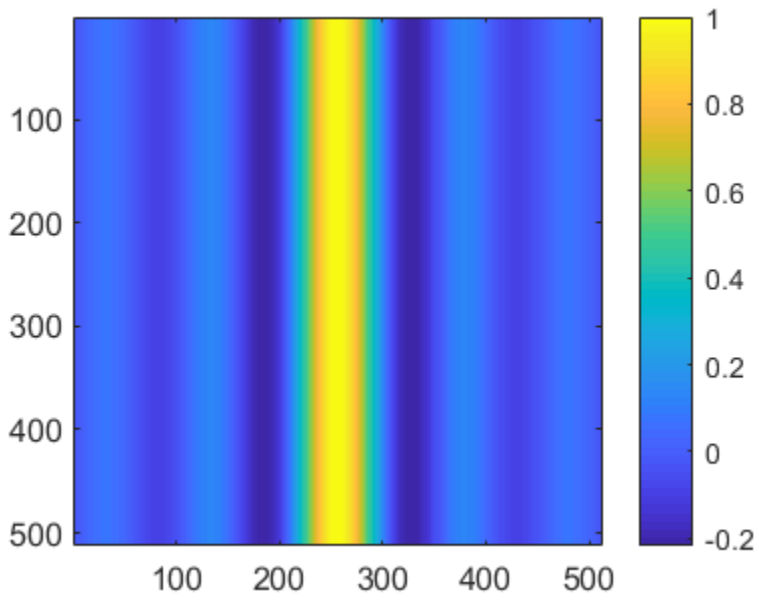

Fig. 3.7: 1-dimensional sinc pattern output from *`otslm.simple.sinc()`*.

To encode the 1-dimensional pattern into the second dimension of the SLM we can use *`otslm.tools.encode1d()`*. This method takes a 2-D amplitude image, the amplitude should be constant in one direction and variable in the other direction. For the above image, the amplitude is constant in the vertical direction and variable in the horizontal direction. The method determines which pixels have a value greater than the location of the pixel in the vertical direction. Pixels within this range are assigned the phase of the pattern (0 for positive amplitude, 0.5 for negative amplitudes). Pixels outside this region should be assigned another value, such as a checkerboard pattern. The encode method also takes an optional argument to scale the pattern by, this can be thought of as the ratio of pattern amplitude and device height.

```
[pattern, assigned] = otslm.tools.encode1d(sinc, 'scale', 200);

% Apply a checkerboard to unassigned regions
checker = otslm.simple.checkerboard(sz);
pattern(~assigned) = checker(~assigned);
```

We can then finalize and visualise our pattern to produce Fig. 3.8.

```
pattern = otslm.tools.finalize(pattern);
subplot(1, 2, 1), imagesc(pattern);
subplot(1, 2, 2), imagesc(visualize(pattern));
```

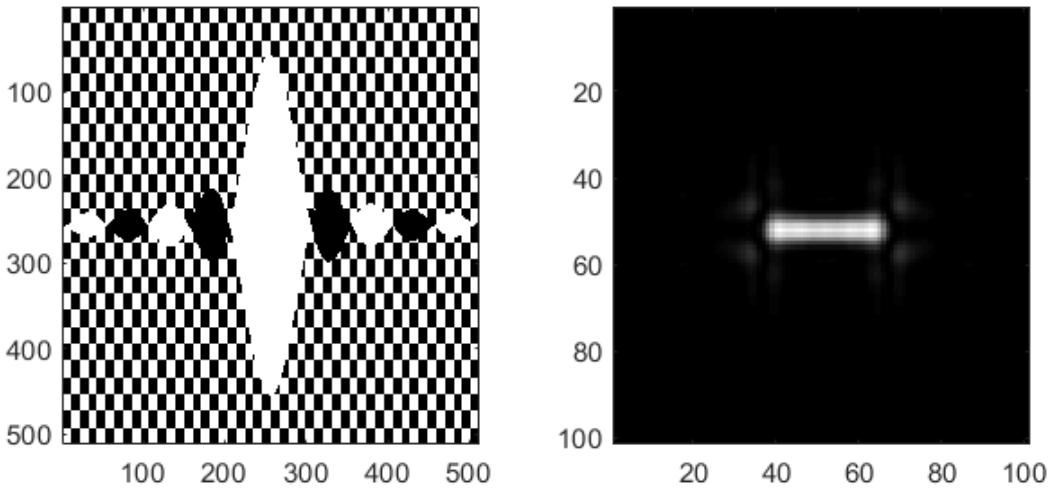

Fig. 3.8: Sinc pattern encoded with *otslm.tools.encode1d()*.

Axicon lens

An axicon (cone shaped) lens can be useful for creating Bessel-like beams in the near-field. In the far-field, the light will have a ring-shaped profile, while in the near-field the light should have a Bessel-like profile. It is also possible to combine the axicon lens with an azimuthal gradient to generate Bessel-like beams with angular momentum. Example output is shown in Fig. 3.9.

```
radius = 50;
pattern = otslm.simple.axicon(sz, -1/radius);
pattern = otslm.tools.finalize(pattern);
subplot(1, 2, 1), imagesc(pattern);
subplot(1, 2, 2), imagesc(visualize(pattern));
```

To see the Bessel-shaped profile, we need to look at the near-field. We can use the *otslm.tools.visualise()* method with a `z` offset to view the near-field of the axicon, as shown in Fig. 3.10.

```
im1 = otslm.tools.visualise(pattern, 'method', 'fft', 'trim_padding', true, 'z',␣
↪50000);
im2 = otslm.tools.visualise(pattern, 'method', 'fft', 'trim_padding', true, 'z',␣
↪70000);
im3 = otslm.tools.visualise(pattern, 'method', 'fft', 'trim_padding', true, 'z',␣
↪90000);
figure();
subplot(1, 3, 1), imagesc(zoom(abs(im1).^2)), axis image;
subplot(1, 3, 2), imagesc(zoom(abs(im2).^2)), axis image;
subplot(1, 3, 3), imagesc(zoom(abs(im3).^2)), axis image;
```

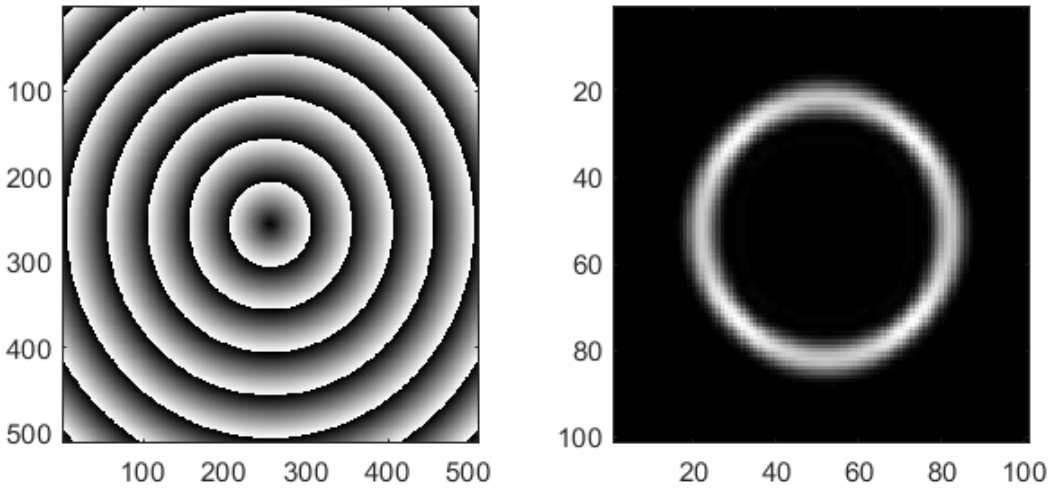

Fig. 3.9: Axicon pattern (left) and simulated far-field (right).

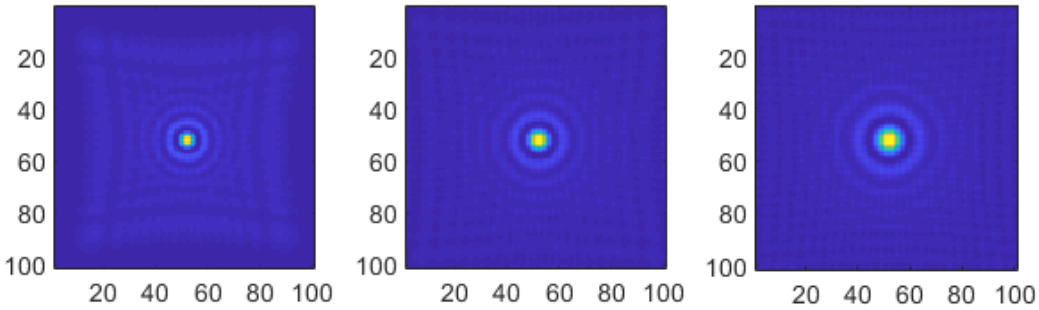

Fig. 3.10: Visualisation of the near-field of the axicon pattern using `otslm.tools.visualise()` with three different axial offsets.

Cubic lens

The cubic lens pattern *otslm.simple.cubic()* can be used to create airy beams. Fig. 3.11 shows example output.

```
pattern = otslm.simple.cubic(sz);
pattern = otslm.tools.finalize(pattern);
subplot(1, 2, 1), imagesc(pattern);
subplot(1, 2, 2), imagesc(visualize(pattern));
```

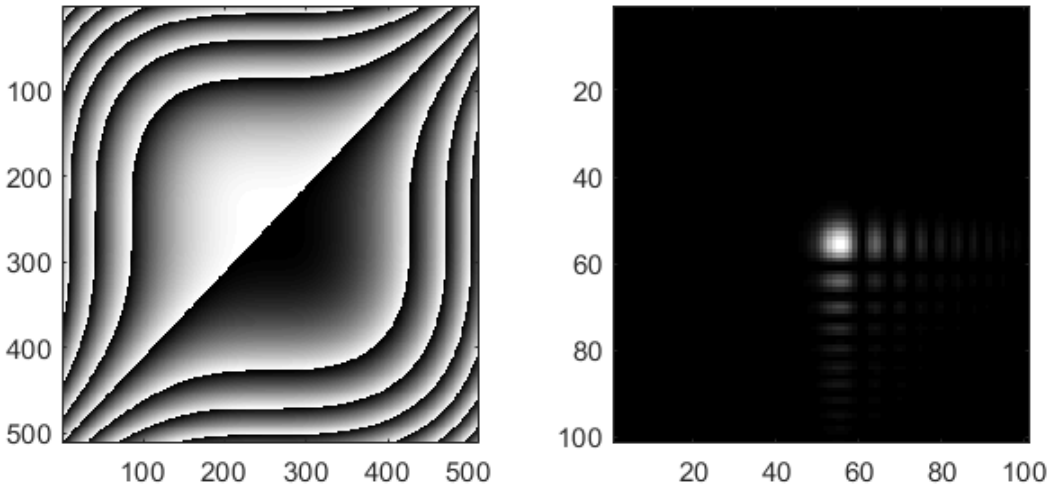

Fig. 3.11: Example output using *otslm.simple.cubic()*.

3.2 Advanced Beams

This page describes the `examples.advanced_beams` example. This example demonstrates some of the more complex hologram generation capabilities in the toolbox including: combining multiple holograms, shaping the amplitude with a phase-only device, iterative algorithms, and binary amplitude patterns.

Note: Many of the images in this documentation include checkerboard patterns. The checkerboard pattern should have a width of 1 pixel to scatter light to high angles, however the lower resolution images shown in the documentation appear to have a courser checkerboard pattern as a result of a Moiré/aliasing effect. To use these patterns, we recommend generating higher resolution versions using the toolbox.

Contents

3.2.1 Initial setup

The start of the script defines parameters and functions for visualising the far-field of the SLM. This is mostly the same as the initial setup in the *Beams* example. Some of the advanced beams include a beam amplitude correction term to compensate for the non-uniform illumination of the pattern from the incident beam. The beam correction term is defined as

```
beamCorrection = 1.0 - incident + 0.5;
beamCorrection(beamCorrection > 1.0) = 1.0;
```

3.2.2 Amplitude control with a phase device

In the *LG Beam* and *HG Beam* examples in `examples.simple_beams` we noted how in order to create pure LG or HG beams we need to control both the phase and amplitude of the beam. In the *Sinc pattern* example we used the *`otslm.tools.encode1d()`* method to encode a 1-dimensional pattern into a 2-dimensional phase pattern. For encoding two dimensional phase patterns we need to create a mixture of two patterns: the pattern we want to generate and a second pattern which scatters light into another direction. Common choices for the second pattern include:

- a uniform pattern, which would leave light in the centre of the beam
- a checkerboard pattern, which would scatter light into large angles, which can easily be filtered with a iris
- a linear grating to deflect light to a specific point
- another desired part of the far-field intensity profile

Creating a HG beam

To create the HG beam, we use the *`otslm.simple.hgmode()`* function we used in the simple beams example, except this time we request both the phase and amplitude outputs:

```
[pattern, amplitude] = otslm.simple.hgmode(sz, 3, 2, 'scale', 50);
```

To combine the phase, amplitude and beam correction factor, which accounts for the non-uniform illumination, we can pass the amplitude terms into *`otslm.tools.finalize()`*:

```
pattern = otslm.tools.finalize(pattern, ...
    'amplitude', beamCorrection.*abs(amplitude));
```

The finalize method generates a phase mask that is a mixture of the desired phase pattern and a checkerboard pattern depending on the amplitude. Internally, the method implements:

```
background = otslm.simple.checkerboard(size(pattern), ...
    'value', [-1, 1]);

% This ratio depends on the background level
% Amplitude must be between -1 and 1
mixratio = 2/pi*acos(abs(amplitude));

% Add the amplitude and mix with the background
pattern = pattern + angle(amplitude)/(2*pi)+0.5;
pattern = pattern + mixratio.*angle(background)/(2*pi)+0.5;
```

The final result, shown in Fig. 3.12, is something that looks a lot more like a HG beam than the simple beams example

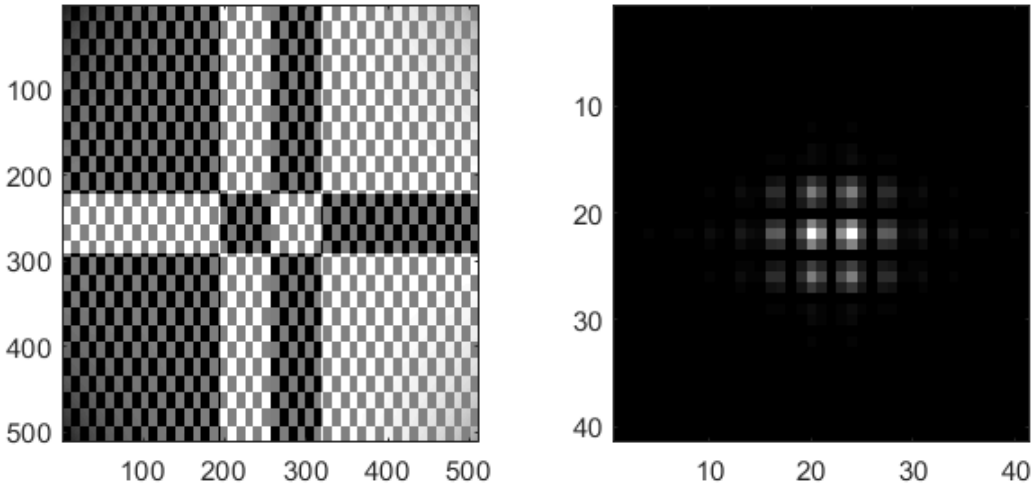

Fig. 3.12: A phase pattern (left) to generate a HG beam in the far-field (right). This pattern accounts for non-uniform incident illumination.

Creating a Bessel beam

A bessel-like beam can be created in the far-field of the SLM by creating a annular ring on the device. The phase of the ring can be constant for Bessel beams without angular momentum, or an azimuthal phase can be added for Bessel beams with angular momentum. To create the Bessel beam, we need a ring with a finite power and infinitely small thickness. This is difficult to achieve, so instead it is better to create a ring with a finite thickness, for this we can use the *otslm.simple.aperture()* function to create a ring. We can replace the regions outside the aperture with a checkerboard pattern to scatter the light to high angles. Example output is shown in Fig. 3.13.

```
pattern = otslm.simple.aperture(sz, [ 100, 110 ], 'shape', 'ring');

% Coorect for amplitude of beam
pattern = pattern .* beamCorrection;

% Finalize pattern
pattern = otslm.tools.finalize(zeros(sz), 'amplitude', pattern);
```

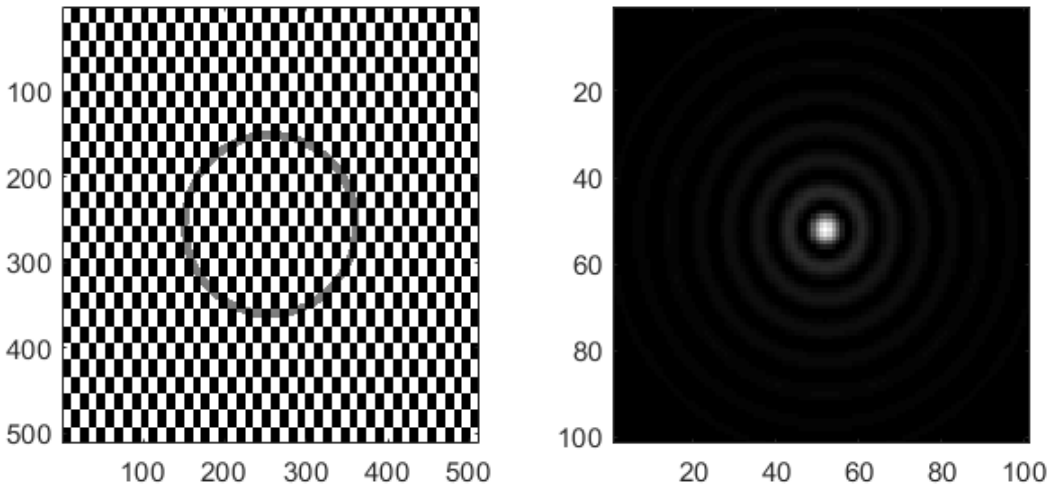

Fig. 3.13: A bessel-like beam generated using a finite thickness ring. A checkerboard pattern is used to scatter unwanted light away from the desired beam.

3.2.3 Combining patterns

There are multiple methods for combining beams. The phases can be added or multiplied or the complex amplitudes can be added or multiplied.

Adding phase patterns

Beam phase patterns can be added together at any time. This can be useful for beam steering, for example, a linear grating or a lens could be added to another pattern to shift the location in the focal plane. It is often better to add the phase patterns before calling the finalize method, since the finalize method applies the modulo to the patterns which may introduce additional artefacts if patterns are added after this operation. An example is shown in Fig. 3.14.

```
pattern = otslm.simple.lgmode(sz, 3, 2, 'radius', 50);
pattern = pattern + otslm.simple.linear(sz, 30);
pattern = otslm.tools.finalize(pattern);
```

Superposition of beams

To create a superposition of different beams we can combine the complex amplitudes of the individual beams. To do this, we can use the *otslm.tools.combine()* function. This function provides a range of methods for combining beams, here we will demonstrate the `super` method. The combine function accepts additional arguments for weighted super-positions and also supports adding random phase offsets using the `rsuper` method. The following code demonstrates using the `super` method, the output is shown in Fig. 3.15.

```
pattern1 = otslm.simple.linear(sz, 30, 'angle_deg', 90);
pattern2 = otslm.simple.linear(sz, 30, 'angle_deg', 0);

pattern = otslm.tools.combine({pattern1, pattern2}, ...
    'method', 'super');

pattern = otslm.tools.finalize(pattern);
```

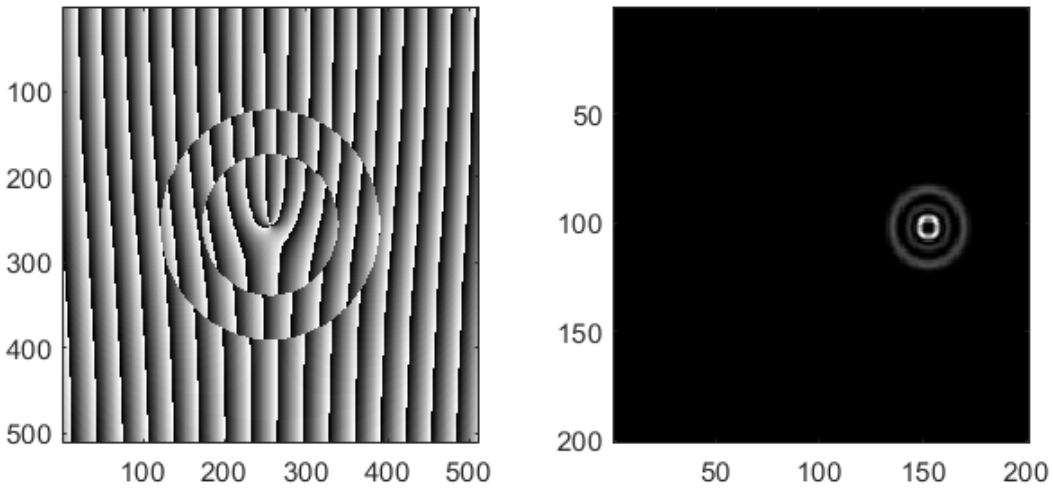

Fig. 3.14: A linear ramp, generated with `otslm.simple.linear()`, is added to a LG beam phase mask to shift the location of the LG beam in the farfield (right).

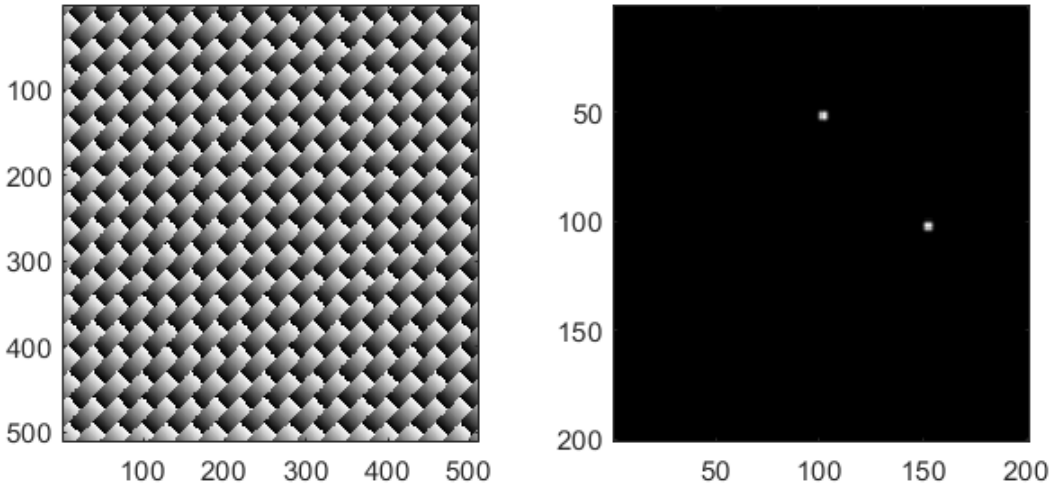

Fig. 3.15: Demonstration of `otslm.tools.combine()` for combining two linear gratings using the super-position method.

Arrays of patterns

By adding a grating, such as a 2-D sinusoidal grating, to the pattern it is possible to create arrays of similar spots. This can be a quick method for creating an array of optical traps for interacting with many similar samples. The following example shows how a sinusoid grating can be combined with a LG-mode pattern to create the output shown in Fig. 3.16.

```
lgpattern = otslm.simple.lgmode(sz, 5, 0);
grating = otslm.simple.sinusoid(sz, 50, 'type', '2dcart');

pattern = lgpattern + grating;
pattern = otslm.tools.finalize(pattern, 'amplitude', beamCorrection);
```

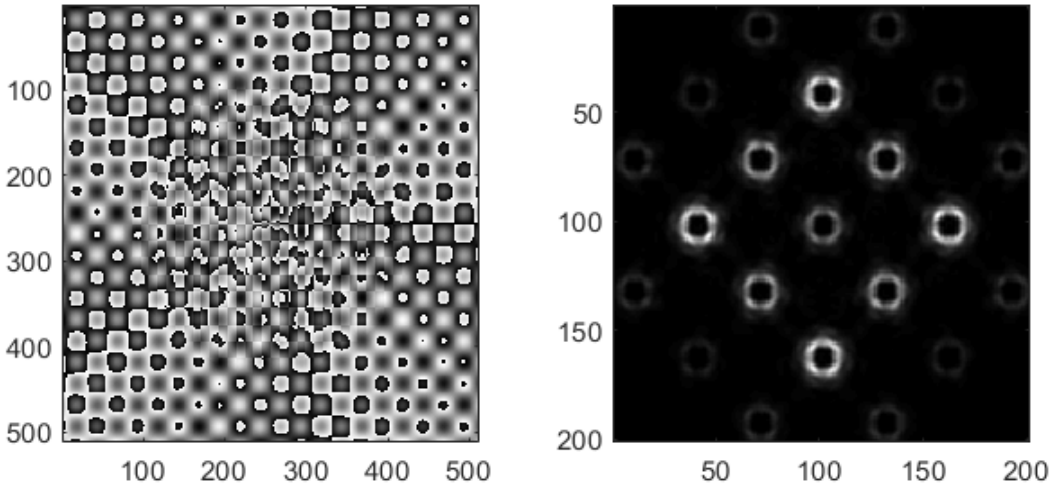

Fig. 3.16: An array of beams generated using a sinusoidal grating.

Selecting regions of interest

Spatial light modulators can be used for creating beams and sampling light from specific regions of beams for novel imaging applications. The toolbox provides a method to help with creating region masks for sampling different regions of the device. In this example, we show how *otslm.tools.mask_regions()* can be used to sample three regions of the device to create three separate beams.

The first stage is to setup three different spots. We specify the location of each spot, the radius and the pattern. We use `otslm.tool.finalize()` to apply amplitude corrections and apply the modulo to the patterns but we request the output remain in the range `[0, 1)`.

```
loc1 = [ 170, 150 ];
radius1 = 75;
pattern1 = otslm.simple.lgmode(sz, 3, 0, 'centre', loc1);
pattern1 = pattern1 + otslm.simple.linear(sz, 20);
pattern1 = otslm.tools.finalize(pattern1, 'amplitude', beamCorrection, ...
    'colormap', 'gray');

loc2 = [ 320, 170 ];
radius2 = 35;
pattern2 = zeros(sz);
```

(continues on next page)

(continued from previous page)

```
loc3 = [ 270, 300 ];
radius3 = 50;
pattern3 = otslm.simple.linear(sz, -20, 'angle_deg', 45);
pattern3 = otslm.tools.finalize(pattern3, 'amplitude', 0.4, ...
    'colormap', 'gray');
```

For the background we use a checkerboard pattern.

```
background = otslm.simple.checkerboard(sz);
```

To combine the patterns, we call *otslm.tools.mask_regions()* with the background pattern, the region patterns, their locations, radii and the mask shape (in this case a circle). We then call *otslm.tools.finalize()* to rescale the resulting pattern from the `[0, 1)` range to the `[0, 2pi)` range needed for the visualisation. The output is shown in Fig. 3.17.

```
pattern = otslm.tools.mask_regions(background, ...
    {pattern1, pattern2, pattern3}, {loc1, loc2, loc3}, ...
    {radius1, radius2, radius3}, 'shape', 'circle');

pattern = otslm.tools.finalize(pattern);
```

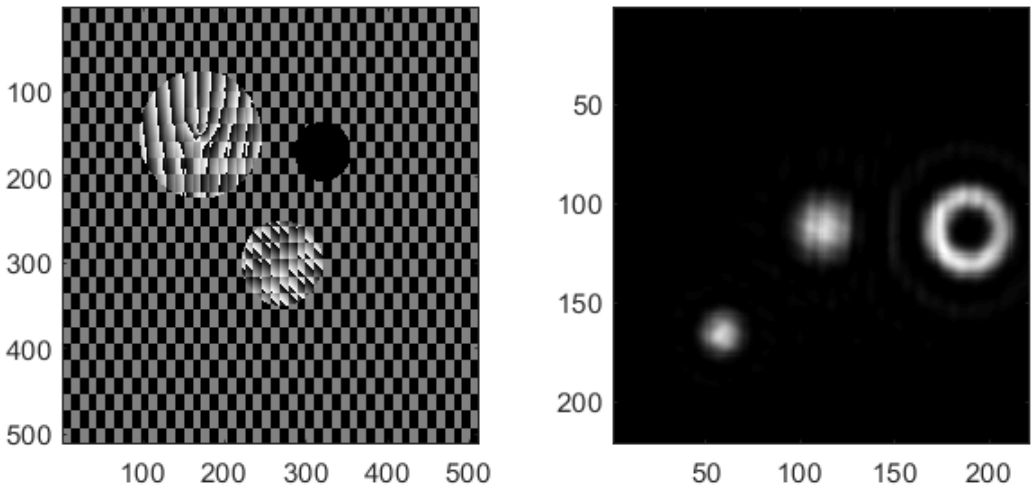

Fig. 3.17: Example output from `otslm.tools.mask_regions()` sampling three regions of interest.

3.2.4 Gerchberg-Saxton

The toolbox provides a number of iterative algorithms for generating patterns. One such algorithm is the Gerchberg-Saxton algorithm. This method attempts to approximate the desired light field by iteratively moving between the near-field and far-field. A more detailed overview of the algorithm can be found in the *GerchbergSaxton* section later in the documentation.

In OTSLM, most iterative algorithms are implemented as Matlab classes. To use the `GerchbergSaxton` class, we need to specify the target image. Additionally, we can specify the propagation methods to use to go between the near-field and far-field and an initial guess. In this example, we setup a propagator with the incident illumination

```
prop = otslm.tools.prop.FftForward.simpleProp(zeros(sz));
vismethod = @(U) prop.propagate(U .* incident);
```

and then create an instance of the iterator class. `GerchbergSaxton` also implements the adaptive-adaptive algorithm via the `adaptive` optional parameter, see the documentation for additional details.

```
target = otslm.simple.aperture(sz, sz(1)/20);
gs = otslm.iter.GerchbergSaxton(target, 'adaptive', 1.0, ...
    'vismethod', vismethod);
```

To run the algorithm, we simply need to call run with the number of iterations we would like to run for. The run method returns the complex amplitude pattern from the output of the last iteration. To retrieve the phase pattern, we can simply access the `phase` class member. This phase pattern has a range of 0 to 2pi, therefore it does not need to be passed to *`otslm.tools.finalize()`* before visualisation. Fig. 3.18 shows example output from this method.

```
gs.run(20);
pattern = gs.phase;
```

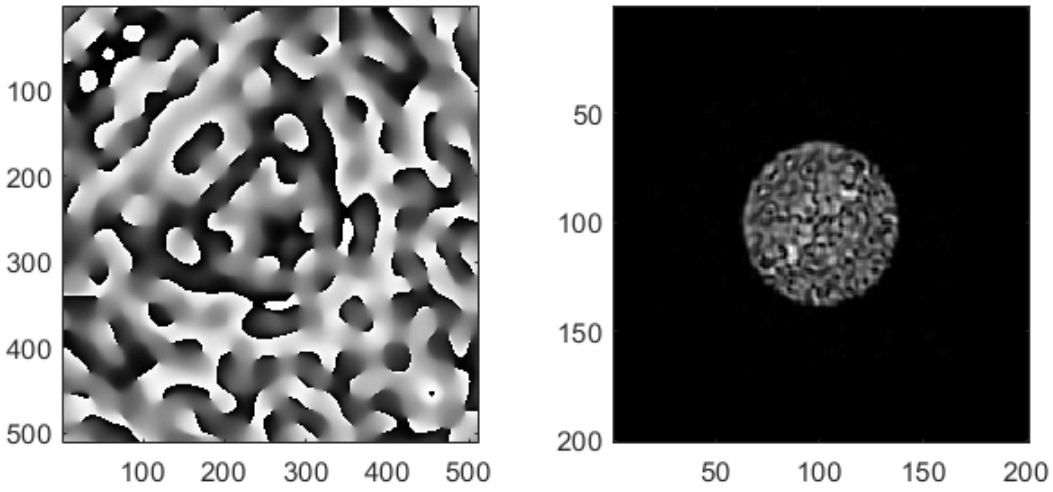

Fig. 3.18: Phase pattern generated using Gerchberg-Saxton (left) and the simulated far-field (right).

3.2.5 Creating patterns for the DMD

A digital micro-mirror device (DMD) is a binary amplitude spatial light modulator which consists of square pixels arranged in a diagonal lattice. The arrangement of pixels means that the device has a 1:2 aspect ratio. Although the device can only control the amplitude of individual pixels, it is still possible to create masks which control both the phase and amplitude of the resulting beam.

In this example, we create a LG beam using a binary amplitude pattern, following a similar approach to Lerner et al., Opt. Lett.37 (23) 4826–4828 (2012). We need to use a different size and aspect ratio for the DMD, for this example we will use a device with 512x1024 pixels.

```
dmdsz = [512, 1024];
aspect = 2;
```

To create the LG-mode pattern, we can use the *`otslm.simple.lgmode()`* function. This function has an optional argument for the aspect ratio and returns both the amplitude and phase for the pattern.

```
[phase, amplitude] = otslm.simple.lgmode(dmdsz, 3, 0, ...
    'aspect', aspect, 'radius', 100);
```

The DMD diffraction efficiency when controlling both the phase and amplitude is fairly low, so we expect there to be a significant amount of light left in the zero order. We can shift our LG beam away from the zero order light using a linear diffraction grating. There are also artefacts from the hard edges of the square (diamond) shaped pixels, to avoid these artefacts we rotate the linear grating.

```
phase = phase + otslm.simple.linear(dmdsz, 40, ...
    'angle_deg', 62, 'aspect', aspect);
```

For this example we are going to assume uniform illumination. To encode both the amplitude and phase into the amplitude-only pattern we can use the finalize function and specify that the device is a DMD and the colormap is grayscale. By default, the finalize function assumes DMDs should be rotated (packed) differently, however we want to leave our pattern unchanged for now and explicitly rotate it at a later stage, so we pass `none` as the `rpack` option.

```
pattern = otslm.tools.finalize(phase, 'amplitude', amplitude, ...
    'device', 'dmd', 'colormap', 'gray', 'rpack', 'none');
```

At this stage, the pattern is for a continuous amplitude device. To convert the continuous amplitude to a binary amplitude, we can use *otslm.tools.dither()*. It is possible to do this all in one step using one call to *otslm.tools.finalize()* but this allows additional control over the dither.

```
pattern = otslm.tools.dither(pattern, 0.5, 'method', 'random');
```

Up until now, our pattern has been in device pixel coordinates. In order to visualise what the pattern will look like in the far-field we need to re-map the device pixel coordinates to the 1:2 aspect ratio found on a physical device. For this we can use *otslm.tools.finalize()* again, this time with the `rpack` argument set to `45deg`. We explicitly set no modulo and a gray-scale colour-map again, however our pattern is already binary so the output will still be zeros and ones.

```
patternVis = otslm.tools.finalize(pattern, ...
    'colormap', 'gray', 'rpack', '45deg', 'modulo', 'none');
```

The final step is to visualise the pattern. For this we create a uniform incident illumination and we call the *otslm.tools.visualise()* method with no phase. The output is shown in Fig. 3.19.

```
dmdincident = ones(size(patternVis));

visOutput = abs(otslm.tools.visualise([], 'amplitude', patternVis, ...
    'method', 'fft', 'padding', padding, 'incident', dmdincident)).^2;

% Zoom into the resulting pattern
visOutput = visOutput(ceil(size(visOutput, 1)/2)-50+(-40:40), ...
    ceil(size(visOutput, 2)/2 +(-40:40)));
```

3.3 Gratings and Lens LiveScript

There are a number of ways to use OTSLM including traditional Matlab scripts, graphical user interfaces (`.mlapp` files) and live scripts. Live scripts offer a method of adding graphical user interface components (widgets) to your script to allow users to easily change script parameters, see for example Fig. 3.20. They are a far simpler method for providing a graphical user interface experience compared to developing a Matlab application. Additionally, they can be used to generate formatted output.

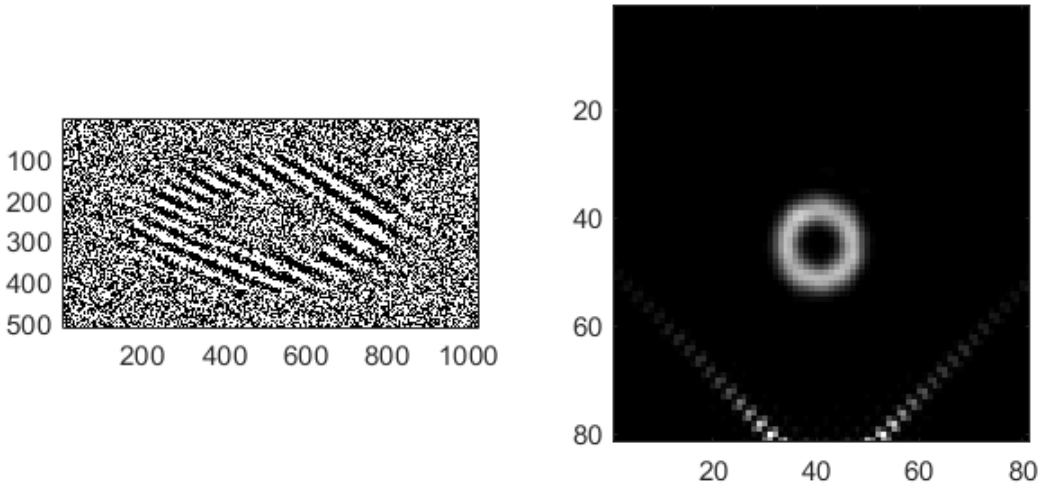

Fig. 3.19: Binary amplitude DMD pattern (left) generating an LG-beam beam in the far-field (right).

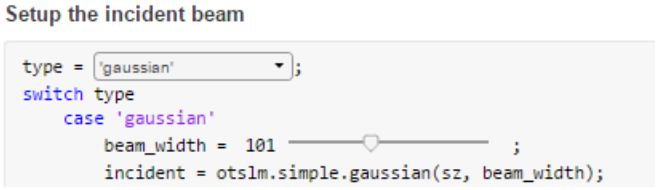

Fig. 3.20: Screenshot showing two different types of widgets that can be created in a live script.

The live script can be used to generate PDF and HTML output files, for an example, view the online documentation or `docs/_static/GratingAndLens.html` downloaded with OTSLM.

At this stage we have provided a single example of using a live script to control OTSLM: `examples.liveScripts.GratingAndLens`. This live script demonstrates the basics of using OTSLM to generate a pattern, simulate the far-field and display the image using a screen device. The pattern generation and simulation functionality is normal Matlab code except for the addition of various widgets for controlling/setting different options. The major difference between live scripts and traditional scripts is how they interact with *otslm.utils.ScreenDevice*.

Setting up a *ScreenDevice* in a live script takes a bit more work than usual. Live scripts have their own non-visible figure object for plotting. When we show the *ScreenDevice* window, a new visible figure object is created. After we show the *ScreenDevice* window for the first time we need to tell the live script to use the old figure for internal plots, otherwise it will replace the *ScreenDevice* output every time we change a slider value. To achieve this, we use the following section of code

```
if 0 == exist('sd', 'var') || ~ishandle(sd.figure_handle)
    sd = otslm.utils.ScreenDevice(1, 'size', sz,...
        'pattern_type', 'phase', 'prescaledPatterns', true);

    % Get the figure handle for the livescript
    % We need to do this to make ScreenDevice run correctly
    figureHandle = gcf();

    % Show the ScreenDevice figure
    sd.show();

    % Change back to the liveScript figure handle
    % This also needs to be done if we click on another figure
    set(0, 'CurrentFigure', figureHandle);
end
```

3.4 Using the GPU

The toolbox provides methods for accelerating the computation of holograms by taking advantage of the computer graphics hardware. There are two approaches for using the graphics hardware: (1) use the GPU as a co-processor, sending instructions to be evaluated on the device as is done in HOTlab; and (2) load a custom shader into the screen render pipeline as is done in RedTweezers.

Both these approaches have advantages and disadvantages. Communication with the graphics hardware for co-processing is typically done using very general languages such as CUDA or OpenCL which do not have direct access to the render pipeline. Instructions and data is sent to the device and the completed image is downloaded from the device once the calculation is complete. In order to display a pattern on the screen, the image must be copied back to the graphics hardware, introducing an additional delay/overhead. The copy requirement is not a problem when the intended target for the pattern is not the screen, for instance, if the pattern is being saved to a file or sent over another connection such as via USB, both these operations would require the pattern to be copied regardless.

Instead, the pattern can be calculated as part of the graphics render pipeline. This can be achieved by loading a custom OpenGL shader program into the graphics pipeline. Unlike CUDA or OpenCL, the OpenGL shader language (GLSL) is optimized for drawing to the screen: GLSL programs are compiled and loaded into the render pipeline. In contrast to CUDA/OpenCL, which allow commands and data to be sent to the hardware, a GLSL shader only allows data to be sent to the pre-compiled shader. The shader must be recompiled every time the render pipeline changes, for instance if we were to change from displaying linear gratings to sinc patterns.

Both co-processing (via Matlab `gpuArrays`) and GLSL shaders (via RedTweezers) are implemented in OTSLM, they are described in the following sections. Although it may be possible to achieve interoperability between CUDA/OpenCL and OpenGL, these features are not currently implemented.

Contents

3.4.1 Using the GPU as a co-processor

Matlab supports calculations on the GPU via `gpuArray` objects. This requires the Matlab Parallel Computing Toolbox and a compatible CUDA enabled graphics card. Functions which create textures can be passed a additional parameter `'gpuArray', true` to enable using `gpuArrays`.

```
im = otslm.simple.checkerboard([1024, 1024], 'gpuArray', true);
```

This pattern remains on the GPU until copied back. It is better to keep the pattern on the GPU until we are finished with it. We can perform operations on this pattern in a similar way to normal Matlab matrices, for instance

```
sz = [1024, 1024];
pattern = otslm.simple.checkerboard(sz, 'gpuArray', true);
lin = otslm.simple.linear(sz, 100, 'gpuArray', true);
ap = otslm.simple.aperture(sz, 512, 'gpuArray', true);

% Combine patterns and finalize
pattern(ap) = lin(ap);
pattern = otslm.tools.finalize(pattern);
```

To copy the final pattern back from the GPU we can use the `gather` function. The result is shown in Fig. 3.21.

```
pattern = gather(pattern);
imagesc(pattern);
```

Creating complex textures

The GPU often has significantly less memory than the main computer. This means that methods like *`otslm.tools.combine()`* become memory limited sooner. In order to work around this, it is sometimes possible to implement a version which calculates each pattern, adds it to the total array and re-uses the same memory to calculate the next pattern. The *`otslm.tools.lensesAndPrisms()`* function implements the Prisms and Lenses algorithm without needing to generate all the patterns before combining.

```
xyz = randn(3, num_points);
pattern = otslm.tools.lensesAndPrisms(sz, xyz, 'gpuArray', true);
```

Using a GeForce GTX 1060 GPU to run the Prisms and Lenses algorithm produces a order of magnitude decrease in run-time for multiple traps compared to a i7-8750H CPU, as shown in Fig. 3.22.

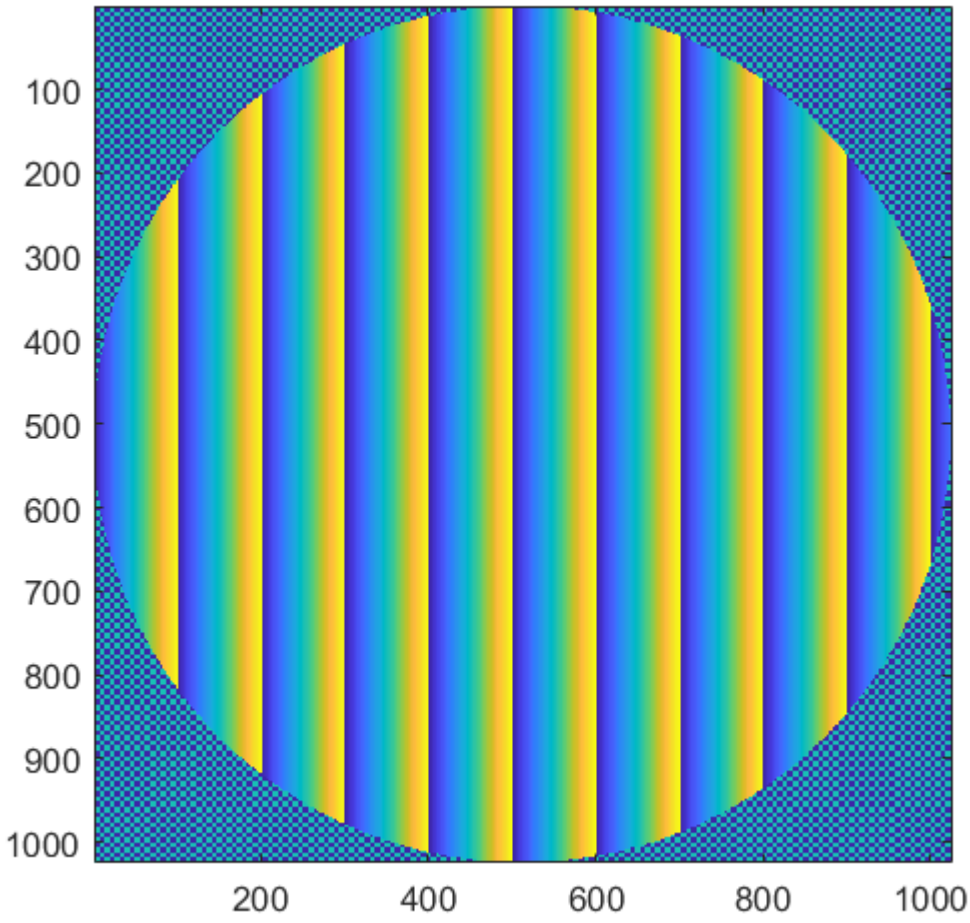

Fig. 3.21: Example of a pattern generated with the GPU

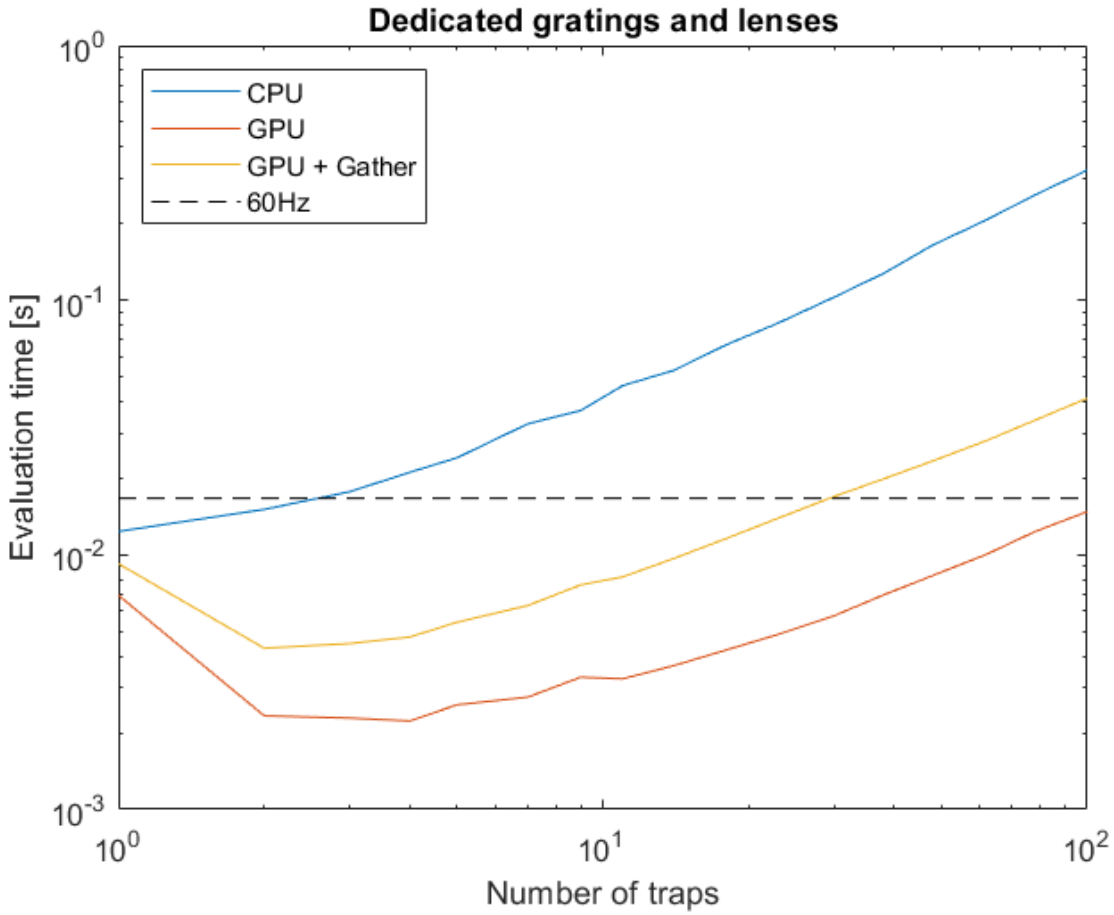

Fig. 3.22: Comparison of hologram generation time using CPU and GPU with different numbers of traps. For reference, a line is marked corresponding to the 60Hz refresh rate of a moderately fast SLM.

Using iterative algorithms

Iterative algorithms can use GPU arrays if either the target or guess are gpuArrays or if the iterative method is constructed using the named parameter `'gpuArray', true`. Not all methods support using the GPU at this stage, for instance, Bowman2017 has not been modified to support the GPU. The iterative methods have not been optimised and they currently involve a lot of copy/matrix resizing operations which will probably slow down optimisation. We aim to address these limitations in future versions.

```
sz = [512, 512];
im = otslm.simple.aperture(sz, sz(1)/20, 'value', [0, 1], 'gpuArray', true);
gs = otslm.iter.GerchbergSaxton(im, 'adaptive', 1.0, 'objective', []);
pattern = gs.run(600, 'show_progress', false);
```

3.4.2 Uploading a shader to the GPU

For uploading OpenGL shaders to the GPU, we provide an interface to RedTweezers. RedTweezers operates as a UDP server that runs independently from Matlab, this means it can run on any computer with OpenGL capabilities connected to your network (with appropriate firewall permission). Images, shaders and other data can be sent to RedTweezers via UDP, the RedTweezers server deals with uploading the shader and managing the shaders memory. RedTweezers interfaces are located in *otslm.utils.RedTweezers*.

Installing RedTweezers

To use RedTweezers, you will need to download the executable and have it running on a computer that is accessible on your network. RedTweezers can be downloaded from the computer physics communications program summaries page. Once downloaded, unzip the file (on windows you can use a program such as 7-zip to extract the files from the `.tar.gz` archive). Once unzipped, run either the `hologram_engine_64.exe` (or `hologram_engine.exe` for the 32-bit version). On the first run you may need to allow access to your network. If everything worked correctly, a new window with the RedTweezers splash screen should be displayed, shown in Fig. 3.23.

Displaying a image with RedTweezers

Displaying images isn't the intended purpose of RedTweezers, however by loading a shader which simply draws a texture to the screen we can implement a *ScreenDevice*-like interface using RedTweezers. This is implemented by *otslm.utils.RedTweezers.Showable*. This class inherits from *otslm.utils.Showable* (in addition to the *RedTweezers* base class) and provides all the same functionality of a *ScreenDevice* object. By default the object is configured to connect to UDP port `127.0.0.1:61557` and display an amplitude pattern. We can change the port and pattern type using the optional arguments.

```
rt = otslm.utils.RedTweezers.Showable('pattern_type', 'phase');
rt.window= [100, 200, 512, 512];   % Window size [x, y, width, height]
rt.show(otslm.simple.linear([200, 200], 20));
```

The main difference between *ScreenDevice* and *Showable* is the size of the pattern and the size/position of the window. *ScreenDevice* requires the pattern size to match the size of the window. For *Showable*, the pattern is stretched to fill the window. A further limitation is the maximum packet size RedTweezers supports only allows images of approximately 400x400 pixels (RedTweezers isn't intended for displaying images).

Using the RedTweezers Prisms and Lenses

otslm.utils.RedTweezers.PrismsAndLenses implements the Prisms and Lenses algorithm described in the RedTweezers paper (and implemented in the LabView code supplied with RedTweezers). To use the Prisms

Fig. 3.23: Red tweezers splash screen.

and Lenses implementation, start by creating a new instance of the object and configure the window and any other RedTweezers properties.

```
rt = otslm.utils.RedTweezers.PrismsAndLenses();
rt.window= [100, 200, 512, 512];   % Window size [x, y, width, height]
```

Then we need to configure the shader properties. These are not set by default since they may already be set by another program.

```
rt.focal_length = 4.5e6;        % Focal length [microns]
rt.wavenumber = 2*pi/1.064;     % Wavenumber [1/microns]
rt.size = [10.2e6, 10.2e6];     % SLM size [microns]
rt.centre = [0.5, 0.5];
rt.total_intensity = 0.0;    % 0.0 to disable
rt.blazing = linspace(0.0, 1.0, 32);
rt.zernike = zeros(1, 12);
```

This should create a blank hologram. To add spots to this hologram use the *addSpot()* method. For example, to add a spot to diffract light to a particular coordinate in the focal plane, use:

```
rt.addSpot('position', [60, 54, 7])
rt.addSpot('position', [-20, 10, -3])
rt.addSpot('position', [40, -37, 0])
```

If we have more than 50 spots we need to send the spot data as a GLSL texture. The class automatically handles this. If we want to always use a texture, we can set

```
rt.use_texture = true;
```

Creating custom RedTweezers shaders

To create a custom GLSL shader and load it using RedTweezers simply inherit from the *otslm.utils.RedTweezers.RedTweezers* class, load the GLSL shader source using the *sendShader()*, and use *sendUniform()* and *sendTexture()* to send data to the shader. For inspiration, look at the *Showable* and *PrismsAndLenses* implementations.

3.5 Accessing OTSLM from LabVIEW

It is possible to use functionality from OTSLM in LabVIEW. This can be useful for implementing user interfaces which can be easily customised or for integrating OTSLM with existing code.

To run MATLAB scripts from LabVIEW, you will need to use MathScript or Interface for MATLAB depending on the version of LabVIEW you are using. We have provided an example package using LabVIEW NXG 3.1 (using LabVIEW Interface for MATLAB) in the `examples/labview` folder of the toolbox.

This section provides an overview of the LabVIEW example package and the PrismsAndLenses example LabVIEW application. The example package only provides the features needed for the PrismsAndLenese example. We welcome contributions from LabVIEW users to improve this package to provide better coverage of the OTSLM functionality.

Contents

- *Calling a function with a cell array*
- *Creating an `otslm` class interface*
- *Building an application*

3.5.1 Creating an `otslm.simple` function interface

LabVIEW Interface for MATLAB (LIFM) provides a system for defining different interfaces to matlab functions. Using LIFM, you can specify the input parameter names and types in order to create an object that can be imported into a VI. The current version of LabVIEW Interface for MATLAB doesn't provide a good method for dealing directly with string constants or named parameters, instead it is better to create a separate VI which wraps the LIFM interface.

In this section we take you through defining a interface for the `otslm.simple.lienar()` function. This function takes two required inputs (the size of the pattern and spacing of the grating) and outputs a 2D array of doubles for the pattern. The complete example can be found in `otslm/examples/labview/OtslmMatlabInterface/otslm.gcomp/simple/linear.mli` and `otslm/examples/labview/OtslmMatlabInterface/otslm.gcomp/simple/linear.gvi`.

Start by creating a new Interface for Matlab, go to: **File > New > Interface for MATLAB**. In the box marked *Select a MATLAB program file or enter a MATLAB function name* enter `otslm.simple.linear` and click **Add interface node**. Click **Add parameter** five times and name the parameters `im`, `sz`, `spacing`, `centre_str` and `centre` as shown in Fig. 3.24.

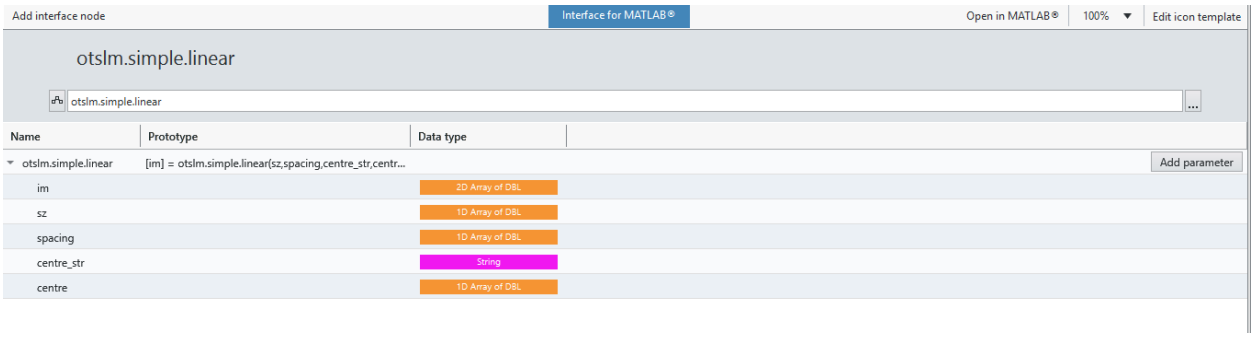

Fig. 3.24: Setting up the parameters for the Matlab interface to *`otslm.simple.linear()`*.

To change the data types and the input/output mode for the parameters, click on the parameter and change the corresponding settings in the Item panel, as shown in Fig. 3.25.

In order to use this interface the `OTSLM` path must be added to the Matlab path. You can do this either by adding the OTSLM path in the `Startup.m` script, as described on the getting started page, or you can run a script which adds OTSLM to the Matlab path. In the example package, we run a script to add OTSLM to the path. The `initOtslm.m` script, located in the `examples/labview` directory contains the following code:

```
function initOtslm()

  fname = mfilename('fullpath');
  [fpath, ~, ~] = fileparts(fname);
  fparts = split(fpath, filesep);

  % Add current path
  addpath(fpath);
```

(continues on next page)

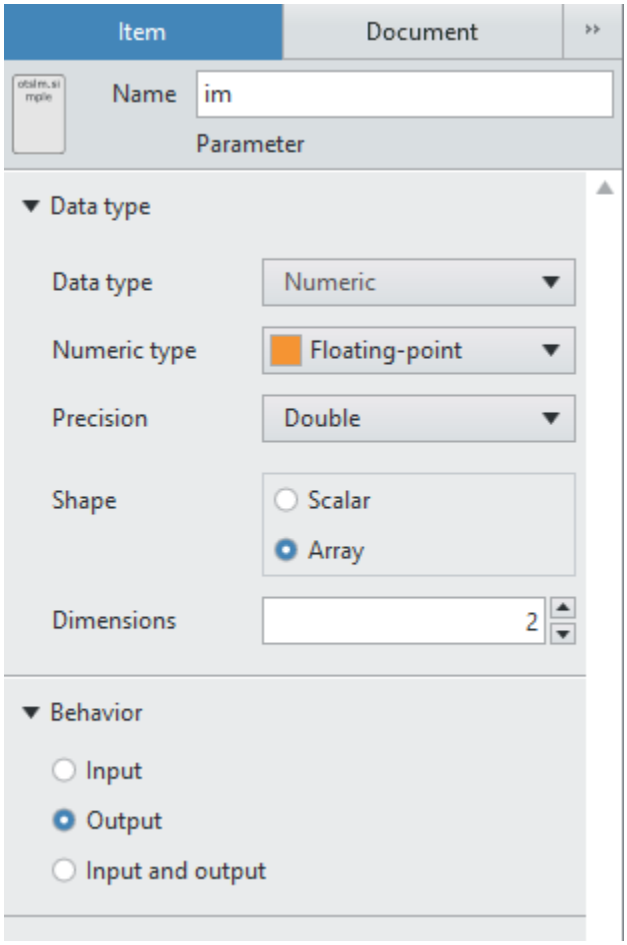

Fig. 3.25: Screenshot of the item configuration panel.

(continued from previous page)

```
  % Add toolbox path
  toolbox_path = fullfile(fparts{1:end-2});
  addpath(toolbox_path);

end
```

The script first finds the path for the mfile, adds the `examples/labview` directory to the path and adds the relative path for the `otslm` directory to the path. To call this script, you will need to create another Matlab interface and specify the file path to this script. The interface for this script doesn't need any parameters. This script should be run at the start of each LabVIEW session or at the start of each LabVIEW application.

The Matlab interface created for *`otslm.simple.linear()`* can now be included in LabVIEW applications or VIs. To use the interface, you must connect values to each of the input and output parameters and optionally the input/output error connectors. However, most of the time you will not need to change all of the parameters, for instance, the `centre_str` parameter will always be the string `'centre'`. To simplify the interface and allow customisation of the icon we can create a wrapper VI for the Matlab interface. To create a new VI, click **File > New > VI**. Add the Matlab Interface VI you just created to the centre of the diagram and connect nodes to the terminals as shown in Fig. 3.26.

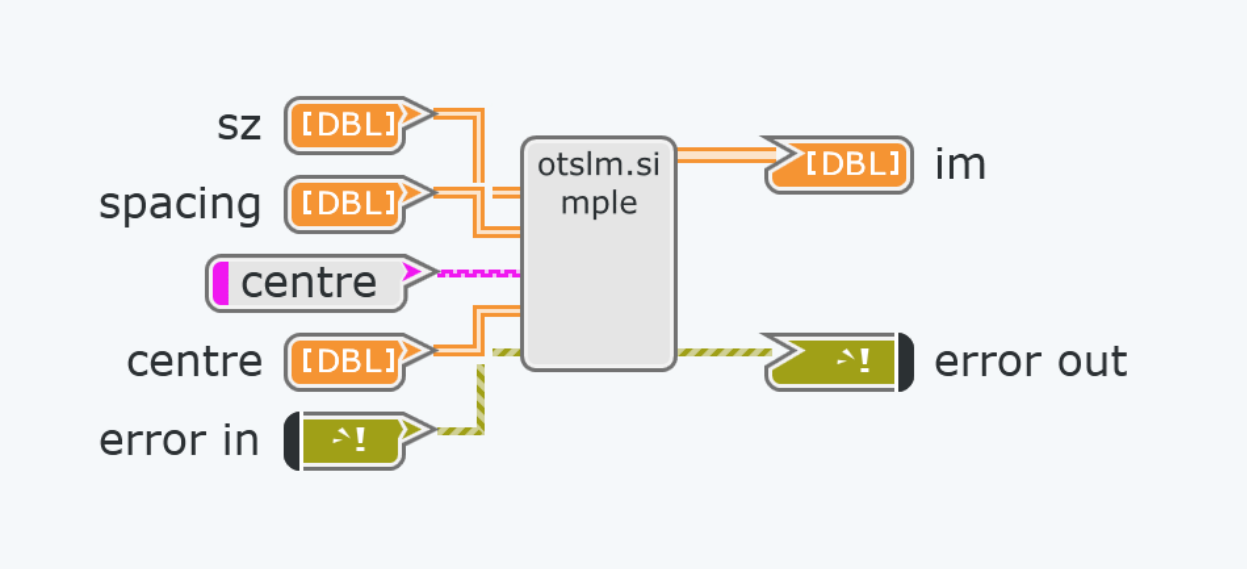

Fig. 3.26: Wrapper for LabVIEW interface for matlab.

This interface can be further improved, for instance, making the centre array optional and checking the length of the array is correct. For example code, see `linear.gvi`.

Once the diagram has been configured, you can create a front panel to test the interface and configure the icon.

3.5.2 Calling a function with a cell array

The *`otslm.tools.combine()`* function takes as input a cell array of patterns to combine and returns a single pattern as the result. LabVIEW doesn't currently provide a mechanism for calling a function with a cell array, however we can work around this by writing a wrapper function which takes a 3D array of images and converts them to a cell array of 2D images. The `unpackCombine.m` function in `examples/labview` does exactly this:

```
function varargout = unpackCombine(input3, varargin)

  input = mat2cell(input3, size(input3, 1), size(input3, 2), ...
    ones(1, size(input3, 3)));

  input = squeeze(input);

  [varargout{1:nargout}] = otslm.tools.combine(input, varargin{:});

end
```

It is now possible to create an LabVIEW Interface for Matlab using this function as described in the previous section.

3.5.3 Creating an `otslm` class interface

In order to use OTSLM classes, such as *otslm.utils.ScreenDevice* we need to construct and instance of the object, call its methods and clean up the instance once we are done. LabVIEW only supports creating function and script interfaces for Matlab. In order to work around this, we can write a dispatch method which creates the class instance and handles calls to the function methods. The following is an example of a dispatch method:

```
function varargout = callClassMethod(varname, classname, methodname, varargin)

assert(~isempty(varname), 'varname must be supplied');

tmpvarname = 'ourargs';

if isempty(methodname) && ~isempty(classname)

  % Create a new instance of the class
  assignin('base', tmpvarname, varargin);
  evalin('base', [varname, ' = ', classname, '(', tmpvarname, '{:});']);

elseif isempty(classname) && ~isempty(methodname)

  % Call a class method
  assignin('base', tmpvarname, varargin);
  [varargout{1:nargout}] = evalin('base', [varname, '.', methodname, '(', tmpvarname,
↪'{:});']);

else
  error('Only classname or methodname must be supplied');

end
```

This function places the Matlab class instance in the base workspace, we keep track of the class instance using a string (`varname`) in LabVIEW. To use this dispatch method, we need to create a LabVIEW Interface for MATLAB for the class and add each class method we wish to use, including the constructor and destructor. For *ScreenDevice*, the interface might look something like the one shown in Fig. 3.27.

We can then implement a wrapper VI for each of these methods as described in the previous sections. The `classname` and `methodname` arguments specify the constructor name and the class method name to be called. For the destructor, use the string `'delete'` for the method name. In order to use this interface, we need to keep track of the class instance name and make sure we construct and delete the object before using other methods of the class. For example usage, see *Building an application*.

Fig. 3.27: An interface example for a Matlab class using the `callClassMethod` dispatch function.

3.5.4 Building an application

This section describes building a LabVIEW application for generating a Prisms and Lenses hologram which is drawn using *`ScreenDevice`*. You can find the finished application in `examples/labview/OtslmMatlabInterface/PrismsAndLenses.gcomp`. This example assumes you have followed the above instructions to implement your own VIs for the `spherical`, `linear`, `combine` and `ScreenDevice` OTSLM functions/classes or you are using the examples provided in the `examples/labview/OtslmMatlabInterface/otslm.gcomp` package. If you use the example application/package, you will need to modify the path in `otslm.gcomp/initOtslm.mli` to find the correct path for the `initOtslm.m` file.

Create a new application in LabVIEW by going to **File > New > Application**. Name the application. Add a new VI to the application for the front panel (where the main user interface will be displayed): right click on the application icon in the project browser and click: **New > VI**, as shown in Fig. 3.28.

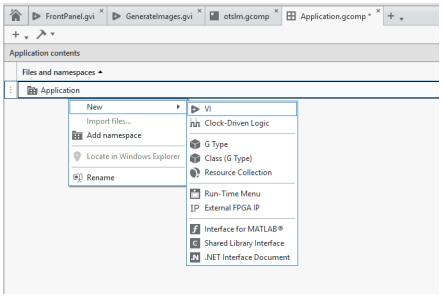

Fig. 3.28: Adding a new VI to an application.

Create the VI by adding the controls shown in Fig. 3.29.

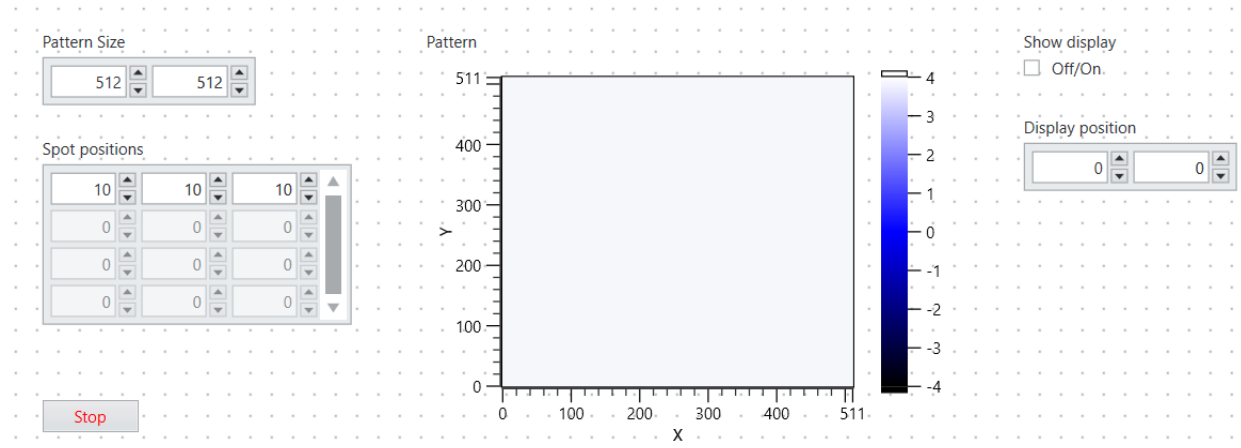

Fig. 3.29: Layout of front panel.

The user interface will allow the user to specify the size and position of the window on the screen, change the number and location of spots in the Prisms and Lenses algorithm, and see a preview of what the image will look like on the screen.

To implement this, we need to initialise OTSLM, construct the screen device object for displaying the patterns, generate the array of patterns to pass to `otslm.tools.combine` for each spot the user requests, and display the result in the previous and on the screen.

To generate the array of patterns for each prisms and lenses spot, we will create a sub-vi which takes as input the pattern size and spot locations and generates a 3D array of patterns which we can pass to combine. Add a new vi to your application and configure it with the nodes shown in Fig. 3.30.

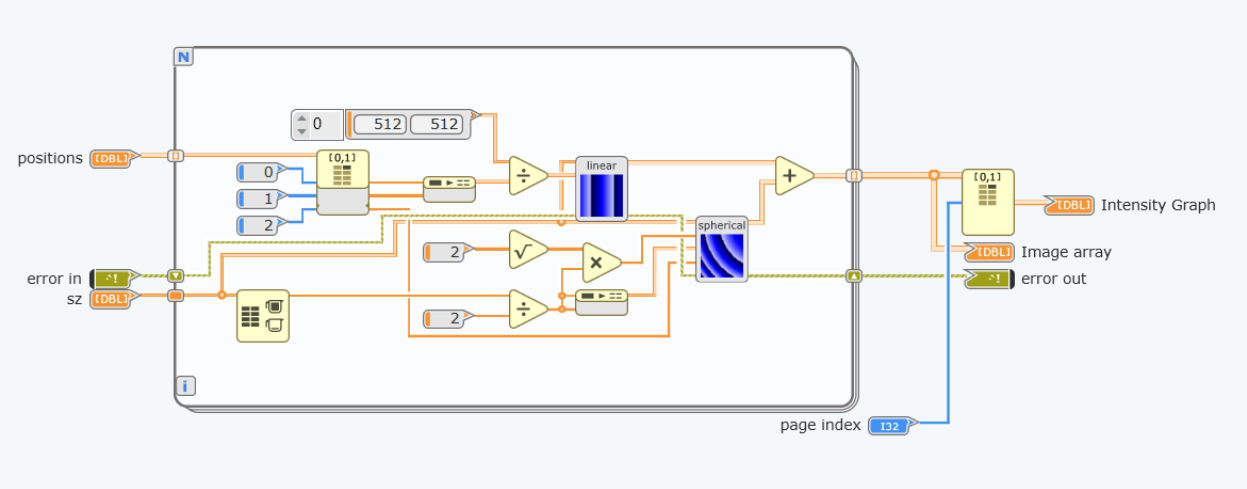

Fig. 3.30: Layout of generate images diagram.

To add the **spherical** and **lenses** sub-vis, either click and drag the VIs from the project file tree or add them from the **Project Items** menu, as shown in Fig. 3.31.

Connect the input and output nodes in the icon diagram as shown in Fig. 3.32.

Next, switch back to the front panel diagram and construct the program shown in Fig. 3.33.

In this example we use a loop to continuously update the display when the user changes inputs to the VI. The Screen-

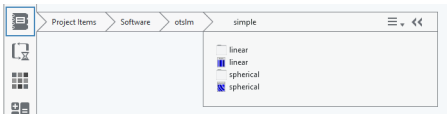

Fig. 3.31: Using the project items menu.

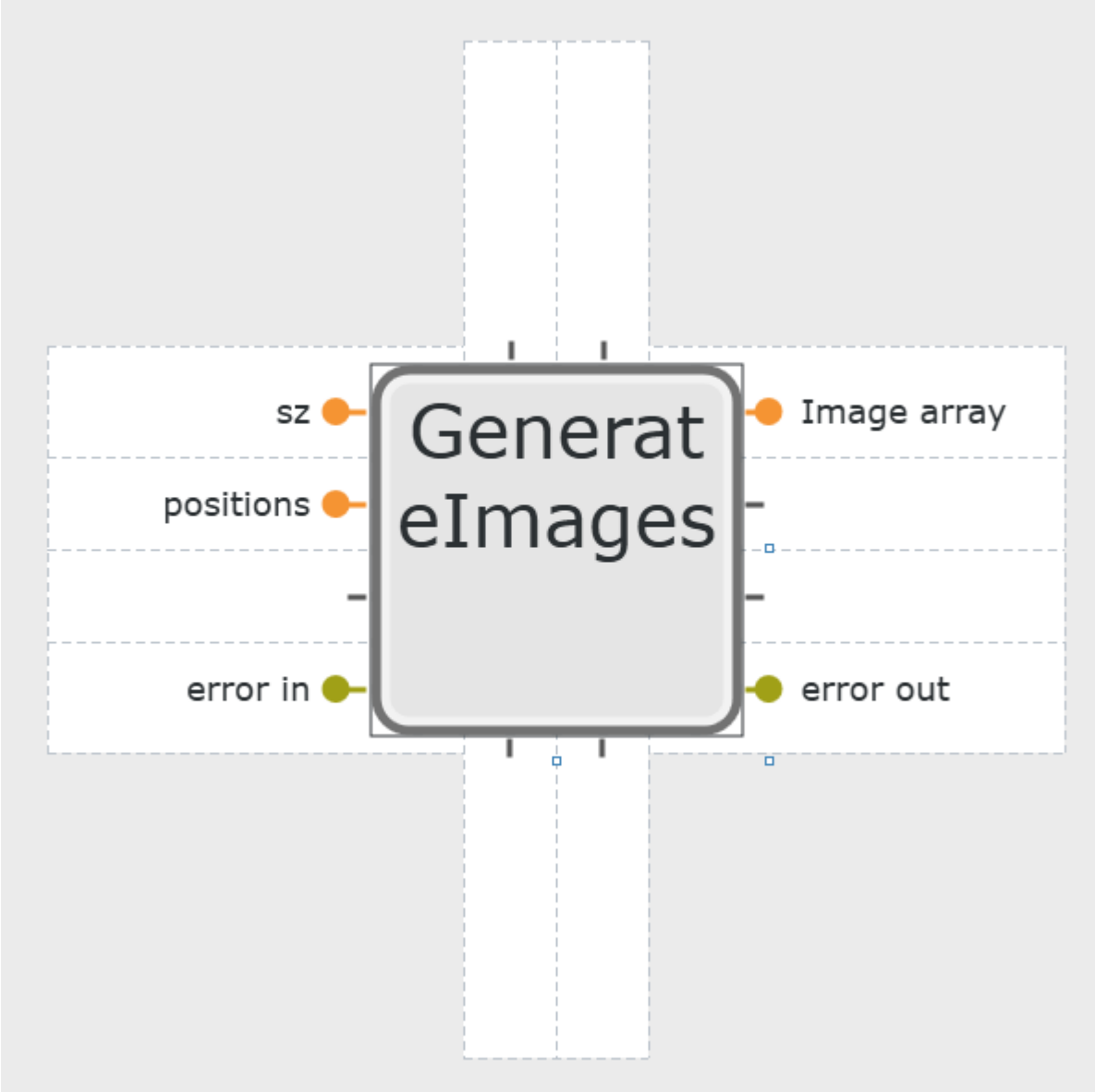

Fig. 3.32: Layout of generate images icon.

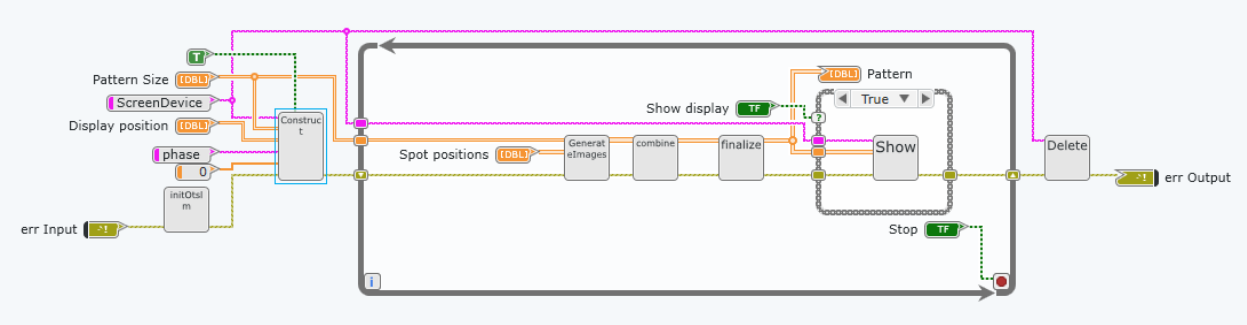

Fig. 3.33: Layout of the front panel diagram.

Device is positioned and constructed outside the loop, this means that the size of the pattern and location are fixed throughout the entire run of the program. If the show display checkbox is not clicked, the `ScreenDevice` is asked to close, otherwise the pattern is displayed to the screen.

CHAPTER 4

Packages

The toolbox is split into the following packages:

- Simple includes simple procedural functions for generating phase and amplitude patterns.
- Iter includes iterative methods for generating patterns based on a target beam.
- Tools provides tools for combining beams and visualising the output.
- Utils provides functions not necessarily related to pattern generation but things our group has found useful for displaying patterns.
- Ui contains graphical user interfaces for most of the functionality in the toolbox. These user interfaces are useful for quickly exploring the functionality of the toolbox.

4.1 *simple* Package

This page contains a description of the functions contained in the `otslm.simple` package. These functions typically have analytic expressions and the functionality can be implemented in just a few lines of code. The implementation in the toolbox contains additional inputs to help with things like centring the patterns or generating the grids.

Most of these functions take as input the size of the image to generate, a two or three element vector with the width and height of the device; and parameters specific to the method. They produce one or more matlab matrices with the specified size. For example, a checkerboard image with 100 rows and 50 columns could be created with:

```
rows = 100;
cols = 50;
sz = [rows, cols];
im = otslm.simple.checkerboard(sz);
imagesc(im);
disp(size(im));
```

The functions have been grouped into categories: *Lens functions*, *Beams*, *Gratings*, *3-D functions* and *Miscellaneous*. This is a very general and non-unique grouping. The output of many of these functions can be placed directly on a

spatial light modulator as a phase or amplitude masks, or output of multiple functions can be combined using functions in the *tools Package* or Matlab operations on arrays (e.g., array addition or logical indexing).

Contents

4.1.1 Lens functions

These functions produce a single array. These arrays can be used to describe the phase functions of different lenses. Most of these functions support 1-D or 2-D variants, for instance, the spherical function can be used to create a cylindrical or spherical lens.

Functions

aspheric

`otslm.simple.`**`aspheric`**(*sz*, *radius*, *kappa*, *varargin*)

Generates a aspherical lens. The equation describing the lens is

$$z(r) = \frac{r^2}{R(1+\sqrt{1-(1+\kappa)r^2/R^2})} + \sum_{i=2}^{N} \alpha_i r^{2i} + \delta$$

where R is the radius of the lens, κ determines if the lens shape:

- `<-1` – hyperbola
- `-1` – parabola
- `(-1, 0)` – ellipse (surface is a prolate spheroid)
- `0` – sphere
- `> 0` – ellipse (surface is an oblate spheroid)

and the α's corresponds to higher order corrections and δ is a constant offset.

Usage pattern = aspheric(sz, radius, kappa, ...) generates a aspheric lens described by radius and conic constant centred in the image.

Parameters

- sz – size of the pattern `[rows, cols]`
- radius – Radius of the lens R
- kappa – conic constant κ

Optional named parameters

- 'alpha' [a1, ...] – additional parabolic correction terms
- 'delta' offset – offset for the final pattern (default: 0.0)
- 'scale' scale – scaling value for the final pattern
- 'background' img – Specifies a background pattern to use for values outside the lens. Can be a matrix; a scalar, in which case all values are replaced by this value; or a string with 'random' or 'checkerboard' for these patterns.
- 'centre' [x, y] – centre location for lens (default: sz/2)
- 'offset' [x, y] – offset after applying transformations
- 'type' type – is the lens cylindrical or spherical (1d or 2d)
- 'aspect' aspect – aspect ratio of lens (default: 1.0)
- 'angle' angle – Rotation angle about axis (radians)
- 'angle_deg' angle – Rotation angle about axis (degrees)
- 'gpuArray' bool – If the result should be a gpuArray

axicon

otslm.simple.**axicon**(*sz*, *gradient*, *varargin*)

Generates a axicon lens. The equation describing the lens is

$$z(r) = -G|r|$$

where G is the gradient of the lens.

Usage pattern = axicon(sz, gradient, ...)

Parameters

- sz – size of the pattern `[rows, cols]`
- gradient – gradient of the lens G

Optional named parameters

- 'centre' [x, y] – centre location for lens (default: sz/2)
- 'offset' [x, y] – offset after applying transformations
- 'type' type – is the lens cylindrical or spherical (1d or 2d)
- 'aspect' aspect – aspect ratio of lens (default: 1.0)
- 'angle' angle – Rotation angle about axis (radians)
- 'angle_deg' angle – Rotation angle about axis (degrees)

- 'gpuArray' bool – If the result should be a gpuArray

Example (see also Fig. 4.1):

```
sz = [128, 128];
gradient = 0.1;
im = otslm.simple.axicon(sz, gradient);
```

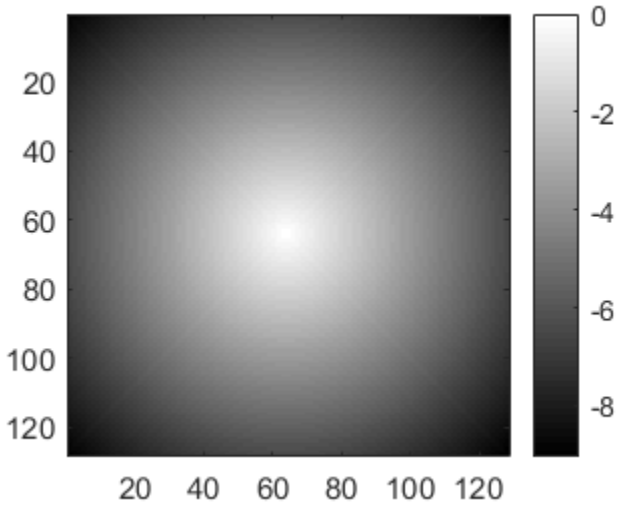

Fig. 4.1: Example of a axicon lens.

cubic

otslm.simple.**cubic**(*sz*, *varargin*)

Generates cubic phase pattern for Airy beams. The phase pattern is given by

$$f(x, y) = (x^3 + y^3)s^3$$

where s is a scaling factor.

Usage pattern = cubic(sz, ...) generates a cubic pattern according to

Parameters

- sz (size) – size of the pattern `[rows, cols]`

Optional named parameters

- scale (numeric) – Scaling factor for pattern.
- centre (numeric) – Centre location for lens (default: sz/2)
- offset (numeric) – Offset after applying transformations `[x,y]`
- type (enum) – Cylindrical `1d` or spherical `2d`
- aspect (numeric) – aspect ratio of lens (default: 1.0)
- angle (numeric) – Rotation angle about axis (radians)
- angle_deg (numeric) – Rotation angle about axis (degrees)
- gpuArray (logical) – If the result should be a gpuArray

Example (see also Fig. 4.2):

```
sz = [128, 128];
im = otslm.simple.cubic(sz);
```

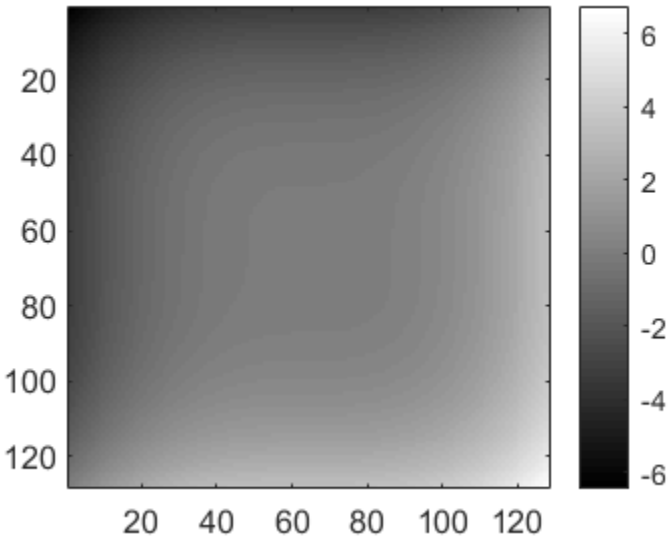

Fig. 4.2: Example of a cubic function.

spherical

otslm.simple.**spherical** (*sz*, *radius*, *varargin*)

Generates a spherical lens pattern. The equation describing the lens is

$$z(r) = \frac{R}{|R|}\frac{A}{r}\sqrt{R^2 - r^2}$$

where A is a scaling factor and R is the lens radius. Imaginary values are undefined and can be replaced by another value.

Usage pattern = spherical(sz, radius, ...) generates a spherical pattern with values from 0 (at the edge) and 1*sign(radius) (at the centre).

Parameters

- sz – size of the lens
- radius – radius of the lens R

Optional named arguments

- 'delta' offset – offset for pattern (default: -sign(radius))
- 'scale' scale – scaling value for the final pattern
- 'background' img – Specifies a background pattern to use for values outside the lens. Can also be a scalar, in which case all values are replaced by this value; or a string with 'random' or 'checkerboard' for these patterns.
- 'centre' [x, y] – centre location for lens

- 'offset' [x, y] – offset after applying transformations
- 'type' type – is the lens cylindrical or spherical (1d or 2d)
- 'aspect' aspect – aspect ratio of lens (default: 1.0)
- 'angle' angle – Rotation angle about axis (radians)
- 'angle_deg' angle – Rotation angle about axis (degrees)
- 'gpuArray' bool – If the result should be a gpuArray

See also *aspheric()*.

The following example creates a spherical lens with radius 128 pixels, as shown in Fig. 4.3. The lens is centred in the pattern and a checkerboard pattern is used for values outside the lens.

```
sz = [256, 256];
radius = 128;
background = otslm.simple.checkerboard(sz);
im = otslm.simple.spherical(sz, radius, 'background', background);
```

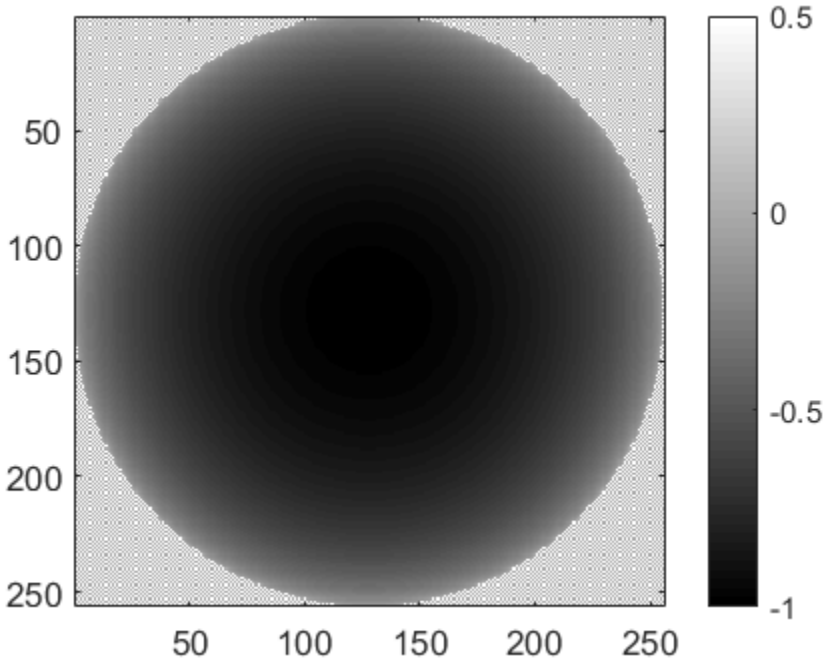

Fig. 4.3: Example of a spherical lens.

parabolic

otslm.simple.**parabolic**(*sz*, *alphas*, *varargin*)

Generates a parabolic lens pattern. The equation describing this lens is

$$z(r) = \alpha_1 * r^2 + \alpha_2 * r^4 + \alpha_3 * r^6 + ...$$

where α_n are the polynomial coefficients.

Usage pattern = parabolic(sz, alphas, ...) generates a parabolic lens.

Parameters

- sz (size) – size of pattern `[rows, cols]`
- alphas – array of polynomial coefficients α_n

The default centre for the lens is the centre of the pattern, this can be modified with named parameters.

See also *aspheric()* for more information and named parameters.

gaussian

otslm.simple.**gaussian**(*sz*, *sigma*, *varargin*)

Generates a Gaussian pattern. A Gaussian pattern can be used as a lens or as the intensity profile for the incident illumination. The equation describing the pattern is

$$z(r) = A \exp -r^2/(2\sigma^2)$$

where A is a scaling factor and σ is the radius of the Gaussian.

Usage pattern = gaussian(sz, sigma, ...)

Parameters

- sz (numeric) – size of the pattern `[rows, cols]`
- sigma (numeric) – radius of the Gaussian σ

Optional named parameters

- 'scale' (numeric) – scaling value A (default: 1).
- 'centre' [x, y] – centre location for lens (default: sz/2)
- 'offset' [x, y] – offset after applying transformations
- 'type' type – is the lens cylindrical or spherical (1d or 2d)
- 'aspect' aspect – aspect ratio of lens (default: 1.0)
- 'angle' angle – Rotation angle about axis (radians)
- 'angle_deg' angle – Rotation angle about axis (degrees)
- 'gpuArray' bool – If the result should be a gpuArray

Example usage (see also Fig. 4.4):

```
sz = [128, 128];
sigma = 64;
im = otslm.simple.gaussian(sz, sigma, 'scale', 2.0);
imagesc(im);
```

4.1.2 Beams

These functions can be used to calculate the amplitude and phase patterns for different kinds of beams. To generate these kinds of beams, and other arbitrary beams, both the amplitude and phase of the beam needs to be controlled. This can be achieved by generating a phase or amplitude pattern which combines the phase and amplitude patterns produced by these functions, for details see the *Advanced Beams* example and *otslm.tools.finalize()*.

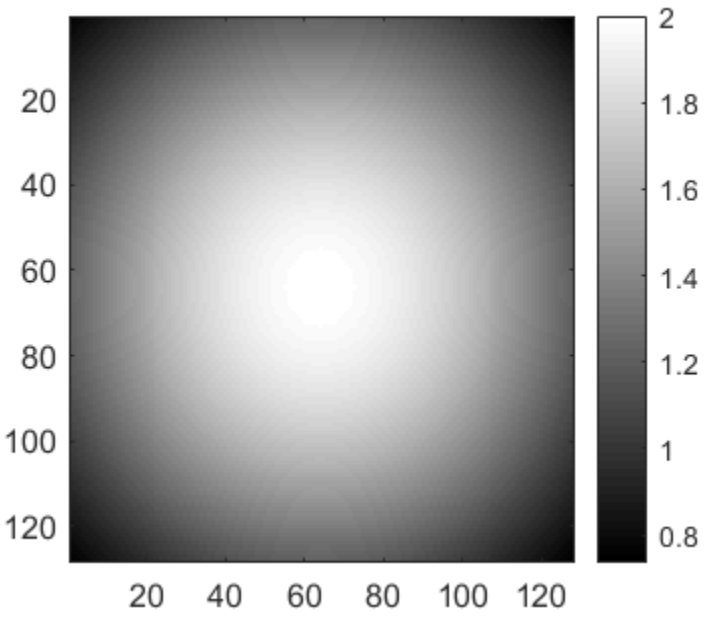

Fig. 4.4: Example output from *gaussian()*.

Functions

- *bessel*
- *hgmode*
- *lgmode*
- *igmode*

bessel

otslm.simple.**bessel**(*sz*, *mode*, *varargin*)

Generates the phase and amplitude patterns for Bessel beams

Usage pattern = bessel(sz, mode, ...) generates the phase pattern for a particular order Bessel beam.

[phase, amplitude] = bessel(...) also calculates the signed amplitude of the pattern in addition to the phase.

Parameters

- sz – size of the pattern `[rows, cols]`
- mode (integer) – bessel function mode

Optional named parameters:

- 'scale' scale – radial scaling factor for pattern
- 'centre' [x, y] – centre location for lens (default: sz/2)
- 'offset' [x, y] – offset after applying transformations

- 'type' type – is the lens cylindrical or spherical (1d or 2d)
- 'aspect' aspect – aspect ratio of lens (default: 1.0)
- 'angle' angle – Rotation angle about axis (radians)
- 'angle_deg' angle – Rotation angle about axis (degrees)
- 'gpuArray' bool – If the result should be a gpuArray

hgmode

Hermite-Gaussian (HG) beams are solutions to the paraxial wave equation in Cartesian coordinates. Beams are described by two mode indices.

`otslm.simple.`**`hgmode`**(*sz*, *xmode*, *ymode*, *varargin*)

Generates the phase pattern for a HG beam

Usage pattern = hgmode(sz, xmode, ymode, ...) generates the phase pattern with x and y mode numbers.

[phase, amplitude] = hgmode(...) also calculates the signed amplitude of the pattern in addition to the phase.

Parameters

- sz – size of the pattern
- xmode – HG mode order in the x-direction
- ymode – HG mode order in the y-direction

Optional named parameters

- 'scale' scale – scaling factor for pattern
- 'centre' [x, y] – centre location for lens (default: sz/2)
- 'offset' [x, y] – offset after applying transformations
- 'aspect' aspect – aspect ratio of lens (default: 1.0)
- 'angle' angle – Rotation angle about axis (radians)
- 'angle_deg' angle – Rotation angle about axis (degrees)
- 'gpuArray' bool – If the result should be a gpuArray

lgmode

Laguerre-Gaussian (LG) beams are solutions to the paraxial wave equation in cylindrical coordinates.

`otslm.simple.`**`lgmode`**(*sz*, *amode*, *rmode*, *varargin*)

Generates the phase pattern for a LG beam

Usage pattern = lgmode(sz, amode, rmode, ...) generates phase pattern.

[phase, amplitude] = lgmode(...) also generates the amplitude pattern.

Parameters

- sz – size of the pattern
- amode – azimuthal order
- rmode – radial order

Optional named parameters

- 'radius' radius – scaling factor for radial mode rings
- 'p0' p0 – incident amplitude correction factor Should be 1.0 (default) for plane wave illumination (w_i = Inf), for Gaussian beams should be p0 = 1 - radius^2/w_i^2. See Lerner et al. (2012) for details.
- 'centre' [x, y] – centre location for lens (default: sz/2)
- 'offset' [x, y] – offset after applying transformations
- 'aspect' aspect – aspect ratio of lens (default: 1.0)
- 'angle' angle – Rotation angle about axis (radians)
- 'angle_deg' angle – Rotation angle about axis (degrees)
- 'gpuArray' bool – If the result should be a gpuArray

In order to generate pure LG beams it is necessary to control both the beam amplitude and phase. However, if the purity of the beam is not important then the phase pattern is often sufficient to generate the desired beam shape.

igmode

Ince-Gaussian (IG) beams are solutions to the paraxial wave equation in elliptical coordinates. The IG modes for a complete basis in elliptic coordinates. When the ellipticity parameter is infinite, IG beams are equivalent to HG beams, and when the ellipticity approaches 0, IG beams are equivalent to LG beams.

`otslm.simple.`**`igmode`**(*sz*, *even*, *modep*, *modem*, *elipticity*, *varargin*)

Generates phase and amplitude patterns for Ince-Gaussian beams

Ince-Gaussian beams are described in Bandres and Gutirrez-Vega (2004).

> **Warning:** This function is a work-in-progress and may not produce clean output.

Usage pattern = igmode(sz, even, modep, modem, elipticity, ...)

[phase, amplitude] = igmode(...) also calculates the signed amplitude of the pattern in addition to the phase.

Parameters

- even – True for even parity
- modep – polynomial order p (integer: 0, 1, 2, ...)
- modem – polynomial degree m ($0 \leq m \leq p$)
- elipticity – elipticity of the coordinates.

Optional named parameters

- 'scale' scale – scaling factor for pattern
- 'centre' [x, y] – centre location for lens (default: sz/2)
- 'offset' [x, y] – offset after applying transformations
- 'aspect' aspect – aspect ratio of lens (default: 1.0)
- 'angle' angle – Rotation angle about axis (radians)

- 'angle_deg' angle – Rotation angle about axis (degrees)
- 'gpuArray' bool – If the result should be a gpuArray

This function uses code from Miguel Bandres, see source code for information about copyright/license/distribution.

4.1.3 Gratings

These functions can be used to create periodic patterns which can be used to create diffraction gratings.

Functions

- *linear*
- *sinusoid*

linear

`otslm.simple.`**`linear`**(*sz*, *spacing*, *varargin*)

Generates a linear gradient. The pattern is described by

$$f(x) = \frac{1}{D}x$$

where the gradient is $1/D$. For a periodic grating with maximum height of 1, D corresponds to the grating spacing.

Usage pattern = linear(sz, spacing, ...)

Parameters

- sz (numeric) – size of pattern `[rows, cols]`
- spacing – inverse gradient D

Optional named parameters

- 'centre' [x, y] – centre location for lens (default: `[1, 1]`)
- 'offset' [x, y] – offset after applying transformations
- 'aspect' aspect – aspect ratio of lens (default: 1.0)
- 'angle' angle – Rotation angle about axis (radians)
- 'angle_deg' angle – Rotation angle about axis (degrees)
- 'gpuArray' bool – If the result should be a gpuArray

To generate a linear grating (a saw-tooth grating) you need to take the modulo of this pattern. This is done by `otslm.tools.finalize()` but can also be done explicitly with:

```
sz = [40, 40];
spacing = 10;
im = mod(otslm.simple.linear(sz, spacing, 'angle_deg', 45), 1);
```

Fig. 4.5 shows example output.

Spacing can be a single number or two numbers for the spacing in the `x` and `y` directions. For an example of how *`linear()`* can be used to shift the beam focus, see *Gratings and Lens LiveScript*.

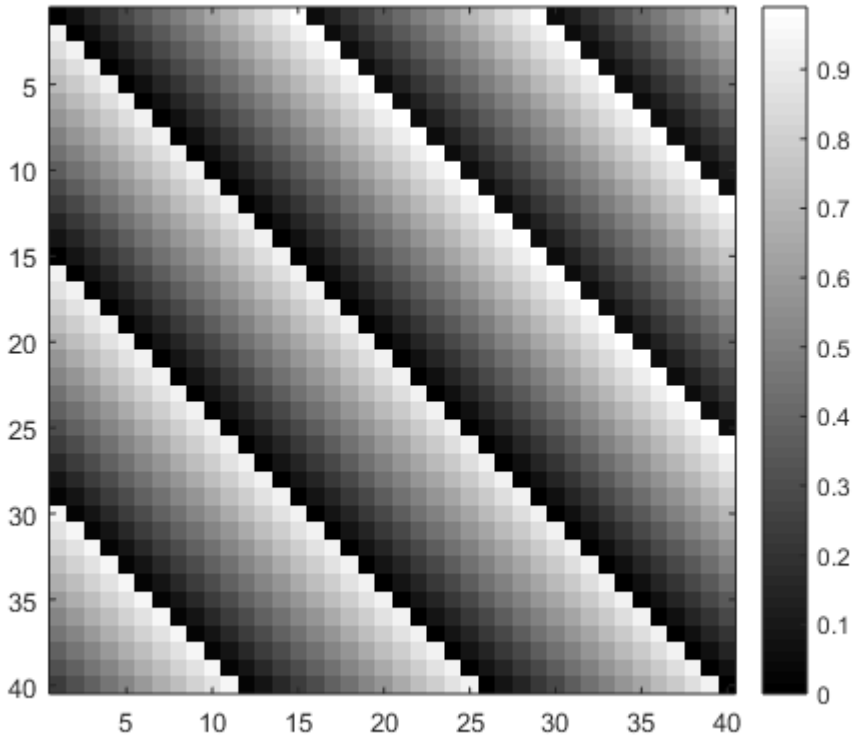

Fig. 4.5: Example output from `linear()`.

sinusoid

otslm.simple.**sinusoid**(*sz*, *period*, *varargin*)

Generates a sinusoidal grating. The function for the pattern is

$$f(x) = A\sin(2\pi x/P) + \delta$$

Where A is the scale, δ is the mean, and P is the period.

This function can create a one dimensional grating in polar (circular) coordinates, in linear coordinates, or a mixture of two orthogonal gratings, see the types parameters for information.

Usage pattern = sinusoid(sz, period, ...) generates a sinusoidal grating with the default scale of 0.5 and default mean of 0.5.

Parameters

- sz (numeric) – size of pattern `[rows, cols]`
- period (numeric) – period P

Optional named parameters

- 'scale' (numeric) – pattern scale A (default: 1)
- 'mean' (numeric) – offset for pattern δ (default: 0.5)
- 'type' (enum) – the type of sinusoid pattern to generate
- '1d' – one dimensional (default)
- '2d' – circular coordinates
- '2dcart' – multiple of two sinusoid functions at 90 degree angle supports two period values [Px, Py].
- 'centre' [x, y] – centre location for lens
- 'offset' [x, y] – offset after applying transformations
- 'aspect' aspect – aspect ratio of lens (default: 1.0)
- 'angle' angle – Rotation angle about axis (radians)
- 'angle_deg' angle – Rotation angle about axis (degrees)
- 'gpuArray' bool – If the result should be a gpuArray

Fig. 4.6 shows different types of sinusoid gratings supported by the function.

Example usage (see also Fig. 4.7):

```
sz = [40, 40];
period = 10;
im = sinusoid(sz, period);
```

4.1.4 Miscellaneous

Various functions for generating patterns not described in other sections. This includes the *grid()* and *aperture()* functions which are used to create the grids and masks used by other toolbox functions.

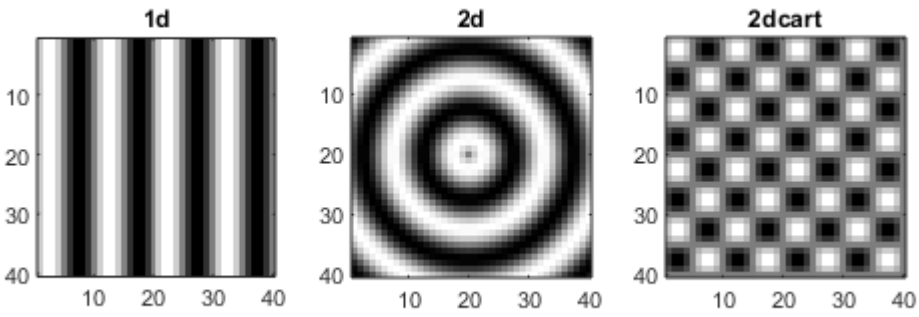

Fig. 4.6: Example of different sinusoid gratings generated using `sinusoid()`.

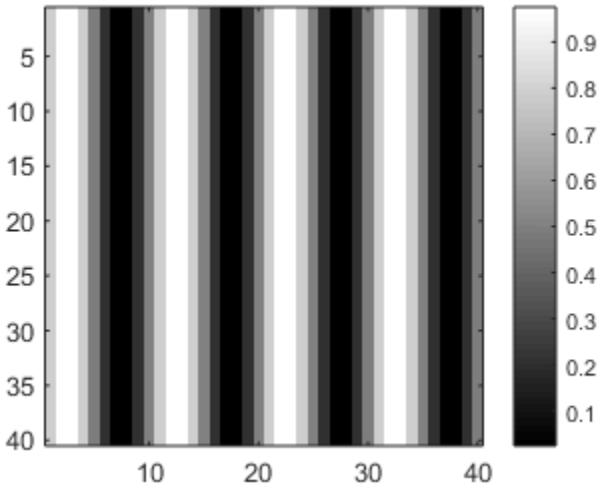

Fig. 4.7: Example output from `sinusoid()`.

Functions

aperture

otslm.simple.**aperture**(*sz*, *dimension*, *varargin*)

Generates different shaped aperture patterns/masks

Usage pattern = aperture(sz, dimension, ...) creates a circular aperture with radius given by parameter dimension. Array is logical array.

Parameters

- sz – size of the pattern `[rows, cols]`
- dimension – List of numbers describing the aperture size. Lens of the list depends on the aperture shape. For a circle dimensions is one element, the radius of the circle.

Optional named parameters

- 'shape' – Shape of aperture to generate. See supported shapes bellow.
- 'value' [l, h] – values for off and on regions (default: [])
- 'centre' [x, y] – centre location for pattern
- 'offset' [x, y] – offset in rotated coordinate system
- 'aspect' (num) – aspect ratio of lens (default: 1.0)
- 'angle' (num) – Rotation angle about axis (radians)
- 'angle_deg' (num) – Rotation angle about axis (degrees)
- 'gpuArray' (logical) – If the result should be a gpuArray

Supported shapes [dimensions]

- 'circle' [radius] – Pinhole/circular aperture
- 'square' [width] – Square with equal sides
- 'rect' [w, h] – Rectangle with width and height
- 'ring' [r1, r2] – Ring specified by inner and outer radius

Fig. 4.8 shows examples of different apertures.

Logical arrays can be used to mask parts of other arrays. This can be useful for creating composite images, for example (see also Fig. 4.9):

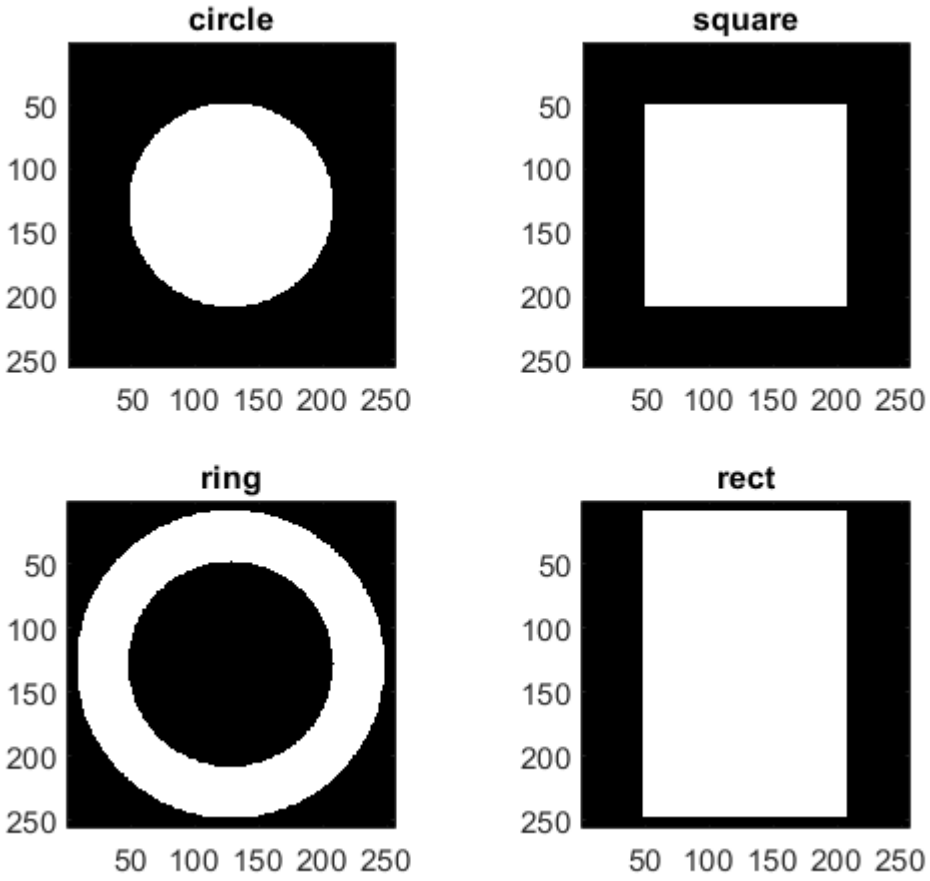

Fig. 4.8: Example of different aperture types generated by `aperture()`.

```
sz = [256, 256];
im = otslm.simple.linear(sz, 256);
chk = otslm.simple.checkerboard(sz);
app = otslm.simple.aperture(sz, 80);
im(app) = chk(app);
```

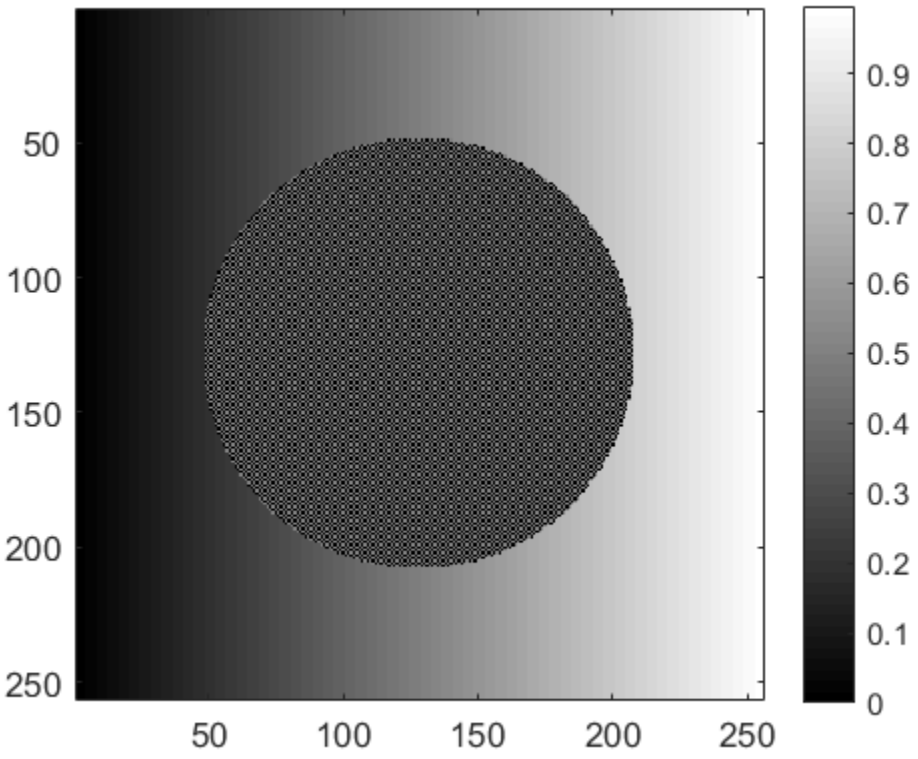

Fig. 4.9: Example of using `aperture()` to generate logical arrays for masking two patterns.

aberrationRiMismatch

`otslm.simple.`**`aberrationRiMismatch`**(*sz*, *n1*, *n2*, *alpha*, *varargin*)

Calculates aberration for a plane interface refractive index mismatch.

The aberration can be described using geometric optics, see Booth et al., Journal of Microscopy, Vol. 192, Pt 2, Nov. 1998. This function calculates the pattern according to

$$z(r) = df(r)n_1 \sin\alpha$$

where

$$f(r) = \sqrt{\csc^2\beta - r^2} - \sqrt{\csc^2\alpha - r^2},$$

d is the depth into medium 2, n_1, n_2 are the refractive indices in the mediums, $n_1 \sin\alpha = n_2 \sin\beta$ and α is the maximum capture angle of the lens which is related to the numerical aperture by $n_1 \sin\alpha$.

The focus is located in medium 2, which is separated from medium 1 and the lens by a plane interface.

Usage pattern = aberrationRiMismatch(sz, n1, n2, alpha, ...)

Parameters

- sz (size) – size of pattern `[rows, cols]`
- n1 (numeric) – refractive index of medium 1
- n2 (numeric) – refractive index of medium 2
- alpha (numeric) – maximum capture angle of lens (radians)

Optional named parameters

- radius (numeric) – radius of aperture. Default `min(sz)/2`.
- depth (numeric) – depth of focus into medium 2 (units of wavelength in medium). Default `1.0`.
- background (numeric|enum) – Specifies a background pattern to use for values outside the lens. Can also be a scalar, in which case all values are replaced by this value; or a string with 'random' or 'checkerboard' for these patterns.
- 'centre' [x, y] – centre location for lens
- 'offset' [x, y] – offset after applying transformations
- 'aspect' aspect – aspect ratio of lens (default: 1.0)
- 'angle' angle – Rotation angle about axis (radians)
- 'angle_deg' angle – Rotation angle about axis (degrees)
- 'gpuArray' bool – If the result should be a gpuArray

See also `examples.liveScripts.booth1998`.

Example usage (see also Fig. 4.10):

```
sz = [512, 512];
n1 = 1.5;
n2 = 1.33;
NA = 0.4;
alpha = asin(NA/n1);
pattern = otslm.simple.aberrationRiMismatch(sz, ...
    n1, n2, alpha, 'depth', 2.0);
```

zernike

`zernike()` generates a pattern based on the Zernike polynomials. The Zernike polynomials are a complete basis of orthogonal functions across a circular aperture. This makes them useful for describing beams or phase corrections to beams at the back-aperture of a microscope objective.

`otslm.simple.`**`zernike`**(*sz*, *m*, *n*, *varargin*)

Generates a pattern based on the zernike polynomials.

The polynomials are parameterised by two integers, m and n. n is a positive integer, and $|m| \leq n$.

Usage pattern = zernike(sz, m, n, ...)

Parameters

- sz (numeric) – size of the pattern `[rows, cols]`

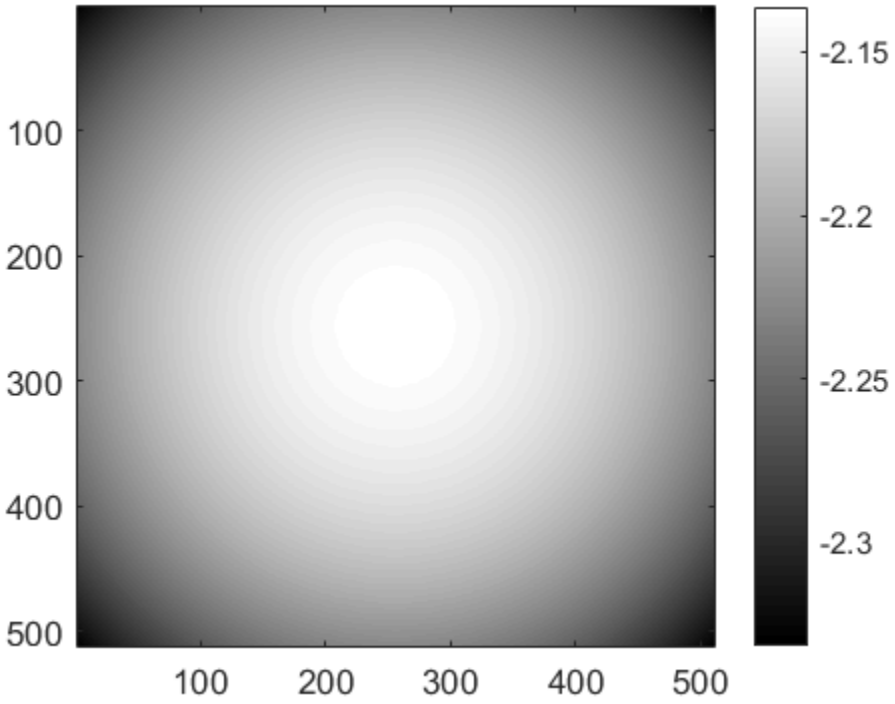

Fig. 4.10: Example output from `aberrationRiMismatch()`.

- m (numeric) – polynomial order parameter (integer)
- n (numeric) – polynomial order parameter (integer)

Optional named parameters

- 'scale' scale – scaling value for the final pattern
- 'rscale' rscale – radius scaling factor (default: min(sz)/2)
- 'outside' val – Value to use for outside points (default: 0)
- 'centre' [x, y] – centre location for lens (default: sz/2)
- 'offset' [x, y] – offset after applying transformations
- 'aspect' aspect – aspect ratio of lens (default: 1.0)
- 'angle' angle – Rotation angle about axis (radians)
- 'angle_deg' angle – Rotation angle about axis (degrees)
- 'gpuArray' bool – If the result should be a gpuArray

See also `examples.liveScripts.booth1998`.

Example usage (see also Fig. 4.11):

```
n = 4;
m = 2;
sz = [512, 512];
im = otslm.simple.zernike(sz, m, n);
```

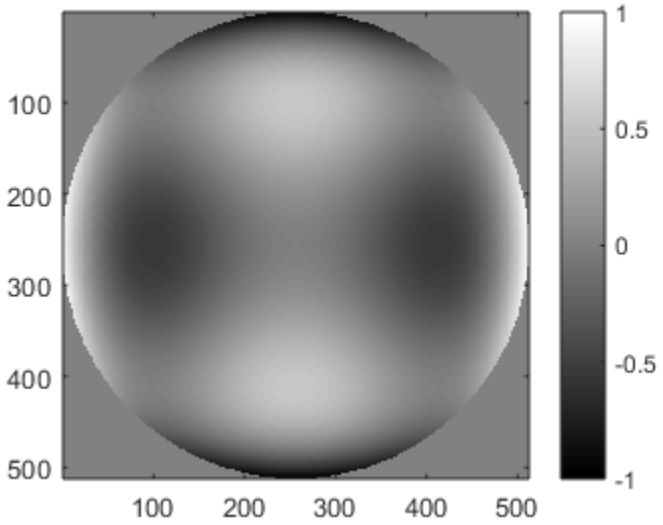

Fig. 4.11: Example output from `zernike()`.

sinc

otslm.simple.**sinc**(*sz*, *radius*, *varargin*)

Generates a sinc pattern. This pattern can be used to create a line shaped trap or as a model for the diffraction pattern from a aperture. The equation describing the pattern is

$$f(x) = \sin(\pi x/R)/(\pi x/R)$$

and 1 when x is zero; where R is the radius (scaling factor).

Usage pattern = sinc(sz, radius, ...)

Parameters

- sz (numeric) – size of the pattern `[rows, cols]`
- radius (numeric) – radial scaling factor

Optional named parameters

- 'type' type – the type of sinc pattern to generate
- '1d' – one dimensional
- '2d' – circular coordinates
- '2dcart' – multiple of two sinc functions at 90 degree angle supports two radius values: radius = [Rx, Ry].
- 'centre' [x, y] – centre location for lens
- 'offset' [x, y] – offset after applying transformations
- 'aspect' aspect – aspect ratio of lens (default: 1.0)
- 'angle' angle – Rotation angle about axis (radians)
- 'angle_deg' angle – Rotation angle about axis (degrees)

- 'gpuArray' bool – If the result should be a gpuArray

Example usage (see also Fig. 4.12):

```
radius = 10;
sz = [100, 100];
im = otslm.simple.sinc(sz, radius);
```

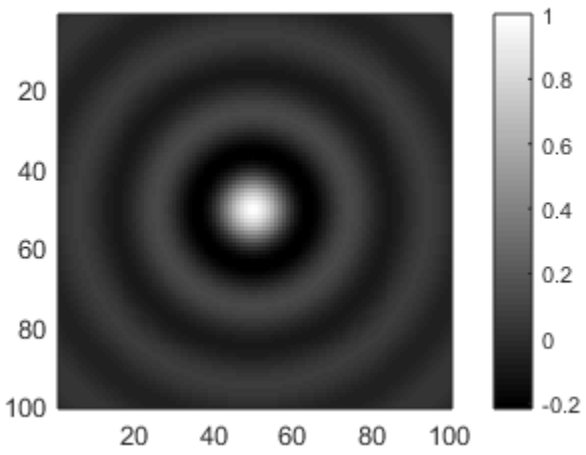

Fig. 4.12: Example output from `sinc()`.

checkerboard

otslm.simple.**checkerboard**(*sz*, *varargin*)

Generates a checkerboard pattern. A checkerboard pattern with equal sized squares can be written mathematically as

$$f(x, y) = \mod(x + y, 2)$$

Usage pattern = checkerboard(sz, ...) creates a checkerboard with spacing of 1 pixel and values of 0 and 0.5.

Parameters

- sz – size of the pattern `[rows, cols]`

Optional named parameters

- 'spacing' spacing – Width of checks (default 1 pixel)
- 'value' [l,h] – Lower and upper values of checks (default: `[0, 0.5]`)
- 'centre' [x, y] – centre location for lens (default: sz/2)
- 'offset' [x, y] – offset after applying transformations
- 'type' type – is the lens cylindrical or spherical (1d or 2d)
- 'aspect' aspect – aspect ratio of lens (default: 1.0)
- 'angle' angle – Rotation angle about axis (radians)
- 'angle_deg' angle – Rotation angle about axis (degrees)
- 'gpuArray' bool – If the result should be a gpuArray

Example usage (see also Fig. 4.13):

```
sz = [5,5];
im = otslm.simple.checkerboard(sz);
imagesc(im);
```

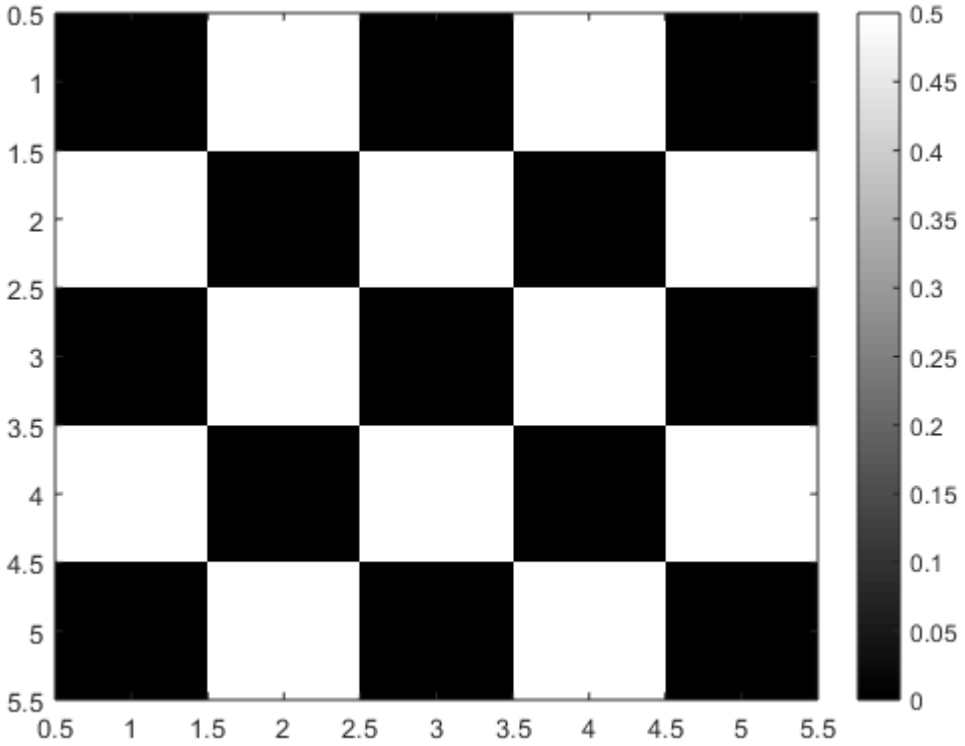

Fig. 4.13: Example output from *checkerboard()*.

grid

otslm.simple.**grid**(*sz*, *varargin*)

Generates a grid of points similar to meshgrid.

This function is used by most other otslm.simple functions to create grids of Cartesian or polar coordinates. Without any optional parameters, this function produces a similar result to the Matlab `meshgrid()` function but with 0 centred at the centre of the image.

Usage xx, yy = grid(sz, ...)

xx, yy, rr, phi = grid(sz, ...) calculates polar coordinates.

Parameters

- sz – size of the pattern `[rows, cols]`

Optional named parameters

- 'centre' [x, y] – centre location for lens (default: sz/2)
- 'offset' [x, y] – offset after applying transformations

- 'type' type – is the lens cylindrical or spherical (1d or 2d)
- 'aspect' aspect – aspect ratio of lens (default: 1.0)
- 'angle' angle – Rotation angle about axis (radians)
- 'angle_deg' angle – Rotation angle about axis (degrees)
- 'gpuArray' bool – If the result should be a gpuArray

Example usage (see also Fig. 4.14):

```
sz = [10, 10];
[xx, yy, rr, phi] = otslm.simple.grid(sz);
```

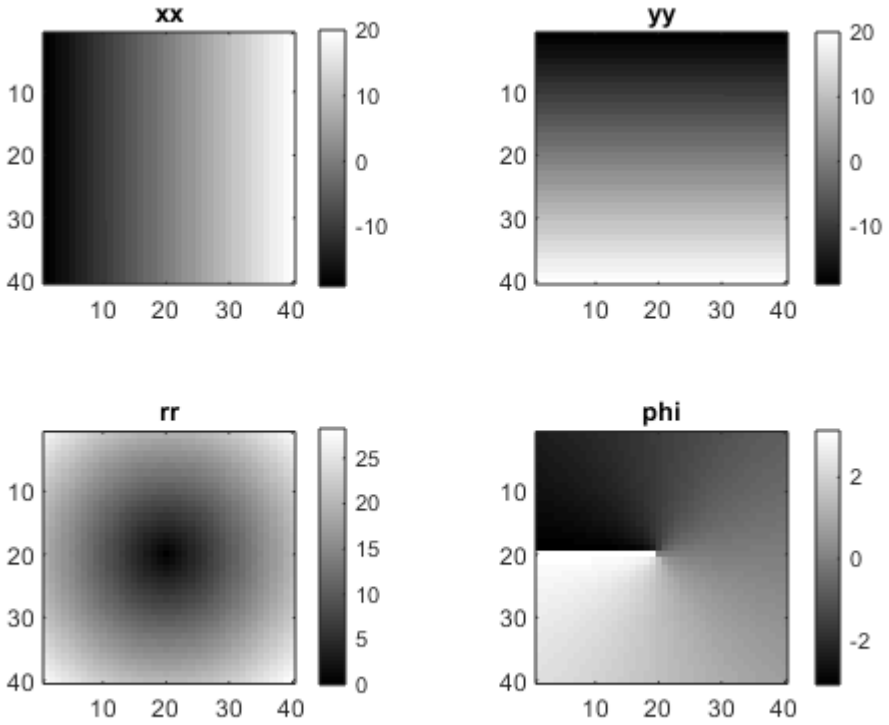

Fig. 4.14: Example output from `grid()`.

random

otslm.simple.**random**(*sz*, *varargin*)

Generates a image filled with random noise. The function supports three types of noise: uniform, normally distributed, and binary.

Usage pattern = random(sz, ...) creates a pattern with uniform random noise values between 0 and 1. See the 'type' argument for other noise types.

Parameters

- sz – size of the pattern

Optional named parameters

- 'range' (numeric) – Range of values (default: [0, 1]).
- 'type' (enum) – Type of noise. Can be 'uniform', 'gaussian', or 'binary'. (default: 'uniform')
- 'gpuArray' (logical) – If the result should be a gpuArray

Example (see also Fig. 4.15):

```
sz = [20, 20];
im = otslm.simple.random(sz, 'type', 'binary');
```

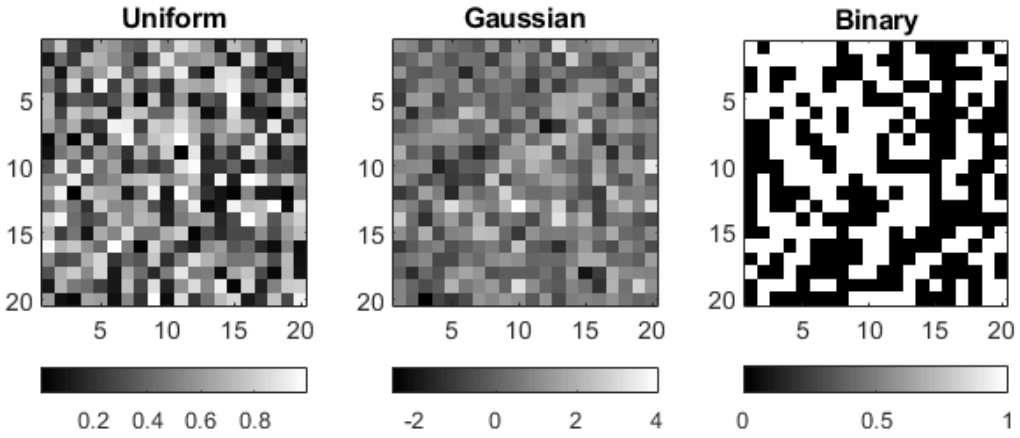

Fig. 4.15: Example output from `random()`.

step

`otslm.simple.`**`step`**(*sz*, *varargin*)

Generates a step. The function is described by

$$f(x) = 0 \qquad x < 0$$
$$f(x) = 1 \qquad x \geq 0$$

Usage pattern = step(sz, ...) generates a step at the centre of the image.

Parameters sz – size of the pattern

Optional named parameters

- 'value' [l, h] – low and high values of step (default: [0, 0.5])
- 'centre' [x, y] – centre location for pattern
- 'offset' [x, y] – offset in rotated coordinate system
- 'aspect' aspect – aspect ratio of lens (default: 1.0)
- 'angle' angle – Rotation angle about axis (radians)
- 'angle_deg' angle – Rotation angle about axis (degrees)

- 'gpuArray' bool – If the result should be a gpuArray

Example usage (see also Fig. 4.16):

```
sz = [5, 5];
im = otslm.simple.step(sz);
```

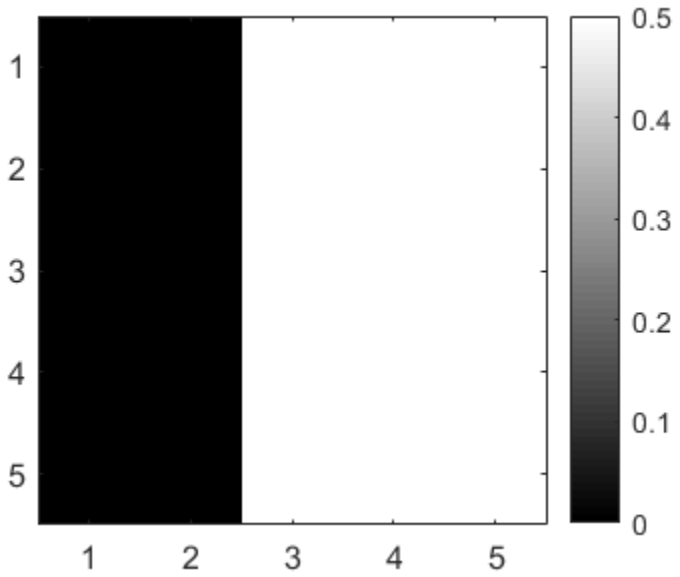

Fig. 4.16: Example output from *step()*.

4.1.5 3-D functions

These functions generate a 3-D volume instead of a 2-D image. The size parameter is a 3 element vector for the `x, y, z` dimension sizes.

Functions

- *aperture3d*
- *grid3d*
- *gaussian3d*
- *linear3d*

aperture3d

`otslm.simple.`**`aperture3d`**(*sz*, *dimension*, *varargin*)

Generate a 3-D volume similar to *aperture()*. Can be used to create a target 3-D volume for beam shape optimisation.

Usage pattern = aperture3d(sz, dimension, ...)

Properties

- sz – Size of the pattern `[rows, cols, depth]`
- dimensions – aperture dimensions (depends on aperture shape)

Optional named parameters

- 'shape' – Shape of aperture to generate. See supported shapes.
- 'value' [l, h] – values for off and on regions (default: [])
- 'centre' [x, y, z] – centre location for pattern
- 'gpuArray' (logical) – If the result should be a gpuArray

Supported shapes [dimensions]

- 'sphere' [radius] – Pinhole/circular aperture
- 'cube' [width] – Square with equal sides
- 'rect' [w, h, d] – Rectangle with width and height
- 'shell' [r1, r2] – Ring specified by inner and outer radius

grid3d

`otslm.simple.`**`grid3d`**(*sz*, *varargin*)

Generates a 3-D grid similar to *`grid()`*

Usage [xx, yy, zz] = grid3d(sz, ...) Equivalent to mesh grid.

[xx, yy, zz, rr, theta, phi] = grid3d(sz, ...) Additionally, calculates spherical coordinates:

- rr – Distance from centre of pattern
- theta – polar angle, measured from +z axis [0, pi]
- phi – azimuthal angle, measured from +x towards +y axes [0, 2*pi)

Parameters

- sz – size of the pattern `[rows, cols]`

Optional named parameters

- 'centre' [x, y, z] – centre location for lens
- 'gpuArray' bool – If the result should be a gpuArray

gaussian3d

`otslm.simple.`**`gaussian3d`**(*sz*, *sigma*, *varargin*)

Generates a gaussian volume similar to *`gaussian()`*.

The equation describing the lens is

$$z(r) = s \exp -r^2/(2\sigma^2)$$

where s is a scaling factor and σ describes the radius of the Gaussian distribution.

Usage pattern = gaussian3d(sz, sigma, ...)

Parameters

- sz – size of the pattern `[rows, cols, depth]`

- sigma – radius of the distribution σ. Can be a 1 or 3 element vector for the radial or `[x, y, z]` scaling.

Optional named parameters 'scale' scale scaling value for the final pattern 'centre' [x, y] centre location for lens 'gpuArray' bool If the result should be a gpuArray

linear3d

`otslm.simple.`**`linear3d`**(*sz*, *spacing*, *varargin*)

Generates a linear gradient similar to *`linear()`*

Usage pattern = linear3d(sz, spacing, ...)

Parameters

- sz – size of pattern to generate
- spacing – Inverse slope (1/spacing). Can be a scalar or a 3 element vector.

Optional named parameters

- 'centre' [x, y, z] – centre location for pattern
- 'gpuArray' bool – If the result should be a gpuArray

4.2 *iter* Package

Package containing algorithms and cost functions for iterative optimisation. This section is split into two parts, a description of the optimisation methods and a description of the objective function classes.

Contents

4.2.1 Iterative optimisation methods

The package contains a series of iterative optimisation algorithms. There are currently two types of iterative optimisation algorithms, methods that attempt to approximate some target far-field and methods which attempt to combine a series of SLM patterns. The former can also be used to combine a series of patterns by first generating a target far-field, for example using *`otslm.tools.combine()`*:

```
lin = otslm.simple.linear(sz, 10);
lg = otslm.simple.linear(sz, -10) + otslm.simple.lgmode(sz, 3, 0);

% Convert to complex amplitudes (could use finalize)
lin = exp(1i*2*pi*lin);
lg = exp(1i*2*pi*lg);

target = otslm.tools.combine({lin, lg}, 'method', 'farfield');
```

The methods which inherit from `IterBase` have an objective function property. For some methods the objective is required for the method to work, for other methods the objective is optional and can be used to track progress of the method. The objective can be set on construction or by setting the objective property. See the *Objective functions*

section for available objectives. For an example of how to use these iterative methods, see `examples.iterative`, `examples.iter_combine` and *Gerchberg-Saxton* examples. A minimal example for methods which attempt to generate a particular far-field is shown below:

```
sz = [512, 512];
incident = ones(sz);

prop = otslm.tools.prop.FftForward.simpleProp(zeros(sz));
vismethod = @(U) prop.propagate(U .* incident);

target = otslm.simple.aperture(sz, sz(1)/20);
gs = otslm.iter.GerchbergSaxton(target, 'adaptive', 1.0, ...
    'vismethod', vismethod);
```

Table 4.1 compares the run-time and required number of iterations for some of the iterative optimisation methods. This table is based on Di Leonardo et al. 2007, a more detailed discussion can be found in the reference. This is only a guide, some methods may work better than other methods under certain circumstances. For instance, the direct search method can be used for fine tuning the output of other methods but takes too long for practical use when given a bad initial guess. The combination algorithm and 2-D optimisation algorithms have been combined, actual performance will be different but similar. There are a range of different extensions to the described methods which may improve performance for particular problems, such as using a super-pixel style approach with the Direct Search algorithm.

Table 4.1: Comparison of iterative methods

Iterative methods	Num. Iterations	Typical Run-time
Gerchberg-Saxton	30	5 s
Weighted GS	30	5 s
Adaptive-adaptive	30	5 s
Bowman 2017	< 200	2 m
Simulated Annealing	10^4	10 m
Direct Search	10^9	days

Methods

- *GerchbergSaxton*
- *DirectSearch*
- *SimulatedAnnealing*
- *GerchbergSaxton3d*
- *CombineGerchbergSaxton*
- *IterBase*
- *IterCombine*
- *IterBaseEwald*
- *bsc*
- *bowman2017*

GerchbergSaxton

The Gerchberg-Saxton algorithm is an iterative algorithm that involves iterating between the near-field and far-field and applying constraints to the fields after each iteration. The constraints could include a particular incident illumination or a desired far-field intensity or phase pattern. Components that are not constrained are free to change. The algorithm was originally described in

> R. W. Gerchberg, O. A Saxton W., A practical algorithm for the determination of phase from image and diffraction plane pictures, Optik 35(1971) 237-250 (Nov 1971).

Details about the algorithm can be found on the Wikipedia page. A sketch of the algorithm for generating a target amplitude pattern using a phase-only device is shown below:

1. Generate initial guess for the SLM phase pattern P.
2. Calculate output for phase pattern: Proj$(P) \rightarrow O$.
3. Multiply output phase by target amplitude: $|T|\frac{O}{|O|} \rightarrow Q$.
4. Calculate the complex amplitude required to generate Q: Inv$(Q) \rightarrow I$.
5. Calculate new guess from the phase of I: Angle$(I) \rightarrow P$.
6. Goto step 2 until converged.

Proj and Inv are the forward and inverse propagation methods, these could be, for example, the forward and inverse Fourier transforms. A constraint for the incident illumination can be added to the forward propagator or the constraint can be added at another step. There are other variants for generating a target phase field or applying other constraints to the far-field. If this guess is symmetric, these symmetries will influence the final output, this can be useful for generating symmetric target fields.

GerchbergSaxton also implements the adaptive-adaptive algorithm, which we can enable by setting the `adaptive` parameter to a non-unity value. The adaptive-adaptive algorithm is similar to the above except step 3 mixes the propagator amplitude output with the target amplitude instead of replacing it

$$t = \alpha|T| + (1-\alpha)|O|$$
$$Q = t\frac{O}{|O|}$$

where α is the adaptive-adaptive factor.

class otslm.iter.**GerchbergSaxton**(*target*, *varargin*)

Implementation of Gerchberg-Saxton and Adaptive-Adaptive algorithms Inherits from *IterBase*.

Methods

- run() – Run the iterative method

Properties

- adaptive – Adaptive-adaptive factor (1 for Gerchberg-Saxton)

Inherited properties

- guess – Best guess at hologram pattern
- target – Target pattern the method tries to approximate
- vismethod – Method used to do the visualisation
- invmethod – Method used to calculate initial guess/inverse-visualisation
- objective – Objective function used to evaluate fitness
- fitness – Fitness evaluated after every iteration

See also GerchbergSaxton.

GerchbergSaxton(*target*, *varargin*)

Construct a new instance of the GerchbergSaxton iterative method

Usage mtd = GerchbergSaxton(target, ...)

Parameters

- target – target pattern

Optional named arguments

- adaptive num Adaptive-Adaptive factor. Default: 1.0, i.e. the method is Gerchberg-Saxton.
- guess im Initial guess at complex amplitude pattern. If not image is supplied, a guess is created using invmethod.
- vismethod fcn Function to calculate far-field. Takes one argument: the complex amplitude near-field. Default: @otslm.tools.prop.FftForward.simpleProp.propagate
- invmethod fcn Function to calculate near-field. Takes one argument: the complex amplitude far-field. Default: @otslm.tools.prop.FftInverse.simpleProp.propagate
- objective fcn Objective function to measure fitness. Default: @otslm.iter.objectives.FlatIntensity

DirectSearch

The direct search algorithm involves choosing a pixel, trying a range of possible values for that pixel, and keeping the choice which maximises some objective function. This is a expensive procedure, on a device with 512x512 pixels and 256 values per pixel, cycling over each pixel requires 67 million Fourier transforms. The process is further complicated since the optimal value for any pixel is not independent of every other pixel. However, this method can be useful for further improving a good guess, such as the output of one of the other methods.

A rough outline for the procedure is

1. Choose an initial guess, P
2. Randomly select a pixel to modify
3. Generate a set of patterns P_i with a set $\{i\}$ of different pixel values.
4. Propagate these patterns and calculate the fitness F_i
5. Choose the pattern which maximises the fitness $P_j \rightarrow P$ where $j = \text{argmax}_i F_i$.
6. Go to 2 until converged

class otslm.iter.**DirectSearch**(*target*, *varargin*)

Optimiser to search through each pixel value to optimise hologram Inherits from *IterBase*.

This method randomly selects a pixel in the pattern and then tries every available level. The pixel value kept is the pixel value whic gives the best fitness.

The algorithm is described in Di Leonardo, et al., Opt. Express 15, 1913-1922 (2007)

Methods

- run() – Run the iterative method

Properties

- levels – Discrete levels that will be search in optimisation

Inherited properties

- guess – Best guess at hologram pattern
- target – Target pattern the method tries to approximate
- vismethod – Method used to do the visualisation
- invmethod – Method used to calculate initial guess/inverse-visualisation
- objective – Objective function used to evaluate fitness
- fitness – Fitness evaluated after every iteration

See also DirectSearch and *SimulatedAnnealing*.

DirectSearch(*target*, *varargin*)

Construct a new instance of the DirectSearch iterative method

Usage mtd = DirectSearch(target, ...) attempts to produce the target using the Direct Search algorithm.

Optional named arguments:

- levels num – Number of discrete phase levels or array of levels between -pi and pi. Default: 256.
- guess im – Initial guess at complex amplitude pattern. If not image is supplied, a guess is created using invmethod.
- vismethod fcn – Function to calculate far-field. Takes one argument: the complex amplitude near-field. Default: @otslm.tools.prop.FftForward.simpleProp.propagate
- invmethod fcn – Function to calculate near-field. Takes one argument: the complex amplitude far-field. Default: @otslm.tools.prop.FftInverse.simpleProp.propagate
- objective fcn – Objective function to measure fitness. Default: @otslm.iter.objectives.FlatIntensity

SimulatedAnnealing

Simulated annealing is a stochastic method that can be useful for optimising systems with many degrees of freedom (such as patterns with many non-independent pixels). A description of the method can be found on the wikipedia page. The algorithm is analogous to cooling (annealing) of solids and chooses new state probabilistically depending on a temperature parameter. An outline follows

1. Starting with an initial pattern P and temperature T
2. Pick a random pattern which is similar to the current pattern
3. Compare fitness of two patterns F_1 and F_2
4. Accept the new pattern if $P(F_1, F_2, T) > \text{rand}(0, 1)$
5. Goto 2 until converged, gradually reducing temperature

There are several parameters that can be chosen which strongly affect the performance and convergence of the algorithm. The implementation currently only supports the following function

$$P(F_1, F_2, T) = \exp -(F_2 - F_1)/T$$

The change in temperature can be controlled via the `temperatureFcn` optional parameter.

This implementation could be improved and we welcome suggestions.

class otslm.iter.**SimulatedAnnealing**(*target*, *varargin*)

Optimise the pattern using simulated annealing. Inherits from *IterBase*.

Methods

- run() – Run the iterative method

Properties

- levels – Discrete levels that will be search in optimisation
- temperature – Current temperature of the system
- maxTemperature – Scaling factor for new pattern guesses
- temperatureFcn – Function used to calculate temperature in iteration
- lastFitness – The fitness associated with the current guess
- guess – Best guess at hologram pattern
- target – Target pattern the method tries to approximate
- vismethod – Method used to do the visualisation
- invmethod – Method used to calculate initial guess/inverse-visualisation
- objective – Objective function used to evaluate fitness
- fitness – Fitness evaluated after every iteration

See also SimulatedAnnealing

SimulatedAnnealing(*target*, *varargin*)

Construct a new instance of the SimulatedAnnealing iterative method

mtd = SimulatedAnnealing(target, ...) attempts to produce the target using the Simulated Annealing algorithm.

Optional named arguments:

- levels num Number of discrete levels or array of levels between -pi and pi. Default: 256.
- temperature num Initial temperature of the solver.
- guess im Initial guess at complex amplitude pattern. If not image is supplied, a guess is created using invmethod.
- vismethod fcn Function to calculate far-field. Takes one argument: the complex amplitude near-field. Default: @otslm.tools.prop.FftForward.simpleProp.propagate
- invmethod fcn Function to calculate near-field. Takes one argument: the complex amplitude far-field. Default: @otslm.tools.prop.FftInverse.simpleProp.propagate
- objective fcn Objective function to measure fitness. Default: @otslm.iter.objectives.FlatIntensity

static simpleTemperatureFcn(*scale*, *decay*)

Returns a exponentially decaying temperature function

fcn = simpleTemperatureFcn(scale, decay) creates a exponentially decaying temperature function. Scale is the initial temperature and decay is the exponential decay rate.

GerchbergSaxton3d

This function implements the 3-D analog of the Gerchberg-Saxton method. The method is described in

Hao Chen et al 2013 J. Opt. 15 035401

and

Graeme Whyte and Johannes Courtial 2005 New J. Phys. 7 117

For an outline of the Gerchberg-Saxton algorithm, see *GerchbergSaxton*.

class otslm.iter.**GerchbergSaxton3d**(*target*, *varargin*)

Implementation of 3-D Gerchberg-Saxton and Adaptive-Adaptive algorithms Inherits from *GerchbergSaxton* and *IterBaseEwald*.

This algorithm attempts to recreate the target volume using the 3-D analog of the Gerchberg-Saxton algorithm.

See Hao Chen et al 2013 J. Opt. 15 035401 and Graeme Whyte and Johannes Courtial 2005 New J. Phys. 7 117

Methods

- run() – Run the iterative method

Properties

- adaptive – Adaptive-adaptive factor (1 for Gerchberg-Saxton)

Inherited properties

- guess – Best guess at hologram pattern
- target – Target pattern the method tries to approximate
- vismethod – Method used to do the visualisation
- invmethod – Method used to calculate initial guess/inverse-visualisation
- objective – Objective function used to evaluate fitness
- fitness – Fitness evaluated after every iteration

See also GerchbergSaxton3d and *GerchbergSaxton*.

GerchbergSaxton3d(*target*, *varargin*)

Construct a new instance of the GerchbergSaxton3d iterative method

USage mtd = GerchbergSaxton3d(target, ...)

Parameters

- target – target pattern to try and generate

Optional named arguments

- adaptive num – Adaptive-Adaptive factor. Default: 1.0, i.e. the method is Gerchberg-Saxton.
- guess im – Initial guess at complex amplitude pattern. If not image is supplied, a guess is created using invmethod.
- vismethod fcn – Function to calculate far-field. Takes one argument: the complex amplitude near-field. Default: @otslm.tools.prop.FftEwaldForward.simpleProp.propagate
- invmethod fcn – Function to calculate near-field. Takes one argument: the complex amplitude far-field. Default: @otslm.tools.prop.FftEwaldInverse.simpleProp.propagate
- objective fcn – Optional objective function to measure fitness. Default: @otslm.iter.objectives.FlatIntensity

CombineGerchbergSaxton

This function implements the Gerchberg-Saxton algorithm and similar iterative optimisers for generating point traps. The method can be used to combine a set of SLM patterns ϕ_m into a single pattern in a similar way to *otslm.*

tools.combine(). Starting with an initial guess at the phase pattern ϕ^0 the method proceeds as

$$\phi^{j+1} = \sum_n e^{i\phi_n} \eta_n^j \frac{V_n^j}{|V_n^j|}$$

where

$$V_m^j = \sum_{x,y} e^{i(\phi^j(x,y) - \phi_m(x,y))}$$

and x, y are the SLM pixel coordinates and η_n^j is an optional parameter for Adaptive-Adaptive or weighted versions of the algorithm (for Gerchberg-Saxton $\eta = 1$). To calculate the pattern we simply need to iterative the above equation for a few steps.

There are two relatively simple extensions to this algorithm. First is the Adaptive-Adaptive algorithm which involves setting

$$\eta = \alpha + \frac{1 - \alpha}{|V_n^j|}$$

where α is a factor between 0 and 1. The second extension is the weighted Gerchberg-Saxton algorithm which involves setting

$$\eta^{j+1} = \eta^j \frac{\langle V_n^j \rangle}{V_n^j}$$

where $\langle \cdot \rangle$ denotes the average and we re-calculate η at each iteration starting with an initial value of 1.

To use the method we need to pass in a set of patterns to combine. For instance, we could have a set of 2 traps:

```
lin1 = otslm.simple.linear(sz, 10);
lin2 = otslm.simple.linear(sz, -5);

components = zeros([sz, 2]);
components(:, :, 1) = lin1;
components(:, :, 2) = lin1;
```

Then to use the iterative method we would run

```
mtd = otslm.iter.CombineGerchbergSaxton(2*pi*components, ...
   'weighted', true, 'adaptive', 1.0);
mtd.run(10);
imagesc(mtd.phase);
```

For a more complete example see `examples.iter_combine`. A more detailed discussion of these algorithms can be found in

> R. Di Leonardo, et al., Opt. Express 15 (4) (2007) 1913-1922. https://doi.org/10.1364/OE.15.001913

class otslm.iter.**CombineGerchbergSaxton**(*components*, *varargin*)

Implementation of Gerchberg-Saxton type combination algorithms. Inherits from *IterCombine*.

This includes Gerchberg-Saxton, Adaptive-Adaptive and weighted GerchbergSaxton algorithms.

For details about these algorithms, see R. Di Leonardo, et al., Opt. Express 15 (4) (2007) 1913-1922. https://doi.org/10.1364/OE.15.001913

Properties

- adaptive (numeric) – adaptive-adaptive factor.
- weighted (logical) – if the method is weighted Gerchberg-Saxton.

Methods (inherited)

- run() – Run the iterative method
- showFitness() – Show the fitness graph

Properties (inherited)

- components (real: 0, 2*pi) – NxMxD matrix of D patterns to be combined.
- guess – Best guess at hologram pattern (complex)
- target – Target pattern for estimating fitness (complex, optional)
- vismethod – Method used to do the visualisation
- invmethod – Method used to calculate initial guess/inverse-visualisation
- phase – Phase of the best guess (real: 0, 2*pi)
- amplitude – Amplitude of the best guess (real)
- objective – Objective function used to evaluate fitness or []
- fitness – Fitness evaluated after every iteration or []

See also CombineGerchbergSaxton and *GerchbergSaxton*.

CombineGerchbergSaxton (*components*, *varargin*)

Construct a new Gerchberg-Saxton combination iterative method.

Usage mtd = IterCombine(components, ...)

Parameters

- components (real: 0, 2*pi) – NxMxD array of D phase patterns to be combined. Phase patterns should have range [0, 2*pi] or equivalent.

Optional named arguments

- adaptive num Adaptive-Adaptive factor. Default: 1.0, i.e. the method is Gerchberg-Saxton.
- weighted (logical) – If the method should use weighted Gerchberg-Saxton. Default: false.
- target (complex) – approximate pattern for the target. This is only used for estimating the current fitness. Default: `otslm.tools.combine(components, 'method', 'farfield')`.
- guess (complex) – Initial guess at combination of patterns. Default: `exp(2*pi*1i*random_super)` where `random_super = tools.combine(components, 'method', 'rsuper')`
- vismethod fcn Function to calculate far-field. Takes one argument: the complex amplitude near-field. Optional, only used for fitness evaluation. Default: @otslm.tools.prop.FftForward.simpleProp.propagate
- invmethod fcn Function to calculate near-field. Takes one argument: the complex amplitude far-field. Optional, not used. Default: @otslm.tools.prop.FftInverse.simpleProp.propagate
- objective fcn Objective function to measure fitness. Default: @otslm.iter.objectives.FlatIntensity

IterBase

This is the base class for iterative methods. It is an abstract class and cannot be directly instantiated. To implement your own iterative method class, inherit from this class and implement the abstract methods/properties.

class otslm.iter.**IterBase**(*varargin*)

Base class for iterative algorithm classes. Inherits from handle.

Methods

- run() – Run the iterative method
- showFitness() – Show the fitness graph

Properties

- guess – Best guess at hologram pattern (complex)
- target – Target pattern the method tries to approximate (complex)
- vismethod – Method used to do the visualisation
- invmethod – Method used to calculate initial guess/inverse-visualisation
- phase – Phase of the best guess (real: 0, 2*pi)
- amplitude – Amplitude of the best guess (real)
- objective – Objective function used to evaluate fitness or []
- fitness – Fitness evaluated after every iteration or []

Abstract methods

- iteration() – run a single iteration of the method

IterBase(*varargin*)

Constructor for iterative algorithm (abstract) base class

Usage mtd = IterBase(target, ...)

Parameters

- target – target pattern to generate

Optional named arguments

- guess im Initial guess at complex amplitude pattern. If not image is supplied, a guess is created using invmethod.
- vismethod fcn Function to calculate far-field. Takes one argument: the complex amplitude near-field. Default: @otslm.tools.prop.FftForward.simpleProp.propagate
- invmethod fcn Function to calculate near-field. Takes one argument: the complex amplitude far-field. Default: @otslm.tools.prop.FftInverse.simpleProp.propagate
- objective fcn Objective function to measure fitness. Default: @otslm.iter.objectives.FlatIntensity

evaluateFitness(*mtd*, *varargin*)

Evaluate the fitness of the current guess

Usage score = mtd.evaluateFitness() visualises the current guess and evaluate the fitness.

score = mtd.evaluateFitness(guess) evaluate the fitness of the given guess. If guess is a stack of matrices, the returned score is a vector with size(trial, 3) elements. Guess should be a complex amplitude.

run(*mtd*, *num_iterations*, *varargin*)

Run the method for a specified number of iterations

Usage result = mtd.run(num_iterations, ...) run for the specified number of iterations.

Parameters

- num_iterations (numeric) – Number of iterations

Optional named arguments

- show_progress bool display a figure with optimisation progress

stopIterations (*mtd*, *src*, *event*)
Callback for the stop button in showFitness

Usage mtd.stopIterations(...) arguments are ignored.

IterCombine

This is the base class for iterative methods which combine multiple input pattern. It is an abstract class inheriting from *IterBase* however not all properties are needed/used by classes inheriting from this method. For instance, the *CombineGerchbergSaxton* class only uses the `vismethod` to calculate the fitness when an objective function is supplied. If the objective is omitted the method doesn't calculate the fitness and doesn't need `vismethod` or `invmethod`.

class otslm.iter.**IterCombine**(*components*, *varargin*)
Base class for iterative combination algorithms. Inherits from *IterBase*.

Iterative methods that inherit from this class attempt to combine a set of SLM phase patterns ϕ_m into a single phase pattern which generates a far-field phase pattern approximating the combination of each input phase pattern.

The target field is a optional and is only used for estimating fitness of the generated pattern.

Methods (inherited)

- run() – Run the iterative method
- showFitness() – Show the fitness graph

Properties

- components (real: 0, 2*pi) – NxMxD matrix of D patterns to be combined.

Properties (inherited)

- guess – Best guess at hologram pattern (complex)
- target – Target pattern for estimating fitness (complex, optional)
- vismethod – Method used to do the visualisation
- invmethod – Method used to calculate initial guess/inverse-visualisation
- phase – Phase of the best guess (real: 0, 2*pi)
- amplitude – Amplitude of the best guess (real)
- objective – Objective function used to evaluate fitness or []
- fitness – Fitness evaluated after every iteration or []

Abstract methods

- iteration() – run a single iteration of the method

IterCombine(*components*, *varargin*)
Constructor for iterative combination algorithms (abstract) base class

Usage mtd = IterCombine(components, ...)

Parameters

- components (real: 0, 2*pi) – NxMxD array of D phase patterns to be combined. Phase patterns should have range [0, 2*pi] or equivalent.

Optional named arguments

- target (complex) – approximate pattern for the target. This is only used for estimating the current fitness. Default: `otslm.tools.combine(components, 'method', 'farfield')`.
- guess (complex) – Initial guess at combination of patterns. Default: `exp(2*pi*1i*random_super)` where `random_super = tools.combine(components, 'method', 'rsuper')`
- vismethod fcn Function to calculate far-field. Takes one argument: the complex amplitude near-field. Optional: only used for fitness evaluation. Default: @otslm.tools.prop.FftForward.simpleProp.propagate
- invmethod fcn Function to calculate near-field. Takes one argument: the complex amplitude far-field. Optional: not used. Default: @otslm.tools.prop.FftInverse.simpleProp.propagate
- objective fcn Objective function to measure fitness. Default: @otslm.iter.objectives.FlatIntensity

IterBaseEwald

This is the base class for iterative methods that 3-D Fourier transforms and an Ewald sphere far-field mapping. This is class can be combined with *IterBase* to provide the 3-D specialisation. Currently only used by *GerchbergSaxton3d*.

class otslm.iter.**IterBaseEwald**(*target*, *varargin*)

Abstract base class for 3-D Ewald iterative algorithm classes Inherits from *IterBase*.

Methods

- run() – Run the iterative method

Properties

- guess – Best guess at hologram pattern (complex, matrix)
- target – Target pattern the method tries to approximate (volume)
- vismethod – Method used to do the visualisation
- invmethod – Method used to calculate initial guess/inverse-visualisation
- phase – Phase of the best guess (real: 0, 2*pi)
- amplitude – Amplitude of the best guess (real)
- objective – Objective function used to evaluate fitness or []
- fitness – Fitness evaluated after every iteration or []

Abstract methods

- iteration() – run a single iteration of the method

IterBaseEwald(*target*, *varargin*)

Abstract constructor for 3-D iterative algorithm base class

Usage mtd = IterBaseEwald(target, ...) constructs a new instance. target should be a 3-D volume. Guess, if supplied, should be a 2-D matrix for the pattern on the SLM.

Optional named arguments:

- guess im Initial guess at complex amplitude pattern. If no image is supplied, a guess is created using invmethod.
- vismethod fcn Function to calculate far-field. Takes one argument: the complex amplitude near-field. Default: @otslm.tools.prop.FftEwaldForward.simpleProp.propagate
- invmethod fcn Function to calculate near-field. Takes one argument: the complex amplitude far-field. Default: @otslm.tools.prop.FftEwaldInverse.simpleProp.propagate
- objective fcn Objective function to measure fitness. Default: @otslm.iter.objectives.FlatIntensity

bsc

This function attempts to optimise the beam using vector spherical wave functions. The function may be unstable/change in future releases but demonstrates how OTT can be used with OTSLM.

otslm.iter.**bsc**(*sz*, *target*, *varargin*)

Optimisation in vector spherical wave function basis

Requires the optical tweezers toolbox (OTT).

Usage [pattern, beam, coeffs] = bsc(target, ...) attempt to produce target using a phase pattern. Returns the phase pattern matched to the beam (bsc) and optimised basis weighting coefficients.

Parameters

- target – target pattern

Optional named parameters

- 'incident' pattern – Incident illumination on SLM
- 'roi' func – Region to optimise (default: roiAll)
- 'basis' str – BSC basis to optimise in (default: vswf_lg)
- 'basis_size' num – Number of basis functions to use
- 'polarisation' [x y] – Polarisation of the basis functions
- 'wavelength' num – Wavelength in medium [m]
- 'speed' num – Speed in medium [m/s]
- 'NA' num – Numberical aperture of objective
- 'pixel_size' num – Size of pixels in target [m]
- 'method' str – Optimisation method to use
- 'radius' num – Radius for hologram unwrapping (default: 1.0)

bowman2017

This function provides an interface for Bowman, et al. Optics Express 25, 11692 (2017). This requires a suitable Python version and various libraries. The wrapper may be unstable and will hopefully be improved in future releases.

otslm.iter.**bowman2017**(*target*, *varargin*)

Wrapper for Bowman 2017 conjugate gradient implementation

See Bowman, et al. Optics Express 25, 11692 (2017) If you use this method, please consider citing Bowman 2017.

> **Warning:** This wrapper may be unstable and may change in future releases.

Usage pattern = bowman2017(target, ...) attempt to generate the target using a phase pattern optimised using conjugate gradient method.

Parameters

- target – target pattern to generate

Optional named parameters

- 'guess' – Initial guess at the phase
- 'iterations' – Number of iterations (default: 200)
- 'steepness' – Steepness for Bowman cost function (default: 9.0)
- 'incident' – Incident illumination (default: ones)
- 'roisize' – Optimisation region size (default: min(size)/2)

4.2.2 Objective functions

Objective functions are contained in the `otslm.iter.objectives` sub-package. These functions are used with the above optimisation methods for both optimisation and diagnostics.

To evaluate the fitness (similarity) between a trial pattern and a target, we can construct a new objective instance and call the evaluate method. For example, using the *`Flatness`* objective:

```
% Setup the trial and target
sz = [256, 256];
target = ones(sz);
trial = randn(sz) + 1.0;

% Setup the objective
obj = otslm.iter.objectives.Flatness('target', target);

% Evaluate the fitness
fitness = obj.evaluate(trial);
```

It is possible to reuse the objective multiple times or test the trial pattern against a different target pattern when evaluate is called:

```
new_target = zeros(sz);
fitness = obj.evaluate(trial, new_target);
```

Objective classes support a region of interest mask. The region of interest can either be a logical mask or a function which selects a region of the image, for example:

```
% Select only half of the image with a function
obj.roi = @(pattern) pattern(1:end/2, :)
fitness = obj.evaluate(trial);

% Use a logical array
obj.roi = otslm.simple.aperture(sz, 128);
fitness = obj.evaluate(trial);
```

Objectives

Objective base class

class otslm.iter.objectives.**Objective**(*varargin*)

Abstract base class for optimisation objective functions.

To use this class, you need to inherit from it and implement the evaluate_internal function.

Methods

- evaluate() – Evaluate the fitness of the specified pattern

Properties

- target (numeric) – Target pattern to compare with (or [] for no default). This is only used if no target is provided in evaluate().
- type (enum) – Type of optimisation function ('min' or 'max') This property isn't widely used (may change in future version). For now, most functions simply have this property set to 'min'.
- roi (logicallemptylfunction_handle) – Region of interest to apply to target and trial. Must be either a logical array the same size as target, an empty matrix for no roi, or a function handle. If roi is a function handle, the function should have the signature `masked = roi(pattern)`. Calling the function should select elements from the pattern for comparison. The function is applied to both the target and the trial pattern.

Abstract methods

- evaluate_internal() – Implementation called by evaluate(). Signature: `fitness = obj.evaluate_internal(target, trial)`. The roi has already been applied to the trial and target.

See also Objective, *`Intensity`* and *`Flatness`*.

Objective(*varargin*)

Construct a new objective function instance

Usage obj = Objective(...) construct a new objective function instance.

Optional named arguments

- roi [] | logical | function_handle – specify the roi to use when evaluating the fitness function. Can be a logical array or a function handle. Default: []
- target [] | matrix – specify the target pattern for this objective. If not supplied, the target must be supplied when the evaluate function is called. Default: []

evaluate (*trial*, *target*)
Evaluate the fitness of the specified trial pattern.

Usage fitness = obj.evaluate(trial, [target]) evaluate the specified trial pattern. If target is not specified, uses the internal target pattern set during construction.

Parameters

- trial (numeric) – pattern to compare to target
- target (numeric) – pattern to compare to trial. Optional. Default target is obj.target.

Bowman2017

class otslm.iter.objectives.**Bowman2017** (*varargin*)
Cost function used in Bowman et al. 2017 paper. Inherits from *Objective*.

$$C = 10^d * (1.0 - \sum_{nm} \sqrt{I_n m T_n m} \cos(phi_n m - psi_n m))^2$$

target and trial should be the complex field amplitudes.

Properties

- scale – d scaling factor in cost function
- field – 'complex', 'phase', or 'amplitude' for optimisation type
- normalize – Normalize target/trial every evaluation

See also Bowman2017 and *Intensity*.

Bowman2017 (*varargin*)
Construct a new objective function instance

Usage obj = Bowman2017(...) construct a new objective function instance.

Optional named arguments

- scale num – *d* scaling factor in cost function. Default: 0.5
- field [char] – One of 'complex', 'phase', or 'amplitude' for optimisation type. Default: 'complex'.
- normalize bool – Normalize target/trial every evaluation. Default: true.
- roi [] | logical | function_handle – specify the roi to use when evaluating the fitness function. Can be a logical array or a function handle. Default: []
- target [] | matrix – specify the target pattern for this objective. If not supplied, the target must be supplied when the evaluate function is called. Default: []

FlatIntensity

class otslm.iter.objectives.**FlatIntensity** (*varargin*)
Objective function for pattern flatness and intensity. Inherits from *Intensity* and *Flatness*.

Evaluates the fitness using the Intensity and Flatness method and adds the result:

Properties

- flatness – scaling factor for pattern flatness

See also FlatIntensity and *Bowman2017*.

FlatIntensity(*varargin*)
Construct a new objective function instance

Usage obj = FlatIntensity(...)

Optional named arguments

- roi [] | logical | function_handle – specify the roi to use when evaluating the fitness function. Can be a logical array or a function handle. Default: []
- target [] | matrix – specify the target pattern for this objective. If not supplied, the target must be supplied when the evaluate function is called. Default: []
- flatness num – Scaling factor for pattern flatness. Default: 0.5.

Flatness

class otslm.iter.objectives.**Flatness**(*varargin*)
Objective function for pattern flatness Inherits from *Objective*.

See also Flatness, *Intensity* and *FlatIntensity*.

Flatness(*varargin*)
Construct a new objective function instance

Usage obj = Flatness(...)

Optional named arguments

- roi [] | logical | function_handle specify the roi to use when evaluating the fitness function. Can be a logical array or a function handle. Default: []
- target [] | matrix specify the target pattern for this objective. If not supplied, the target must be supplied when the evaluate function is called. Default: []

Goorden2014

class otslm.iter.objectives.**Goorden2014**(*varargin*)
Fidelity function from Goorden, et al. 2014 paper.

$$F = |\text{conj}(target) * trial|^2$$

Error is 1 - F.

Properties

- normalize (logical) – True if the patterns should be normalized by the area (i.e., $F = F/A^2$).

See also Goorden2014, *Flatness* and :class‘Bowman2017‘.

Goorden2014(*varargin*)
Construct a new objective function instance

Usage obj = Goorden2014(...)

Optional named arguments

- normalize logical – If true, normalized the pattern by the number of pixels in the pattern. Default: true.
- roi [] | logical | function_handle – specify the roi to use when evaluating the fitness function. Can be a logical array or a function handle. Default: []

- target [] | matrix – specify the target pattern for this objective. If not supplied, the target must be supplied when the evaluate function is called. Default: []

Intensity

class otslm.iter.objectives.**Intensity**(*varargin*)

Objective function for pattern intensity.

See also Intensity, *Flatness* and *FlatIntensity*.

Intensity(*varargin*)

Construct a new objective function instance

Usage obj = Intensity(...)

Optional named arguments

- roi [] | logical | function_handle specify the roi to use when evaluating the fitness function. Can be a logical array or a function handle. Default: []
- target [] | matrix specify the target pattern for this objective. If not supplied, the target must be supplied when the evaluate function is called. Default: []

RmsIntensity

class otslm.iter.objectives.**RmsIntensity**(*varargin*)

Objective function for pattern RMS intensity Inherits from *Objective*.

Evaluates the fitness according to

$$F = \sqrt{\text{mean}((|t|^2 - |T|^2)^2)}$$

Where t and T are the trial and target pattern complex amplitudes.

See also RmsIntensity, *Intensity* and *Flatness*.

RmsIntensity(*varargin*)

Construct a new objective function instance

Usage obj = RmsIntensity(...)

Optional named arguments

- roi [] | logical | function_handle specify the roi to use when evaluating the fitness function. Can be a logical array or a function handle. Default: []
- target [] | matrix specify the target pattern for this objective. If not supplied, the target must be supplied when the evaluate function is called. Default: []

4.3 *tools* Package

The otslm.tools package is a collection of functions for working with and combining patterns. This includes tools for visualising patterns, generating patterns which combine the phase and amplitude information of a target beam into a single pattern, and various other tools.

These functions are commonly used to modify the output of functions in the *simple Package* or the *iter Package*. Patterns are represented by 2-D matrices and volumes by 3-D matrices.

This package also contains the *prop sub-package*. This package contains classes for simulating the propagation of patterns.

Some functionality requires the optical tweezers toolbox. Functions requiring the toolbox have a note in their documentation (in the Matlab help and this documentation).

4.3.1 Functions

combine

`otslm.tools.`**`combine`**(*inputs*, *varargin*)

Combines multiple patterns

Typical input should be a pattern between 0 and 1. Most methods output range is between 0 and 1.

For iterative combination methods, see `otslm.iter.IterCombine` or generate a target field using the `farfield` method and use an `otslm.iter.IterBase` iterative method.

Usage pattern = combine(inputs, ...) combines the cell array of patterns.

Parameters

- inputs (cell) – cell array of input images to combine. These images should all be the same size.

Optional named parameters

- 'method' (enum) – Method to use when combining patterns.

Methods to create multiple beams

- dither – Randomly chooses values from different patterns
- super – Uses phi = angle(sum_ii exp(1i*2*pi*inputs(ii)))
- rsuper – Superposition with random offset for each layer

Methods to modulate a beam pattern

- add – Adds the patterns: sum_ii inputs(ii)
- multiply – Multiplies the patterns: prod_ii inputs(ii)
- addangle – Uses phi = angle(prod_ii exp(1i*2*pi*inputs(ii)))
- average – Weighted average of inputs. (default weights: ones) $\sum_i w_i I_i / \sum_i w_i$

Miscelanious

- farfield – Calculate farfield sum: sum_ii Prop(inputs(ii)). This method assumes the input has the currect range for the propagator. The default propagator is a FFT, so the inputs should be complex amplitudes.

Default method: super.

- 'weights' (numeric) – Array of weights, one for each pattern. (default: [], uses equal weights for each pattern)
- 'vismethod' (fcn) – Used by `farfield` method. Default: `@otslm.tools.prop.FftForward.simpleProp.evaluate`.

See also Di Leonardo, Ianni and Ruocco (2007).

dither

The *`dither()`* function can be used to convert a continuous gray-scale image into a binary pattern. This can be useful for converting gray-scale amplitude images into binary images for display on a digital micro-mirror device.

The function supports a range of different dither methods including Matlab's builtin dither, raw thresholding, random dithering and using the Floyd-Steinberg algorithm. The following code example demonstrates a couple of different methods, the results are shown in Fig. 4.17.

```
im = otslm.simple.linear([256, 256], 256);
d1 = otslm.tools.dither(im, 0.5, 'method', 'threshold');
d2 = otslm.tools.dither(im, 0.5, 'method', 'mdither');
d3 = otslm.tools.dither(im, 0.5, 'method', 'floyd');
d4 = otslm.tools.dither(im, 0.5, 'method', 'random');
```

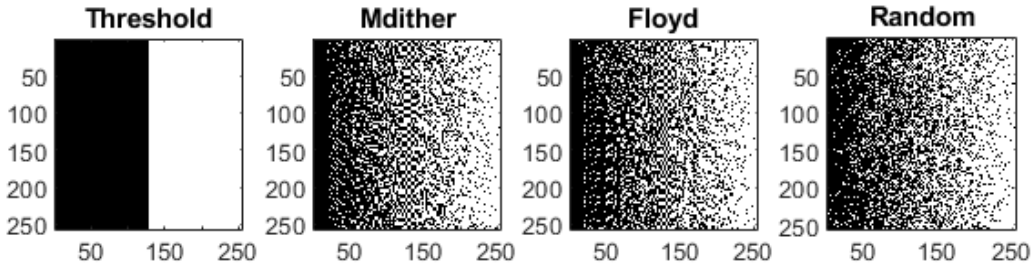

Fig. 4.17: Example output from *`dither()`* using different methods.

otslm.tools.**dither**(*pattern*, *level*, *varargin*)
Creates a binary pattern from gray-scale image. Supports several different dithering methods.

Usage pattern = dither(pattern, level, ...) applies the default dithering, binary threshold, to the pattern.

Parameters

- pattern (numeric) – the gray-scale pattern. Most methods assume the pattern has values in the range 0 to 1.
- level (numeric) – threshold level

Optional named parameters

- 'method' (enum) – Method to use for dithering. Supported methods:
- 'threshold' – Apply threshold filter to image (default)
- 'mdither' – Use matlab dither function
- 'floyd' – Floyd-Steinberg algorithm
- 'random' – Does random dithering
- 'value' [min, max] Value range for output image (default: [] for logical images). See *castValue()*.

encode1d

otslm.tools.**encode1d**(*target*, *varargin*)
Encode the target pattern amplitude into the phase pattern size

Usage [pattern, assigned] = encode1d(target, ...) encodes the complex target pattern into a 1-D phase mask. Returns the encoded pattern and a logical matrix of the same size that specifies if pixels were used to encode the pattern.

Parameters

- target (numeric) – vector containing values of 1-D amplitude function to be encoded.

Optional named arguments

- 'scale' (numeric) – Scale for the height of the pattern
- 'angle' (numeric) – Rotation angle about axis (radians)
- 'angle_deg' (numeric) – Rotation angle about axis (degrees)

finalize

otslm.tools.**finalize**(*pattern*, *varargin*)
Finalize a pattern, applying a color map and taking the modulo.

Usage pattern = finalize(input, ...) finalizes the pattern. For dmd type devices, the input is assumed to be the amplitude. For slm type devices, the input is assumed to be the phase.

pattern = finalize(phase, 'amplitude', amplitude', ...) attempts to generate a pattern encoding both the phase and amplitude.

Parameters

- pattern (numeric) – phase pattern to be finalized

Optional named parameters

- 'modulo' (numeric|enum) – Applies modulo to the pattern. Modulo should either be a scalar or 'none' for no modulo. (default: 1.0 for slm type devices and 'none' for dmd type devices).
- 'colormap' – Colormap to apply to pattern. For a list of valid values, see *colormap()*. (default: 'pmpi' for slm and 'gray' for dmd type devices).
- 'rpack' (enum) – rotation packing of the pixels.
- 'none' – No additional steps required (default slm)
- '45deg' – Device is rotated 45 degrees (aspect 1:2, default dmd)
- 'device' (enum) – Specifies the type of device, changes the default value for most arguments. If all arguments are provided, this argument has no impact.
- 'dmd' – Digital micro mirror (amplitude) device
- 'slm' – Spatial light modulator (phase) device
- 'encodemethod' method Method to use when encoding phase/amplitude
- 'checker' – (default) use checkerboard pattern and acos correction (phase)
- 'grating' – Use linear grating and sinc correction (phase)
- 'magnitude' – Use grating magnitude modulation (phase)
- 'amplitude' pattern Amplitude pattern to generate output for

hologram2volume

otslm.tools.**hologram2volume** (*hologram*, *varargin*)

Generate 3-D volume representation from hologram.

This function is only the inverse of volume2hologram when interpolation is disabled for both.

Usage volume = hologram2volume(hologram, ...) generates a 3-D volume for 2-D complex amplitude hologram. Unwraps hologram onto Ewald sphere.

Parameters hologram (numeric) – 2-D hologram to map to Ewald sphere.

Optional named arguments

- 'interpolate' (logical) – Interpolate between the nearest two pixels in the z-direction. (default: True)
- 'padding' (numeric) – Padding in the axial direction (default 0).
- 'focal_length' (numeric) – focal length in pixels (default: min(size)/2).
- 'zsize' (size) – size for z depth (default: []) The total z size is zsize + 2*padding.

See also *volume2hologram()* and `prop.FftEwaldForward`

mask_regions

otslm.tools.**mask_regions** (*base*, *patterns*, *locations*, *sizes*, *varargin*)

Adds patterns to base using masking

Usage pattern = mask_region(base, patterns, locations, sizes, ...)

Parameters

- base (numeric) – base pattern to mask and add regions to
- patterns (cell) – cell array of patterns to be added. Each pattern must be the same size as base. Patterns should be numeric.
- locations (cell) – cell array containing vectors for the centre of each mask region. Must be the same length as patterns.
- sizes – size parameters for each shape (see options below). Number of sizes must be 1 (for a single shape) or match the length of patterns.

Optional named parameters

- 'shape' (cell|enum) – shape to use for masking. Must either be a single shape or a cell array of shapes with the same number of elements as patterns. Supported shapes and [sizes] include:
- 'circle' [radius] Use a circular aperture.
- 'square' [width] Square with equal sides.
- 'rect' [w, h] Rectangle with width and height.

sample_region

`otslm.tools.`**`sample_region`**(*sz*, *locations*, *detectors*, *varargin*)

Generates a pattern for sampling regions on SLM.

Usage pattern = sample_region(sz, locations, detectors, ...) generates the patterns for sampling regions at different SLM locations onto detectors located at detector locations. The range for the pattern is 0 to 1, so the output should be passed to otslm.tools.finalize.

Parameters

- sz (size) – size of the generated pattern `[rows, cols]`
- locations (cell{numeric}) – cell array of locations in the generated pattern to place regions. `{[x1, y1], [x2, y2], ...}`. Location shave units of pixels.
- detectors (cell{numeric}) – cell array of numbers describing the locations in the far-field. `{[x1, y1], [x2, y2], ...}`. This locations are passed into *`otslm.simple.linear()`* as the spacing argument. Must be the same length as `locations` or be a single location. If detectors is a single location, all the patterns will point to the same detector.

Most optional named parameters can also be cell arrays (or cell arrays of cell arrays) for different options for each location.

Optional named parameters

- radii (numeric) – Radius of each SLM region. Should be a single element or an array the same length as `locations`.
- amplitude (enum) – Specifies a method for amplitude modulation. See below for a list of methods and arguments.
- ampliutdeargs (cell) – Cell array of arguments to pass to amplitude method.
- background (enum) – Specifies the background type. Possible values are:
- 'zero' – Uses 0 phase as the background.
- 'nan' – Uses NaN phase as the background.
- 'checkerboard' – Uses a checkerboard for the background.

- 'random' – Uses noise for the background.
- 'randombin' – Uses binary noise for the background.

Possible amplitude methods are

- 'step' – Sharp step between background and pattern. No parameters.
- 'gaussian_dither' – Randomly mixes in background. Parameters:
- 'offset' (numeric) – offset for dither threshold.
- 'noise' (numeric) – scale of uniform noise to add to pattern.
- 'gaussian_noise' – Adds noise to edge of the pattern. Parameters:
- 'offset' (numeric) – Offset.
- 'scale' (numeric) – Uniform noise range or Gaussian width.
- 'type' (enum) – type of noise, either 'uniform' or 'gaussian'
- 'gaussian_scale' – Scales the pattern by a Gaussian and then uses the mix method to combine the pattern with the background. The mix method must be 'add' for adding the result to the background, or 'step' for placing the scaled pattern on the background as a step. Parameters:
- 'mix' (enum) – mix method, 'add' or 'step'
- 'mixargs' (cell) – arguments for mix method
- 'scale' (numeric) – scaling factor

spatial_filter

`otslm.tools.`**`spatial_filter`**(*input*, *filter*, *varargin*)

Applies a spatial filter to the image spectrum. This can be used to simulate imaging or focussing of light using an objective with different shaped apertures or for adding spherical aberration to the system.

Usage [output, filtered] = spatial_filter(input, filter, ...) Applies filter to the Fourier transform of input and calculates the inverse Fourier transform to give output. Optional output filtered is the filtered pattern.

Parameters

- input (numeric) – image to apply filter to.
- filter (numeric) – a mask pattern to apply to the far-field of the input. Should be the same size or smaller than the output of the forward propagation method. If it is smaller, the array is padded with zeros.

Optional named parameters

- vismethod (fcn) – Function to calculate far-field. Takes one argument, the complex amplitude near-field. Default: @otslm.tools.prop.FftForward.simpleProp.evaluate
- invmethod (fcn) – Function to calculate near-field. Takes one argument: the complex amplitude far-field. Default: @otslm.tools.prop.FftInverse.simpleProp.evaluate
- gpuArray (logical) – If the result should be a gpuArray. Default: `isa(input, 'gpuarray')`.

Padding can be controlled by changing the vis and inv methods.

See also `examples.liveScripts.booth1998`

visualise

otslm.tools.**visualise**(*phase*, *varargin*)

Generates far-field plane images of the phase pattern

Usage [output, ...] = visualise(phase, ...) visualise the phase plane. Some methods output additional parameters, such as the ott-toolbox beam.

If phase is an empty array and one of the other images is supplied, the phase is assumed to be an array of zeros the same size as one of the other images.

[output, ...] = visualise(complex_amplitude, ...) visualise the field with complex amplitude.

Parameters

- phase (real) – The phase image should be in a range from 0 to 2*pi. If the range is approximately 1 a warning is issued.
- complex_amplitude (complex) – a complex amplitude pattern to visualise.

Optional named parameters

- 'method' (enum) – Method to use when calculating visualisation. Current supported methods:
- 'fft' Use fourier transform approach described in https://doi.org/10.1364/JOSAA.15.000857
- 'fft3' Use 3-D Fourier transform, if original image is 2-D, converts to volume and takes Fourier transform. If input is 3-D, directly applies 3-D Fourier transform.
- 'ott' Use optical tweezers toolbox (OTT).
- 'rs' Rayleigh-Sommerfeld diffraction formula
- 'rslens' Use rs to propagate to a lens, apply the lens phase pattern and propagate some distance from the lens.
- 'type' type Type of transformation: 'nearfield' or 'farfield'
- 'amplitude' image Specifies the amplitude pattern
- 'incident' image Specifies the incident illumination Default illumination is uniform intensity and phase.
- 'z' z z-position of output plane. For fft/ott this is an offset from the focal plane. For rs/rslens, this is the distance along the beam axis.
- 'padding' p Add padding to the outside of the image. Default: ceil(size(phase)/2)
- trim_padding (logical) – Trim padding before returning result (default: false).
- NA (numeric) – Numerical aparture of the lens (default: 0.1)
- resample (numeric) – Number of samples per each pixel (default: []).

bsc2hologram

otslm.tools.**bsc2hologram**(*sz*, *beam*, *varargin*)

Calculates the far-field hologram for a BSC beam

Requires the optical tweezers toolbox (OTT).

> **Warning:** The current version of the optical tweezers toolbox may introduce additional phase artifacts in the far-field.

Usage [phase, cpattern] = bsc2hologram(sz, beam, ...) calculates the phase pattern that transforms the incident beam to the BSC beam. Additionally, outputs the complex x and y polarisation complex amplitudes of the BSC beam in the far-field (szx2 matrix).

Parameters

- sz – size of pattern
- beam – Optical tweezers toolbox Bsc beam object

Optional named parameters

- 'incident' im – Incident beam complex amplitude (default: ones)
- 'polarisation' [x y] – Polarisation of incident beam (default: [1 1i])
- 'encodemethod' str – Amplitude encode method (see tools.finalize)
- 'radius' r – Radius of the hologram pattern (default: 1.0)

colormap

otslm.tools.**colormap**(*pattern*, *cmap*, *varargin*)

Applies a colormap to a pattern.

This method either applies nearest value interpolation or uses a predefined lookup table.

If a discrete colormap is provided, only values present in the colormap are used for the output pattern, allowing the colormap to contain discrete device specific values.

Usage pattern = colormap(pattern, colormap, ...) applies the colormap to the pattern. The input pattern should have a typical range from 0 to 1. If the colormap is a LookupTable, the input pattern is scaled by the lookup table range.

Parameters

- pattern (numeric) – the pattern to be converted
- colormap (LookupTable|numeric|cell|enum) – colormap to apply. The way colormaps are applied depends on the colormap type:
- LookupTable – Uses phase, value and range properties of the `utils.LookupTable` object.
- numeric – assumes colormap is a vector of equally spaced values for the phase corresponding to pattern values between 0 and 1.
- cell – assumes colormap is a 2 element cell array. The first element is a vector with pattern values (range 0 to 1) and the second column is the corresponding output values.
- enum – 'pmpi', '2pi', 'bin' or 'gray' for output range between plus/minus pi, 0 to 2pi, binary, or grayscale (unchanged).

Optional named parameters

- 'inverse' (logical) – Apply inverse colormap. The output will have a typical range from 0 to 1. Not implemented for all colormap types.

hologram2bsc

otslm.tools.**hologram2bsc**(*pattern*, *varargin*)

Convert 2-D paraxial pattern to beam shape coefficients

This function uses the Optical Tweezers Toolbox BscPmParaxial class to calculate the beam shape coefficients using point matching.

Usage beam = hologram2bsc(pattern, ...) converts the pattern to a BSC beam. If pattern is real, assumes a phase pattern, else assumes complex amplitude.

Parameters pattern (numeric) – the pattern to convert

Optional named parameters

- incident (numeric) – Uses the incident illumination
- amplitude (numeric) – Specifies the amplitude of the pattern
- Nmax num – The VSWF truncation number
- polarisation [x,y] – Polarisation of the VSWF beam. Ignored if pattern is a NxMx2 matrix. Default `[1, 1i]`.
- radius (numeric) – Radius of lens back aperture (pixels). Default `min([size(pattern, 1), size(pattern, 2)])/2`.
- index_medium num – Refractive index of medium. Default `1.0`.
- NA num – Numerical aperture of objective. Default `0.9`.
- wavelength0 num – Wavelength of light in vacuum (default: 1)
- omega num – Angular frequency of light (default: 2*pi)
- beamData beam – Pass an existing Paraxial beam to reuse the pre-computed special functions. This requires the previous beam to have been generated with the keep_coefficient_matrix option.
- keep_coefficient_matrix (logical) – Calculate the inverse coefficient matrix and store it with the beam. This is slower for a single calculation but can be faster for repeated calculation. Default: false.

phaseblur

The *phaseblur()* function can be used to simulate how a pattern is affected by cross-talk between the pixels. For example, the following example shows how the effect of cross-talk on a checkerboard could be simulated. Results are shown in Fig. 4.18.

```
sz = [128, 128];

% Normal checkerboard
chk = otslm.simple.checkerboard(sz, 'spacing', 10);
chk = otslm.tools.finalize(chk);

% Blurred checkerboard
blur = otslm.tools.phaseblur(chk);

% Simulate far-field
im1 = otslm.tools.visualise(chk, 'trim_padding', true);
im2 = otslm.tools.visualise(blur, 'trim_padding', true);
```

otslm.tools.**phaseblur**(*pattern*, *varargin*)

Simulate pixel phase blurring

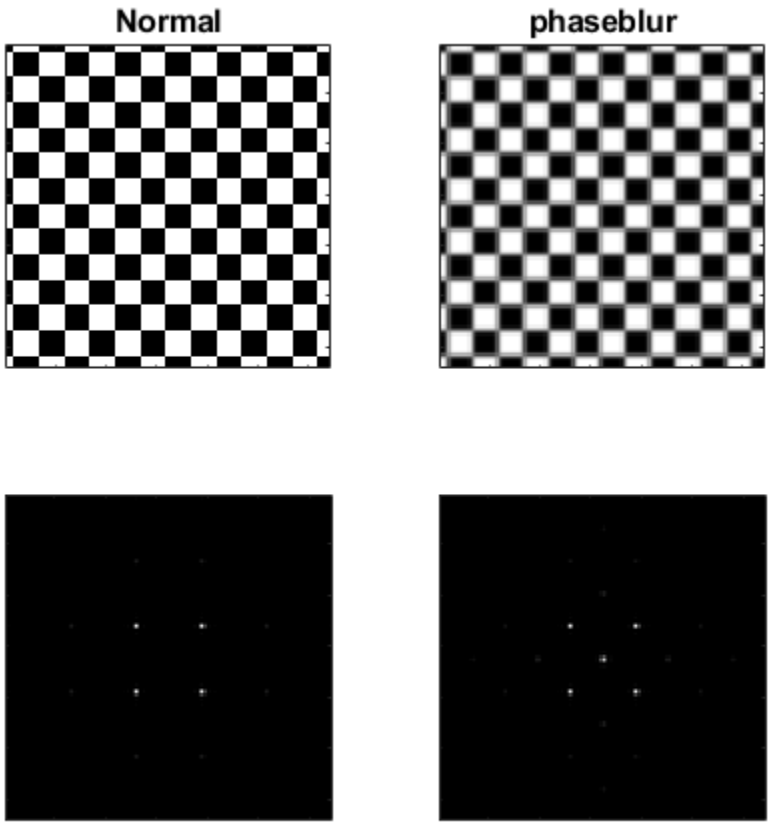

Fig. 4.18: Checkerboard pattern and simulated far-field (left) and the same checkerboard pattern after using the *phaseblur()* function (right).

Usage pattern = phaseblur(pattern, ...) applies Gaussian blur to the pattern.

Parameters

- pattern (numeric) – pattern to blur

Optional named arguments

- colormap – colormap to apply before/after blurring. For a list of valid values, see *colormap()*. (default: [])
- invmap (logical) – apply inverse colormap at end (default: true)
- sigma (numeric) – size of the Gaussian kernel

volume2hologram

otslm.tools.**volume2hologram**(*volume*, *varargin*)

Generate hologram from 3-D volume by un-mapping the Ewald sphere

This function is only the inverse of *hologram2volume()* when interpolation use_weight is disabled and there is no blurring.

Usage hologram = volume2hologram(volume, ...) calculates the overlap of the Ewald sphere with the volume and projects it to a 2-D hologram.

Parameters

- volume (numeric) – 3-D volume to un-map

Optional named arguments

- 'interpolate' (logical) – interpolate between the nearest two pixels in the z-direction. (default: True)
- 'padding' (numeric) – padding in the axial direction (default 0).
- 'focal_length' (numeric) – focal length in pixels (default: estimated from z)
- 'use_weight' (logical) – use weights when sampling interpolated pixels

See also *hologram2volume()* and `prop.FftEwaldForward`.

castValue

This function is used to convert from logical patterns to another data type, such as double or integer. It is mainly used by functions which create binary masks including `simple.aperture()` and `simple.step()`. When the resulting pattern is used for indexing another pattern, the output should be logical. However, if the resulting pattern corresponds to phase or amplitude values, this function can be used to perform the cast.

For example, to convert from a logical pattern to a pattern with two discrete integer levels, we could do

```
in = [false(3, 3), true(3, 3)];
out = otslm.tools.castValue(in, uint8([0, 27]));

% Check that the output class matches the desired class (uint8)
disp(class(out));
```

otslm.tools.**castValue**(*pattern*, *value*)

Convert from logical pattern to specified value range

Usage pattern = castValue(pattern, value)

Parameters

- pattern (logical) – the pattern to be converted
- value (numeric) – values for logical false and logical true pattern values. Should be either a 2 element vector `[false_value, true_value]`.or an empty array to leave the values as logical.

lensesAndPrisms

This function implements the lenses and prisms algorithm. The algorithm can be used to generate multiple spots by adding the complex amplitude of each pattern together. The location of each spot is controlled using a lens (for axial position) or a linear grating (prism, for radial positioning). The algorithm can be implemented in just a few lines of code using the toolbox:

```
sz = [128, 128];

lin1 = otslm.simple.linear(sz, [10, 5]);
len1 = otslm.simple.spherical(sz, sz(1)*2);

lin2 = otslm.simple.linear(sz, [-5, 15]);
len2 = otslm.simple.spherical(sz, -sz(1)*2.7);

pattern = otslm.tools.combine({lin1+len1, lin2+len2}, 'method', 'super');
```

However, this requires each pattern to be stored in memory until they can be combined in a single call to *`combine()`*. This can become a problem when hundreds of patterns need to be combined or if running on hardware with limited memory such as a GPU.

The *`lensesAndPrisms()`* function is a more memory efficient implementation of the above. Firstly, it performs the combination after each pattern has been created, removing the need to store all the component patterns. Further, instead of calculating multiple linear gratings and spherical lenses, the function calculates a single x, y and lens pattern and scales these patterns to generate a component pattern. In code, the operation performed is

```
% Locations of 2 spots (3x2 matrix)
xyz = [1, 2, 3; 4, 5, 6].';

for ii = 1:size(xyz, 2)
   pattern = pattern + exp(1i*2*pi* ...
      (xyz(3, ii)*lens + xyz(1, ii)*xgrad + xyz(2, ii)*ygrad));
end
```

To use the function, we simply need to pass in a matrix for the spot locations. Additionally, we could pass in weights for the different components or custom patterns for the lens and gratings. Example output is shown in Fig. 4.19.

```
sz = [100, 100];
xyz = [10, 5, 0.1; -3, -2, -0.2] ./ sz(1);
pattern = otslm.tools.lensesAndPrisms(sz, xyz.');
```

otslm.tools.**lensesAndPrisms**(*sz*, *xyz*, *varargin*)

Generates a hologram using the Lenses and Prisms algorithm

This function has the same affect as using multiple linear gratings and spherical lenses combined using otslm.tools.combine. The advantage of this function is a smaller memory footprint.

Usage pattern = lensesAndPrisms(sz, xyz, ...) The output pattern is in the range [0, 1). If supplied, the lens, xgrad and ygrad functions should have range [0, 1).

Parameters

- sz (size) – size of the pattern `[rows, cols]`

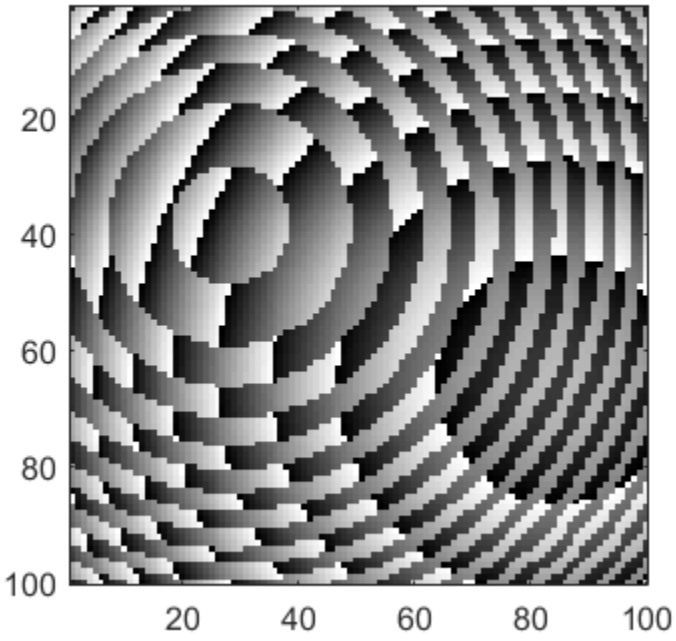

Fig. 4.19: Example output from `lensesAndPrisms()`.

- xyz (numeric) – 3xN matrix for target spot locations. Each column describes a different target, the first two rows describe the linear gradient and the final row describes the lens magnitude.

Optional named parameters

- 'lens' – pattern to use for lens. (default: xx^2 + yy^2 where xx and yy are from otslm.simple.grid)
- 'xgrad' – pattern to use for x gradient. (default: xx from otslm.simple.grid)
- 'ygrad' – pattern to use for y gradient. (default: yy from otslm.simple.grid)
- 'amplitude' – vector of amplitudes for each location
- 'gpuArray' (logical) – True if the result should be a gpuArray

make_beam

This function combines the amplitude and phase image into a single complex amplitude pattern. Mathematically, the function does

$$U = I \times A \times \exp(i\phi)$$

where I is the incident illumination, ϕ is the phase and A is the amplitude.

The function handles both 2-D patterns and 3-D volumes as well as a bunch of the size of the patterns and default values for any empty inputs.

`otslm.tools.`**`make_beam`**(*phase*, *varargin*)

Combine the phase, amplitude and incident patterns.

Usage U = make_beam(phase, ...) converts the phase pattern with a 2*pi range into a complex field amplitude. If phase is already complex the result is U = phase.

Parameters

- phase (numeric) – pattern to convert

Named parameters

- incident (numeric) – incident illumination.
- amplitude (numeric) – specify amplitude of the field. Only used when phase is a real matrix.

4.3.2 *prop* sub-package

The `otslm.tools.prop` package contains classes for propagating the fields. For simple beam propagation, see `tools.visualise()`. This documentation contains information on the Propagator base class and the propagator sub-classes. The package contains additional base classes for the common code shared between the forward and inverse methods.

For most propagators there are three methods that can be used to create a new instance. The class constructor creates a new instance where you specify all the options. The `simple` and `simpleProp` static functions create an instance of the propagator from an input pattern and return an output image or propagator depending on the method.

Propagator base class

class otslm.tools.prop.**Propagator**

Base class for field propagation methods.

Inherits from handle. This means that we can reuse the data block through multiple calls to propagate and easily split our code up into separate overload-able functions.

Abstract methods: out = propagate(in, ...) propagates the complex field amplitudes

Fft3Forward

class otslm.tools.prop.**Fft3Forward**(*sz*, *varargin*)

Propagate using forward 3-D fast Fourier transform

Methods

- Fft3Forward() – construct a new propagator instance
- propagate() – propagate the field forward using 3-D FFT

Properties

- data – Memory allocated for transform input
- padding – Padding around image
- size – Size of image
- roi – Region of interest within data for image
- roi_output – Region to crop output image after transformation

Static methods

- simple() – propagate the field with a simple interface
- simpleProp() – construct the propagator for input pattern

See also Fft3Inverse, FftForward and otslm.tools.visualise.

Fft3Forward(*sz*, *varargin*)

Construct a FFT propagator instance

FFT3FORWARD(sz, ...) construct a new propagator instance for the specified pattern size. sz must be a 3 element vector.

Optional named arguments:

- padding num | [xy, z] | [x, y, z] padding to add to edges of the image. Either a single number for uniform padding, two numbers for separate axial and radial padding, or three numbers for x, y and z padding. Default: ceil(sz/2)
- trim_padding bool if the output_roi should be set to remove the padding added before the transform. Default: false.
- gpuArray bool if true, allocates memory on the GPU and does the transform with the GPU instead of the CPU. Default: false.

static simple(*pattern*, *varargin*)

propagate the field with a simple interface

[output, prop] = simple(pattern, ...) construct a new FFT propagator and apply it to the pattern. Returns the propagated pattern and the propagator.

See also *simpleProp()* for named arguments.

static simpleProp(*pattern*, *varargin*)

Generate the propagator for the specified pattern.

prop = simpleProp(pattern, ...) construct a new propagator.

Optional named arguemnts:

- padding num | [num, num] Padding for transform. For details, see FftForward. Default: ceil(size(pattern)/2)
- trim_padding bool if padding should be trimmed from output. Default: true.
- gpuArray bool if we should use the GPU. Default: `isa(pattern, 'gpuArray')`

Fft3Inverse

class otslm.tools.prop.**Fft3Inverse**(*sz*, *varargin*)
Propagate using inverse 3-D fast Fourier transform

Methods

- Fft3Inverse() – construct a new propagator instance
- propagate() – propagate the field forward using 3-D FFT

Properties

- data – Memory allocated for transform input
- padding – Padding around image
- size – Size of image
- roi – Region of interest within data for image
- roi_output – Region to crop output image after transformation

Static methods

- simple() – propagate the field with a simple interface
- simpleProp() – construct the propagator for input pattern

See also Fft3Forward, FftInverse and otslm.tools.visualise.

Fft3Inverse(*sz*, *varargin*)
Construct a FFT propagator instance

FFT3INVERSE(sz, ...) construct a new propagator instance for the specified pattern size. sz must be a 3 element vector.

Optional named arguments:

- padding num | [xy, z] | [x, y, z] padding to add to edges of the image. Either a single number for uniform padding, two numbers for separate axial and radial padding, or three numbers for x, y and z padding. Default: ceil(sz/2)
- trim_padding bool if the output_roi should be set to remove the padding added before the transform. Default: false.
- gpuArray bool if true, allocates memory on the GPU and does the transform with the GPU instead of the CPU. Default: false.

static simple(*pattern*, *varargin*)
propagate the field with a simple interface

[output, prop] = simple(pattern, ...) construct a new FFT propagator and apply it to the pattern. Returns the propagated pattern and the propagator.

See also simpleProp for named arguments.

static simpleProp(*pattern*, *varargin*)
Generate the propagator for the specified pattern.

prop = simpleProp(pattern, ...) construct a new propagator.

Optional named arguemnts:

- padding num | [num, num] Padding for transform. For details, see FftForward. Default: ceil(size(pattern)/2)

- trim_padding bool if padding should be trimmed from output. Default: true.
- gpuArray (logical) – if we should use the GPU. Default: `isa(pattern, 'gpuArray')`

FftEwaldForward

class otslm.tools.prop.**FftEwaldForward**(*sz*, *varargin*)

Propagate using forward Ewald sphere and 3-D FFT. Inherits from `EwaldBase` and *Fft3Forward*.

Ewald surfaces are described in

> Gal Shabtay, Three-dimensional beam forming and Ewald surfaces, Optics Communications, Volume 226, Issues 16, 2003, Pages 33-37, https://doi.org/10.1016/j.optcom.2003.07.056.

and

> P.P. Ewald, J. Opt. Soc. Am., 9 (1924), p. 626

Methods

- FftEwaldForward() – construct a new propagator instance
- propagate() – propagate the field

Properties

- data – Memory allocated for transform input (3-D)
- padding – Padding around image [x, y, z]
- size – Size of image [x, y, z]
- roi – Region of interest within data for image
- roi_output – Region to crop output image after transformation
- focal_length – Focal length of the lens

Static methods

- simple() – propagate the field with a simple interface
- simpleProp() – construct the propagator for input pattern

See also FftEwaldInverse, FftForward and otslm.tools.visualise.

FftEwaldForward(*sz*, *varargin*)

Construct a Ewald sphere FFT propagator instance

FFTEWALDFORWARD(sz, ...) construct a new propagator instance for the specified pattern size. sz must be a 3 element vector.

Optional named arguments:

- focal_length num focal length of the lens in pixels. Default: min(sz/2).
- interpolate bool If the Ewald mapping should interpolate. Default: true.
- padding num | [xy, z] | [x, y, z] padding to add to edges of the image. Either a single number for uniform padding, two numbers for separate axial and radial padding, or three numbers for x, y and z padding. Default: ceil(sz/2).
- trim_padding bool if the output_roi should be set to remove the padding added before the transform. Default: false.

- gpuArray bool if true, allocates memory on the GPU and does the transform with the GPU instead of the CPU. Default: false.

static simple(*pattern*, *varargin*)
propagate the field with a simple interface

[output, prop] = simple(pattern, ...) construct a new propagator and apply it to the pattern. Returns the propagated pattern and the propagator.

See also simpleProp for named arguments.

static simpleProp(*pattern*, *varargin*)
Generate the propagator for the specified pattern.

prop = simpleProp(pattern, ...) construct a new propagator.

Optional named arguemnts:

- diameter num Diameter of the lens. Default: min(size(pattern))
- zsize num Depth of the FFT volume. Default: Calculated from focal_length and diameter
- focal_length num Set the focal length of the lens. Default: diameter/2 (unless NA is set)
- NA num Set the focal length via NA. Default: [] (i.e. defer to focal_length default)
- interpolate bool If the Ewald mapping should interpolate. Default: true.
- padding num | [xy, z] | [x, y, z] Padding for transform. For details, see FftEwaldForward. Default: ceil([size(pattern), zsize]/2)
- trim_padding bool if padding should be trimmed from output. Default: true.
- gpuArray bool if we should use the GPU. Default: isa(pattern, 'gpuArray')

FftEwaldInverse

class otslm.tools.prop.**FftEwaldInverse**(*sz*, *varargin*)
Propagate using inverse Ewald sphere and 3-D FFT. Inherits from `EwaldBase` and *`Fft3Inverse`*.

Ewald surfaces are described in

> Gal Shabtay, Three-dimensional beam forming and Ewald surfaces, Optics Communications, Volume 226, Issues 16, 2003, Pages 33-37, https://doi.org/10.1016/j.optcom.2003.07.056.

and

> P.P. Ewald, J. Opt. Soc. Am., 9 (1924), p. 626

Methods

- FftEwaldInverse() – construct a new propagator instance
- propagate() – propagate the field

Properties

- data – Memory allocated for transform input (3-D)
- padding – Padding around image [x, y, z]
- size – Size of image [x, y, z]
- roi – Region of interest within data for image
- roi_output – Region to crop output image after transformation

- focal_length – Focal length of the lens

Static methods

- simple() – propagate the field with a simple interface
- simpleProp() – construct the propagator for input pattern

See also FftEwaldForward, FftInverse and otslm.tools.visualise.

FftEwaldInverse(*sz*, *varargin*)
Construct a Ewald inverse FFT propagator instance.

FFTEWALDINVERSE(sz, ...) construct a new propagator instance for the specified pattern size. sz must be a 3 element vector.

Optional named arguments:

- focal_length num focal length of the lens in pixels. Default: `((min(sz(1:2))/2).^2 + sz(3).^2)/(2*sz(3))`
- interpolate bool If the Ewald mapping should interpolate. Default: true.
- padding num | [xy, z] | [x, y, z] padding to add to edges of the image. Either a single number for uniform padding, two numbers for separate axial and radial padding, or three numbers for x, y and z padding. Default: `ceil(sz/2)`
- trim_padding (logical) – if the output_roi should be set to remove the padding added before the transform. Default: false.
- gpuArray (logical) – if true, allocates memory on the GPU and does the transform with the GPU instead of the CPU. Default: false.

static simple(*pattern*, *varargin*)
propagate the field with a simple interface

[output, prop] = simple(pattern, ...) construct a new Ewald FFT propagator and apply it to the pattern. Returns the propagated pattern and the propagator.

See also simpleProp for named arguments.

static simpleProp(*pattern*, *varargin*)
Generate the propagator for the specified pattern.

prop = simpleProp(pattern, ...) construct a new propagator.

Optional named arguemnts:

- padding num | [xy, z] | [x, y, z] Padding for transform. For details, see *FftEwaldInverse()*. Default: `ceil(size(pattern)/2)`
- trim_padding bool if padding should be trimmed from output. Default: true.
- gpuArray bool if we should use the GPU. Default: isa(pattern, 'gpuArray')

FftForward

class otslm.tools.prop.**FftForward**(*sz*, *varargin*)
Propagate using forward 2-D fast Fourier transform

Methods

- FftForward() – construct a new propagator instance
- propagate() – propagate the field forward using 2-D FFT

Properties

- data – Memory allocated for transform input
- lens – Lens to be applied before transformation
- padding – Padding around image
- size – Size of image
- roi – Region of interest within data for image
- roi_output – Region to crop output image after transformation

Static methods

- simple() – propagate the field with a simple interface
- simpleProp() – construct the propagator for input pattern
- calculateLens() – lens function used by simple and FftInverse.simple.

See also FftInverse, Fft3Forward and otslm.tools.visualise.

`FftForward`(*sz*, *varargin*)

FFTFORWARD Construct a FFT propagator instance

FFTFORWARD(sz, ...) construct a new propagator instance for the specified pattern size. sz must be a 2 element vector.

Optional named arguments

- padding num | [num, num] – padding to add to edges of the image. Either a single number for uniform padding or two numbers to pass to `padarray()`. Default: `ceil(sz/2)`
- lens pattern – lens function to add to the transform. This can be useful for shifting the pattern in the axial direction. Pattern should have same size as sz + padding. The lens function should be a complex field amplitude. Default: [].
- trim_padding (logical) – if `output_roi` should be set to remove the padding added before the transform. Default: false.
- gpuArray (logical) – if true, allocates memory on the GPU and does the transform with the GPU instead of the CPU. Default: false.

`static simple`(*pattern*, *varargin*)

propagate the field with a simple interface

[output, prop] = simple(pattern, ...) construct a new FFT propagator and apply it to the pattern. Returns the propagated pattern and the propagator.

See also simpleProp for input arguments.

`static simpleProp`(*pattern*, *varargin*)

Generate the propagator for the specified pattern.

prop = simpleProp(pattern, ...) construct a new propagator.

Optional named arguemnts:

- axial_offset num Offset along the propagation axis Default: 0.0.
- NA num Numerical aperture for axial offset lens. Default: 0.1.
- padding num | [num, num] Padding for transform. For details, see FftForward. Default: ceil(size(pattern)/2)
- trim_padding bool if padding should be trimmed from output. Default: true.

- gpuArray bool if we should use the GPU. Default: isa(pattern, 'gpuArray')

FftInverse

class otslm.tools.prop.**FftInverse**(*sz*, *varargin*)

Propagate using inverse 2-D fast Fourier transform

Methods

- FftInverse() – construct a new propagator instance
- propagate() – propagate the field using 2-D inverse FFT

Properties

- data – Memory allocated for transform input
- lens – Lens to be applied after transformation
- padding – Padding around image
- size – Size of image
- roi – Region of interest within data for image
- roi_output – Region to crop output image after transformation

Static methods

- simple() – propagate the field with a simple interface
- simpleProp() – construct the propagator for input pattern
- calculateLens() – lens function used by simple and FftInverse.simple.

See also FftForward, Fft3Inverse and otslm.tools.visualise.

FftInverse(*sz*, *varargin*)

Construct a inverse FFT propagator instance

FFTINVERSE(sz, ...) construct a new propagator instance for the specified pattern size. sz must be a 2 element vector.

Optional named arguments:

- padding num | [num, num] padding to add to edges of the image. Either a single number for uniform padding or two numbers to pass to the *padarray* function. Default: ceil(sz/2)
- lens pattern lens function added after transformation. This can be useful for shifting the pattern in the axial direction. Pattern should have same size as sz + padding. The lens function should be a complex field amplitude. Default: [].
- trim_padding (logical) – if `output_roi` should be set to remove the padding added before the transform. Default: false.
- gpuArray (logical) – if true, allocates memory on the GPU and does the transform with the GPU instead of the CPU. Default: false.

static simple(*pattern*, *varargin*)

propagate the field with a simple interface

[output, prop] = simple(pattern, ...) construct a new inverse FFT propagator and apply it to the pattern. Returns the propagated pattern and the propagator.

See also simpleProp for named arguments.

static simpleProp(*pattern*, *varargin*)

Generate the propagator for the specified pattern.

prop = simpleProp(pattern, ...) construct a new propagator.

Optional named arguemnts

- axial_offset num Offset along the propagation axis Default: 0.0.
- NA num Numerical aperture for axial offset lens. Default: 0.1.
- padding num | [num, num] Padding for transform. For details, see FftForward. Default: ceil(size(pattern)/2)
- trim_padding bool if padding should be trimmed from output. Default: true.
- gpuArray bool if we should use the GPU. Default: isa(pattern, 'gpuArray')

FftDebyeForward

class otslm.tools.prop.**FftDebyeForward**(*sz*, *varargin*)

Propagate using forward 2-D FFT formulation of Debye integral. Inherits from *FftForward*.

This method is useful for simulating focusing of paraxial fields by high numerical aperture (NA) objectives. The method accounts for some of the polarisation and phase affects present in high NA focussing.

The method and conditions for obtaining acurate results are described in

> M. Leutenegger, et al., Fast focus field calculations, Optics Express Vol. 14, Issue 23, pp. 11277-11291 (2006) https://doi.org/10.1364/OE.14.011277

Methods

- FftForward() – construct a new propagator instance
- propagate() – propagate the field forward using 2-D FFT

Properties

- NA – Numerical aperture of lens
- radius – Radius of lens
- polarisation – Default polarisation for scalar input to propagate

Properties (inherited)

- data – Memory allocated for transform input
- lens – Lens to be applied before transformation
- padding – Padding around image
- size – Size of image
- roi – Region of interest within data for image
- roi_output – Region to crop output image after transformation

Static methods

- simple() – propagate the field with a simple interface
- simpleProp() – construct the propagator for input pattern
- calculateLens() – generates the lens required by the method

See also FftForward, Fft3Forward and OttForward

FftDebyeForward(*sz*, *varargin*)

Construct a FFT Debye forward propagator instance.

Usage obj = FftDebyeForward(sz, ...) construct a new propagator instance for the specified pattern size. sz must be a 2 element vector.

Optional named arguments

- polarisation [num, num] – X and Y polarisation to use when *propagate()* is called with only a single argument. Default: `[1.0, 0.0]`.
- NA (numeric) – Numerical aperture for axial offset lens. Default: 1.0.
- radius (numeric) – Radius of lens. Default: `min(sz)/2`.
- padding num | [num, num] – padding to add to edges of the image. Either a single number for uniform padding or two numbers to pass to `padarray()`. Default: `ceil(sz/2)`
- lens (complex) – lens pattern to add to the transform. This should typically be a result of *calculateLens()*. Pattern should have same size as sz + padding. The lens function should be a complex field amplitude. Default: `[]`.
- trim_padding (logical) – if `output_roi` should be set to remove the padding added before the transform. Default: false.
- gpuArray (logical) – if true, allocates memory on the GPU and does the transform with the GPU instead of the CPU. Default: false.

static calculateLens(*sz*, *NA*, *radius*, *z*)

Calculate lens function for FftDebyeForward.

The lens function is given by

where $\theta = \arcsin(r)$ and r is the normalized radius of the lens.

Usage lens = calculateLens(sz, NA, radius, z)

Parameters

- sz (size) – Size of the pattern `[rows, cols]`.
- NA (numeric) – Numerical aperture of lens. Assumes medium has refractive index of 1. NA should be adjusted if medium has different refractive index (NA/n_medium).
- radius (numeric) – Radial scaling factor for lens. (units: pixels).
- z (numeric) – Axial offset (units: inverse wavelength in medium).

propagate(*input*, *varargin*)

Propagate the input image

Usage output = propagate(input, ...) propagates the complex input image using 2-D FFT formulation of the Debye integral. Returns a NxMx3 matrix for the complex vector field at the focus.

Parameters

- input (numeric) – paraxial far-field image. Should either be a NxM or NxMx2 matrix. If the matrix is single channel, the method `polarisation` property is used.

static simple(*pattern*, *varargin*)

propagate the field with a simple interface

[output, prop] = simple(pattern, ...) construct a new FftDebye propagator and apply it to the pattern. Returns the propagated pattern and the propagator.

See also *simpleProp()* for input arguments.

static simpleProp(*pattern*, *varargin*)
Generate the propagator for the specified pattern.

prop = simpleProp(pattern, ...) construct a new propagator.

Optional named arguments:

- polarisation [num, num] – X and Y polarisation to use when *propagate()* is called with only a single argument. Default: `[1.0, 0.0]`.
- axial_offset num Offset along the propagation axis Default: 0.0.
- NA num Numerical aperture for axial offset lens. Default: 1.0.
- radius (numeric) – Radius of lens. Default: `min(size(pattern))/2`.
- padding num | [num, num] Padding for transform. For details, see FftForward. Default: ceil(size(pattern)/2)
- trim_padding bool if padding should be trimmed from output. Default: true.
- gpuArray bool if we should use the GPU. Default: isa(pattern, 'gpuArray')

OttForward

class otslm.tools.prop.**OttForward**(*sz*, *varargin*)
Propagate the field using the optical tweezers toolbox

Requires the optical tweezers toolbox (OTT).

Properties

- size – Size of input beam image
- beam_data – Beam with saved data for repeated computations
- Nmax – Nmax for VSWF
- polarisation – Polarisation of beam (jones vector)
- index_medium – Refractive index in medium
- NA – Numerical aperture
- wavelength0 – Wavelength in vacuum
- omega – Angular frequency of light

Static methods

- simple() – propagate the field with a simple interface
- simpleProp() – construct the propagator for input pattern

See also Ott2Forward and otslm.tools.visualise.

OttForward(*sz*, *varargin*)
OTTFORWARD Construct a new propagator instance

Usage prop = OttForward(sz, ...) construct a new propagator instance.

Parameters

- sz (size) – 2 element vector for the far-field size.

Optional named arguments

- pre_calculate bool If beam_data should be set at construction or at first use of propagate(). Defalut: true
- beam_data ott.Bsc Beam object to use instead of calculating the VSWF expansion. Incompatible with pre_calculate. Default: []
- Nmax num The VSWF truncation number
- polarisation [x,y] Default polarisation of the VSWF beam. Only used for single channel input images. Default `[1, 1i]`.
- radius (numeric) – Radius of lens aperture. Default: `min(sz)/2`.
- index_medium num Refractive index of medium
- NA num Numerical aperture of objective
- wavelength0 num Wavelength of light in vacuum (default: 1)
- omega num Angular frequency of light (default: 2*pi)

static simple(*pattern*, *varargin*)

SIMPLE propagate the field with a simple interface

[output, prop] = simple(pattern, ...) propagates the 2-D complex field amplitude *pattern* using the optical tweezers toolbox. Returns the beam and the propagator.

Additional named arguments are passed to Ott2Forward.

static simpleProp(*pattern*, *varargin*)

Generate the propagator for the specified pattern.

prop = simpleProp(pattern, ...) construct a new propagator.

Additional named arguments are passed to Ott2Forward.

Ott2Forward

class otslm.tools.prop.**Ott2Forward**(*sz*, *varargin*)

Propagate the field using the optical tweezers toolbox. Provides a wrapper to calculate the 2-D field after beam calculation.

Requires the optical tweezers toolbox (OTT).

Properties

- axis – Axis perpendicular to output image plane
- offset – Offset along axial direction
- field – Type of field to calculate
- output_size – Size of the output image
- range – Range of values to calculate field over

Inherited properties

- size – Size of input beam image
- beam_data – Beam with saved data for repeated computations
- Nmax – Nmax for VSWF
- polarisation – Polarisation of beam (jones vector)
- index_medium – Refractive index in medium

- NA – Numerical aperture
- wavelength0 – Wavelength in vacuum
- omega – Angular frequency of light

Static methods

- simple() – propagate the field with a simple interface
- simpleProp() – construct the propagator for input pattern

See also OttForward, FftForward and otslm.tools.visualise.

Ott2Forward(*sz*, *varargin*)

OTT2FORWARD Construct a new propagator instance

Usage prop = Ott2Forward(sz, ...) construct a new propagator instance.

Parameters

- sz (size) – size of the pattern in far-field `[rows, cols]`

Optional named arguments

- axis (enum) – 'x', 'y' or 'z' for axis perpendicular to image. Default: z.
- offset (numeric) – Offset along axial direction. Default: 0.0.
- field (enum) – Field to calculate. See `ott.Bsc.visualise()` for a list of valid parameters. Default: 'irradiance'.
- output_size [num, num] – Size of output image. Default: [80, 80]
- range [x, y] Range of points to visualise. Can either be a cell array { x, y }, two scalars for range [-x, x], [-y, y] or 4 scalars [x0, x1, y0, y1]. Default: [] (parameter is omitted, see ott.Bsc.visualise)
- pre_calculate (logical) – If beam_data should be set at construction or at first use of propagate(). Defalut: true

static simple(*pattern*, *varargin*)

SIMPLE propagate the field with a simple interface

[output, prop] = simple(pattern, ...) propagates the 2-D complex field amplitude *pattern* using the optical tweezers toolbox. Returns the field in the specified output plane and the propagator. The propagator contains the OTT.Bsc beam.

Additional named arguments are passed to Ott2Forward.

static simpleProp(*pattern*, *varargin*)

Generate the propagator for the specified pattern.

prop = simpleProp(pattern, ...) construct a new propagator.

Additional named arguments are passed to Ott2Forward.

RsForward

Warning: This method may be unstable.

class otslm.tools.prop.**RsForward**(*sz*, *distance*, *varargin*)

Propagate the field forward using Rayleight-Sommerfeld integral

Properties

- size – Size of the pattern
- distance – Distance to propagate pattern

Static methods

- simple() – propagate the field with a simple interface
- simpleProp() – construct the propagator for input pattern

See also FftForward, OttForward and otslm.tools.visualise.

RsForward(*sz*, *distance*, *varargin*)
RSFORWARD Construct a new propagator instance

obj = RsForward(sz, distance, ...) construct a propagator instance for the specified image size and propagation distance. distance must be a scalar, sz must be a 2 element vector.

static simple(*pattern*, *distance*, *varargin*)
SIMPLE propagate the field with a simple interface

output = simple(pattern, distance, ...) propagates the 2-D complex field amplitude *pattern* using the Rayleigh-Sommerfeld integral by the specified distance.

See also simple and RsForward.

static simpleProp(*pattern*, *distance*, *varargin*)
Generate the propagator for the specified pattern.

prop = simpleProp(pattern, ...) construct a new propagator.

See also simple and RsForward.

4.4 *utils* Package

The `otslm.utils` package contains functions for controlling, interacting with and simulating hardware.

Hardware (and simulated hardware) is represented by classes inheriting from the *`Showable`* and *`Viewable`* base classes. `Test*` devices are used for simulating non-physical devices, these are used mainly for testing algorithms. For converting from a `[0, 2*pi)` phase range to a device specific lookup table, the *`LookupTable`* class can be used. This package contains three sub-packages containing *imaging* algorithms, *calibration* methods and the *RedTweezers* interface.

4.4.1 LookupTable

class otslm.utils.**LookupTable**(*phase*, *value*, *varargin*)

Class representing the phase and pixel values of a lookup table.

Lookup tables can be used by Showable devices, otslm.tools.finalize and otslm.tools.colormap.

Methods

- load – load a human readable lookup table from a file
- save – save a human readable lookup table to a file.
- sorted – Returns a new lookup table sorted by phase
- resample – Re-sampled lookup table at the specified phases
- linearised – New re-sampled lookup with evenly spaced phase values
- valueMinimised – Arrange lookup table so values are ascending

Properties

- phase – phase values in lookup table [Nx1 matrix]
- value – pixel values in lookup table [NxM matrix]
- range – range of the lookup table (for phase based tables)

See also LookupTable, `otslm.tools.finalize()` and *Showable*

LookupTable(*phase*, *value*, *varargin*)

Construct a new LookupTable instance

Usage lt = LookupTable(phase, value, ...)

Parameters

- phase (numeric) – [Nx1] vector with phase values
- value (numeric) – [NxM] values to map phase too

Optional named arguments

- range (numeric) – The range of the look up table. This will typically be either 1 or 2*pi depending on if the lookup table is normalized or un-normalized. The actual ranges of phase values may be less or greater than this range.

linearised(*lt*, *numpts*, *varargin*)

Generates a new lookup table with evenly spaced values

Usage nlt = lt.linearised(numpts, ...) generates a resampled lookup table with evenly spaced values.

Parameters

- numpts (numeric) – number of evenly spaced values.

Optional named arguments

- range (numeric) – range to re-sample `[min, max]`. (default: `[min(lt.phase), max(lt.phase)]`).
- periodic (logical) – specifies if the range is periodic, if so, the end points count as the same point. Default false.

static load (*filename*, *varargin*)
Load a lookup table from a file

This is useful if you want to use the lookup table in another program. Otherwise, the recommended way to save a lookup table is by using the matlab save function.

Usage lt = LookupTable.load(filename, ...) loads the lookup table.

Parameters filename (char) – filename for lookup table to load

Optional named arguments

- 'channels' channels Array of columns numbers in input file 0 correspond to 0 in output. Negative values correspond to columns in reverse order.
- 'phase' column Column of input file taken as phase value. If omitted, i.e. [], assumes 0 to 2*pi linear phase range.
- 'oformat' format Output format string (default: uint8)
- 'format' format Input format handle (default @uint8)
- 'mask' mask Mask for input format (default: none)
- 'morder' order Array for order of bits in mask length should be 8, (0: zero bit, 1:8 mask bit, other: one bit). 1:8 is normal bit order, 8:-1:1 is reverse order.
- 'delim' delim Deliminator in input file
- 'nheaderlines' num Number of header lines in file

The number of channels in the output is determined by the length of the channels array. Each element in the channels array determines which column of the input file (starting at 1) is used to generate the channel data. A value of 0 means that this channel is empty.

The format argument specifies the input data type for each column. Data is read, cast to this type, and then the mask is applied. The output value is then calculated as a uint8 from the bits that were masked using the morder argument.

Example Load a 16-bit lookup table with values assigned to the first two channels. The input file has two columns, we use the second.

```
lookup_table = 'LookupTable.txt';
colormap = otslm.utils.LookupTable.load(lookup_table, ...
  'channels', [2, 2, 0], 'phase', [], 'format', @uint16, ...
  'mask', [hex2dec('00ff'), hex2dec('ff00')]);
```

resample (*lt*, *nphase*)
Generates a new lookup table re-sampled at the specified phases

Usage nlt = lt.resample(nphase) returns a new lookup table re-sampled at the specified phases. Values assigned to new phases correspond to the nearest values in the old table.

Parameters

- phase (numeric) – new phase values

save (*lt*, *filename*, *varargin*)
Save the lookup table to a human readable file

This is useful if you want to use the lookup table in another program. Otherwise, the recommended way to save a lookup table is by using the matlab save function.

Usage lt.save(filename, ...) saves the lookup table to file.

Parameters

- filename (char) – filename to save lookup table too.

Optional named arguments

- header (char) – header lines describing file contents. Default is a message about when the file was generated.
- cols (numeric) – specifies which columns of values will be written.
- format (enum) – type type of lookup table to write. All formats write the phase in the first column. This argument controls what is placed in additional columns. Currently supported types are:
- 8bit – write a single column of 8 bit integers
- 16bit – write a single column of 16 bit integers
- none – don't write any additional column
- multi – write one column for each value channel

sorted (*lt*)

Returns a new lookup table sorted by phase

Usage nlt = lt.sorted()

valueMinimised (*lt*, *valueRangeSz*)

Arranges lookup table so phase values are ascending but attempts to minimise change in linear index between steps.

Usage nlt = lt.valueMinimised(valueRangeSz) requires information about the size of each valueRange dimension (vector).

Todo: document parameters

See also otslm.utils.Showable.valueRangeSize()

4.4.2 imaging

This sub-package contains functions for generating an image of the intensity at the surface of a phase-only SLM in the far-field of the SLM.

To demonstrate how these function work, we can use the `TestFarfield` and `TestSlm` classes. From `examples/imaging.m`, the following code demonstrates how we can image the incident illumination on the device. Fig. 4.20 shows the incident illumination and output from the two imaging methods.

```
% Setup camera and slm objects
slm = otslm.utils.TestSlm();
slm.incident = otslm.simple.gaussian(slm.size, 100);
cam = otslm.utils.TestFarfield(slm);

% Generate 1-D profile
im = otslm.utils.imaging.scan1d(slm, cam, 'stride', 10, 'width', 10);

% Generate 2-D raster scan
im = otslm.utils.imaging.scan2d(slm, cam, ...
    'stride', [50,50], 'width', [50,50]);
```

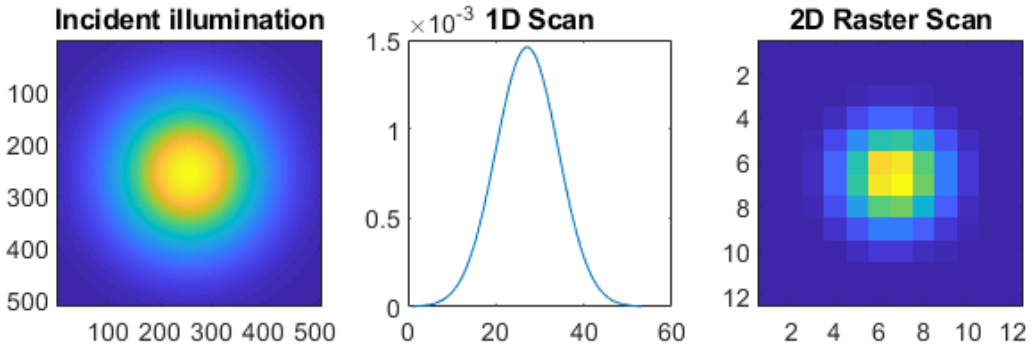

Fig. 4.20: Example output from imaging functions. (left) incident illumination. (middle) 1-D scan. (right) 2-D raster scan.

- *scan1d*
- *scan2d*

scan1d

`otslm.utils.imaging.`**`scan1d`**(*slm*, *cam*, *varargin*)

Scans a bar region across device.

This function scans a vertical stripe across the surface of the SLM with flat phase. Pixels outside this region are assigned a random phase, a checkerboard pattern or some other pattern in order to scatter light away from the zero order. The camera (or a photo-diode) should be placed in the far-field to capture only light from the flat phase region. This function generates a 1-D profile of the light on the SLM.

Usage im = scan1d(slm, cam, ...) scans a bar region across the device and returns a array representing the intensities at each location.

Parameters

- slm (`Showable`) – device to display pattern on. The slm.showComplex function is used to display the pattern. The pattern used for pixels outside the main region depends on the SLM configuration.
- cam (`Viewable`) – device viewing the display. This device could be a single photo-diode or the average intensity from all pixels on a camera.

Optional named arguments

- width (numeric) – width of the region to scan across the device
- stride (numeric) – number of pixels to step
- padding (numeric) – offset for initial window position
- delay (numeric) – number of seconds to delay after displaying the image on the SLM before imaging (default: [], i.e. none)
- angle (numeric) – direction to scan in (rad)
- angle_deg (numeric) – direction to scan in (deg)
- verbose (logical) – display additional information about run

See also `image2d()`.

scan2d

otslm.utils.imaging.**scan2d**(*slm*, *cam*, *varargin*)

Scans a 2-D region region across device.

This function is similar to *scan1d()* except it scans a rectangular region in a raster pattern across the surface of the SLM to form a 2-D image of the intensity.

Usage im = scan2d(slm, cam, ...) scans a bar region across the device and returns a matrix representing the intensities at each location.

Parameters

- slm (Showable) – device to display pattern on. The slm.showComplex function is used to display the pattern. The pattern used for pixels outside the main region depends on the SLM configuration.
- cam (Viewable) – device viewing the display. This device could be a single photo-diode or the average intensity from all pixels on a camera.

Optional named arguments

- width [x,y] (numeric) – width of the region to scan across the device
- stride [x,y] (numeric) – number of pixels to step
- padding [x0 x1 y0 y1] (numeric) – offset for initial window position
- delay (numeric) – number of seconds to delay after displaying the image on the SLM before imaging (default: [], i.e. none)
- angle (numeric) – direction to scan in (rad)
- angle_deg (numeric) – direction to scan in (deg)
- verbose (logical) – display additional information about run

See also image1d().

4.4.3 calibration

This sub-package contains functions for calibrating the device and generating a lookup-table. Most of these methods assume the SLM and camera are positioned in one of the configurations shown in Fig. 4.21.

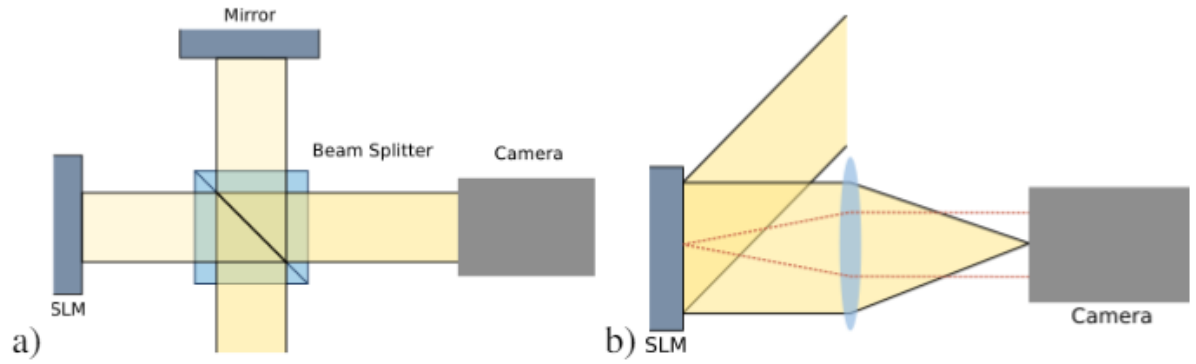

Fig. 4.21: Two different SLM configurations. (a) shows a Michelson interferometer setup. The SLM and reference mirror will typically be tilted slightly relative to each other. (b) shows a camera imaging the far-field of the device.

The sub-package contains several methods using these configurations. Some of the methods can be fairly unstable. The most robust methods, from our experience, are `smichelson` and `step`, both are described below. For information on the other methods, see the file comments and `examples/calibration.m`.

smichelson

This setup requires the device to be imaged using a sloped Michelson interferometer. The method applies a phase shift to half of the device and measures the change in fringe position as a function of phase change. The unchanged half of the device is used as a reference.

The easiest way to use this method is via the `otslm.ui.CalibrationSMichelson` graphical user interface.

The method takes two slices through the output image of the Viewable object. The slices should be perpendicular to the interference fringes on the SLM. The step width determines how many pixels to average over. One slice should be in the unshifted region of the SLM, and the other in the shifted region of the SLM. The slice offset, angle and width describe the location of the two slices. The `step_angle` parameter sets the direction of the phase step.

In order to understand the function parameters, we recommend using the `otslm.ui.CalibrationSMichelson` GUI with the `otslm.ui.TestMichelson` GUI. A possible configuration is shown in Fig. 4.22.

`otslm.utils.calibration.`**`smichelson`**(*slm*, *cam*, *varargin*)

Uses images from a sloped Michelson interferometer

Calculate the SLM lookup table using interference fringes on a sloped Michelson interferometer setup. Either the SLM or reference beam mirror must be sloped with respect to the illumination causing interference fringes in the output. By varying the phase on the device, the fringes can be made to move. This can be done on half the device allowing the other half to be used as a reference.

Usage lt = smichelson(slm, cam, ...) calibrate using the smichelson method.

Parameters

- slm (`Showable`) – device to generate the lookup table for.
- cam (`Viewable`) – device imaging the slm. The camera should be viewing the output of a Michelson interferometer, with the SLM on one arm and a mirror on the other. The mirror should be tilted slightly to create interference fringes on the camera.

Optional named parameters

- slice1_offset num – slice 1 distance from image centre
- slice1_width num – width of the slice 1 to average over
- slice2_offset num – slice 2 distance from image centre
- slice2_width num – width of the slice 2 to average over
- slice_angle num – angle for the slice (deg)

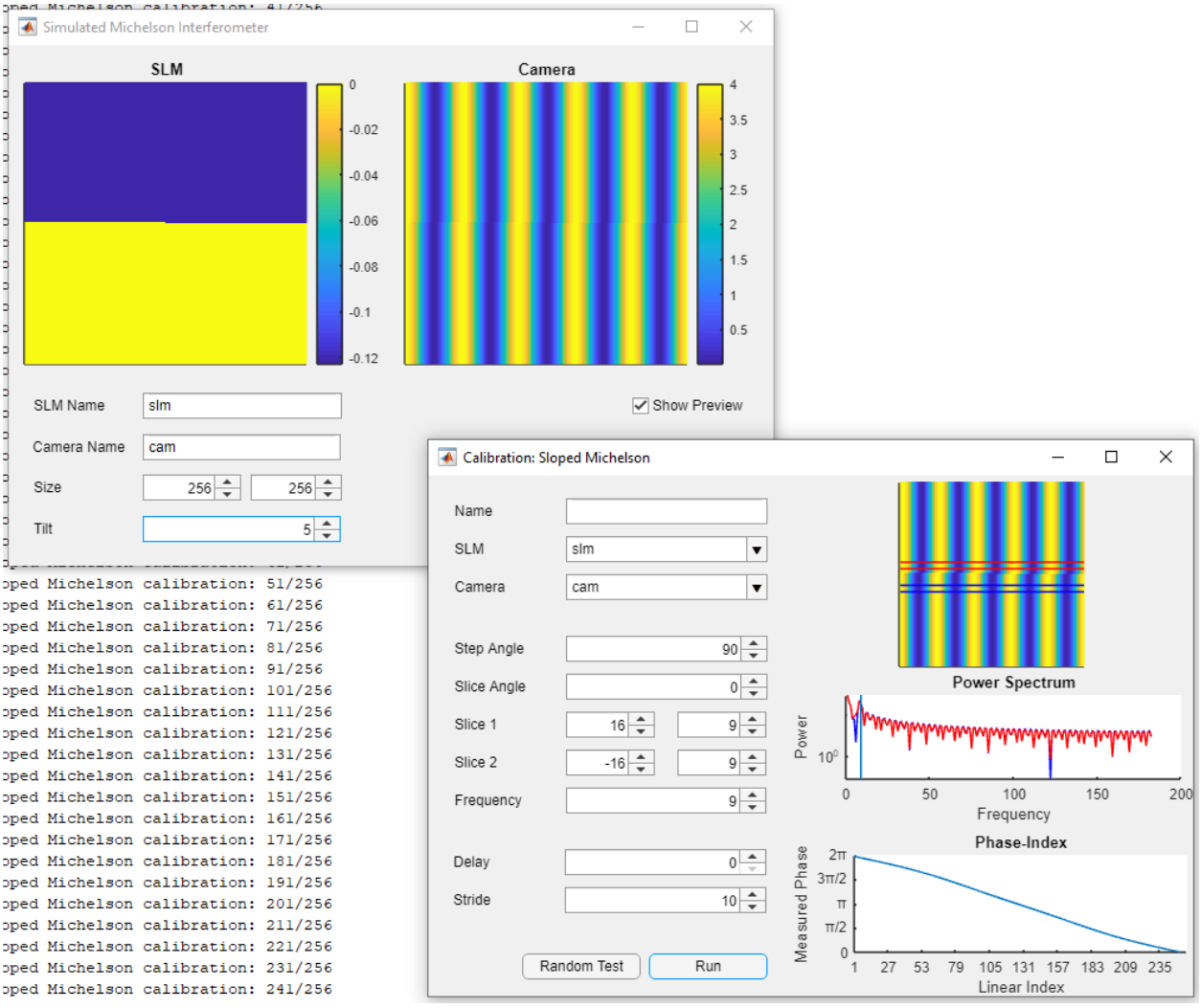

Fig. 4.22: `otslm.ui.CalibrationSMichelson` and : `otslm.ui.TestMichelson` used to demonstrate the smichelson calibration method.

- freq_index idx – index for frequency sample
- step_angle num – angle for the step function (deg)
- delay num – delay after displaying slm image
- stride num – number of linear indexes to step
- basevalue num – value to use for the first region
- verbose bool – display progress in console
- show_progress bool – show progress figure
- show_camera bool – show what the camera sees
- show_spectrum bool – show the 1-D Fourier spectrum of the images

step

This function requires the camera to be in the far-field of the device. The function applies a step function to the device, causing a interference line to appear in the far-field. The position of the interference line changes depending on the relative phase of the two sides of the step function. An extension to this function is *pinholes()* which uses two pinholes instead of a step function, allowing for more precise calibration.

The easiest way to use this method is via the `CalibrationStepFarfield` GUI. In order to understand the function parameters, we recommend using the `CalibrationStepFarfield` GUI with the `TestFarfield` GUI. An example configuration is shown in Fig. 4.23.

`otslm.utils.calibration.`**`step`**(*slm*, *cam*, *varargin*)

Applies a step function and looks at interference.

This function creates a step phase pattern with two regions. The far-field interference pattern of these regions contains a fringe which moves depending on the relative phase between the two regions.

The function uses a Fourier transform to determine the position of the interference fringe. The frequency for the Fourier transform is specified by the `freq_index` parameter. The width and angle parameters control the number of pixels to average over and the angle of the slice.

Usage lt = step(slm, cam, ...) calibrates using the step method.

Parameters

- slm (`Showable`) – device to generate the lookup table for.
- cam (`Viewable`) – device imaging the slm in the far-field.

Optional named arguments

- slice_offset num – slice distance from image centre
- slice_width num – width of the slice to average over
- slice_angle num – angle for the slice (deg)
- freq_index idx – index for frequency sample
- step_angle num – angle for the step function (deg)
- delay num – delay after displaying slm image
- stride num – number of linear indexes to step
- basevalue num – value to use for the first region
- verbose bool – display progress in console

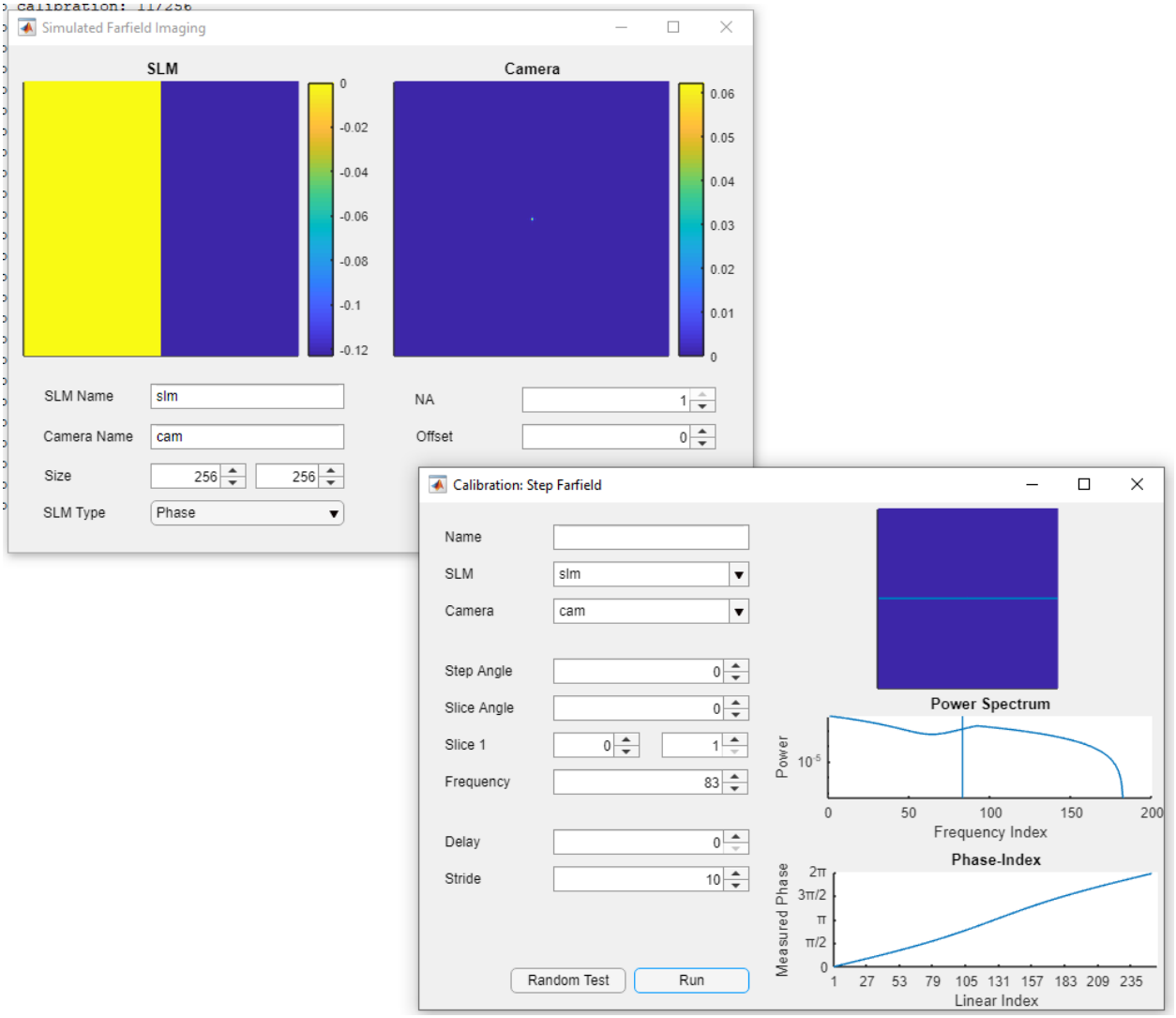

Fig. 4.23: `otslm.ui.CalibrationStepFarfield` and : `otslm.ui.TestFarfield` used to demonstrate the step calibration method.

- show_progress bool – show progress figure
- show_camera bool – show what the camera sees
- show_spectrum bool – show the 1-D Fourier spectrum of the images

pinholes

otslm.utils.calibration.**pinholes**(*slm*, *cam*, *varargin*)

Generates virtual pinholes with different phase.

Similar to step but looks at interference of two regions on different parts of the device allowing per-pixel or per-region calibration.

Usage lt = pinholes(slm, cam, ...) calibrates using the pinholes method.

Parameters

- slm (`Showable`) – device to generate the lookup table for.
- cam (`Viewable`) – device imaging the slm in the far-field.

Optional named arguments

- slice_offset num – slice distance from image centre
- slice_width num – width of the slice to average over
- slice_angle num – angle for the slice (deg)
- freq_index idx – index for frequency sample
- delay num – delay after displaying slm image
- stride num – number of linear indexes to step
- basevalue num – value to use for the first region
- radius num – radius of pinholes (pixels)
- verbose (logical) – display progress in console
- show_progress (logical) – show progress figure
- show_camera (logical) – show what the camera sees
- show_spectrum (logical) – show the 1-D Fourier spectrum of the images

checker

otslm.utils.calibration.**checker**(*slm*, *cam*, *varargin*)

Generate phase device lookup table using checkerboard pattern.

This method displays a checkerboard pattern on the device and looks at the intensity of the zero-th order. This may not be very effective if the device is not efficient or the device doesn't cover the full 2*pi phase range.

Usage lt = checker(slm, cam, ...)

Parameters

- slm (`Showable`) – device to generate the lookup table for.
- cam (`Viewable`) – device imaging the slm in the far-field.

Optional named arguments

- spacing (numeric) – size of checkerboard grid
- delay (numeric) – delay after updating slm
- stride (numeric) – number of linear indexes to step
- verbose (logical) – display progress in console
- show_progress (logical) – display progress of the method
- show_camera (logical) – show what the camera sees

linear

`otslm.utils.calibration.`**`linear`**(*slm*, *cam*, *varargin*)

Attempt to optimise diffraction from a linear grating.

This method does not produce a good estimate of the lookup table but is useful for generating a lookup table to maximise deflection into a particular direction or region.

> **Warning:** This method is experimental.

Usage lt = linear(slm, cam, . . .)

Parameters

- slm (`Showable`) – device to generate the lookup table for.
- cam (`Viewable`) – device imaging the slm in the far-field.

Optional named arguments

- grating str – type of grating to display on the device
- max_iterations num – maximum number of iterations to run
- show_progrerss bool – show progress of the method
- method str – method to use for optimisation
- dof num – number of degrees of freedom
- spacing num – diffraction grating spacing
- initial_cond str – initial condition

michelson

`otslm.utils.calibration.`**`michelson`**(*slm*, *cam*, *varargin*)

Uses images from a standard Michelson interferometer

Requires the SLM to be configured in Michelson interferometer setup where the screen is perpendicular to the incident beam and so is the reference beam mirror.

This method could be extended to allow calibration of individual pixels on the device but requires the uniform illumination.

Usage lt = michelson(slm, cam, . . .) calibrate the slm using the Michelson interferometer method.

Parameters

- slm (`Showable`) – device to generate the lookup table for.

- cam (`Viewable`) – device imaging the slm. The camera should be viewing the output of a Michelson interferometer, with the SLM on one arm and a mirror on the other.

Optional named arguments

- delay (numeric) – delay after displaying slm image (seconds).
- stride (numeric) – number of linear indexes to step
- verbose (logical) – display progress in console

4.4.4 RedTweezers

The RedTweezers sub-package provides classes for displaying patterns using OpenGL using the GPU or OpenGL enabled CPU. The main class is `RedTweezers` which provides methods for setting up the connection to the RedTweezers UDP server and sending OpenGL uniforms, textures and shaders programs to the UDP server. For these classes to work, you must have a running RedTweezers UDP server setup on your network. For example usage, see the *Using the GPU* example.

RedTweezers base class

class otslm.utils.RedTweezers.**RedTweezers**(*address*, *port*)

Interface to RedTweezers.

RedTweezers is a software package which calculates the hologram using OpenGL and directly displays the hologram on the hardware. This has the advantage over other methods that it does not require the pattern to be downloaded from the graphics hardware and re-uploaded for display on the hardware.

This class connects to RedTweezers via UDP. The RedTweezers library must be running for this to work.

For details and download link, see the RedTweezers CPC paper: https://doi.org/10.1016/j.cpc.2013.08.008

Methods

- RedTweezers – Construct an instance of this object
- sendCommand – Send a command (adds data block wrapper)
- updateAll – Resend all commands
- readGlslFile – Read a GLSL file into a character array

Properties

- udp_port – port of RedTweezers server
- live_update – True if property changes should be sent to RedTweeezers immediately, otherwise `updateAll` can be used to send properties at a later time.

RedTweezers properties

- vsync (logical) – Synchronise updating with monitor refresh rate
- window (celllnumeric) – Size of the window [x, y, width, height] or {'fullscreen', monitor_id} for fullscreen. Use `resizeWindow` to change the window size.

- network_reply (logical) – If True, requests RedTweezers server sends a reply.

See also RedTweezers, *Showable* and *PrismsAndLenses*.

RedTweezers (*address*, *port*)
RedTweezers construct a new RedTweezers interface

Usage rt = RedTweezers([address, port]) specifies a custom address/port.

Parameters

- address – IP address, passed to `udp()`. (default: '127.0.0.1').
- port – UDP port to connect to (default: 61556).

static readGlslFile (*filename*)
Read a GLSL file into a character array

Usage contents = RedTweezers.readGlslFile(filename) Returns a character vector with file contents.

Parameters

- filename (char) – GLSL file to read

sendCommand (*rt*, *cmd*)
Send a command string to the device

Usage rt.sendCommand(cmd) sends the command string to the device and adds the data block.

Parameters

- cmd (char) – command string to send to device

sendShader (*rt*, *shader*, *send*)
Send a shader to the device

Usage cmd = rt.sendShader(shader, [send])

Parameters

- shader (char) – shader character vector.
- send (logical) – If send is false, the command isn't sent. (default: nargout == 0)

sendTexture (*rt*, *id*, *texture*, *varargin*)
Send a texture blob to the device

Usage cmd = sendTexture(id, texture, [send, ...])

Paramters

- id – OpenGL uniform id to store texture at
- texture – The texture. Should be a 4xWxH in RGBA order. If the texture is a 3xWxH uint8 matrix, the function adds 255 for the A channel.
- send (logical) – If send is false, the command isn't sent. (default: nargout == 0)

Optional named arguments

- endian (enum) – 'L' or 'B' endian-ness of byte stream (for float)

sendUniform (*rt*, *id*, *values*, *send*)
Sends an array of namers to the device and renders the pattern

Usage cmd = rt.sendUniform(id, values, [send]) sends an array of numbers. The array of numbers will be stored in uniform register id. The first uniform in the program is id=0. Array length must be less than 200 elements. Returns the string for the command.

Parameters

- id (numeric) – OpenGL register to store values in
- values (numeric) – array of values to send
- send (logical) – If send is false, the command isn't sent. (default: nargout == 0)

updateAll (*rt*, *send*)

Resends all information to RedTweezers

Only sends set options (leaves others at RedTweezers defaults)

Usage rt.updateAll([send]) send all commands to the device.

cmd = rt.updateAll([send]) send all commands to the device and return the string that is sent.

Parameters

- send (logical) – If send is False, don't actually send the commands to the device. (default: nargout == 0)

Showable

class otslm.utils.RedTweezers.**Showable**(*varargin*)

RedTweezers interface for displaying pre-computed patterns. Inherits from *otslm.utils.Showable* and *RedTweezers*.

Loads a shader into RedTweezers for displaying images. This is roughly equivilant to the ScreenDevice class.

Showable(*varargin*)

Connects to RedTweezers and loads the Prisms and Lenses shader

rt = RedTweezers(...) connect to a running instance of RedTweezers. Default address is 127.0.0.1 port 61556.

rt = RedTweezers(address, port, ...) specifies a custom address/port.

Optional named arguments

- 'lookup_table' table – Lookup table to use for device Default lookup table is value_range{1} repeated for each channel.
- 'value_range' table – Cell array of value ranges Default is 256x3 for a RGB screen
- 'pattern_type' str – Type of pattern the device displays. Default is 'amplitude'. Can also be 'phase'.
- prescaledPatterns bool – Default value for prescaled argument in show. Default: false.

showRaw(*rt*, *img*)

Show the pattern on the device (update the texture)

rt.showRaw() clears the screen.

rt.showRaw(img) display an image on the screen. The image should have 1 or 3 channels.

Images must be single, double or unit8. Float images should be in range [0, 1), uint8 in range [0, 256).

PrismsAndLenses

class otslm.utils.RedTweezers.**PrismsAndLenses**(*varargin*)

Prisms and Lenses algorithm for RedTweezers. Inherits from *RedTweezers*.

Implements the Prisms and Lenses algorithm in an OpenGl shader.

See also PrismsAndLenses and *Showable*.

PrismsAndLenses (*varargin*)
Connects to RedTweezers and loads the Prisms and Lenses shader

rt = RedTweezers() connect to a running instance of RedTweezers. Default address is 127.0.0.1 port 61556.

rt = RedTweezers(address, port) specifies a custom address/port.

addSpot (*rt*, *varargin*)
Add a spot to the pattern

rt.addSpot(position, ...) declares a new spot at the specified position [x, y, z]. Uses *PrismsAndLensesSpot* to represent the spot.

Optional named parameters:

- 'oam' int – Vortex charge
- 'phase' float – Phase offset for the spot
- 'intensity' float – Intensity for the spot
- 'aperture' [x, y, R] – Position and radius of aperture
- 'line' [x, y, z, phase] – Direction, length and phase of line

removeSpot (*rt*, *index*)
Remove the specified spot from the pattern

rt.removeSpot() removes a spot from the end of the array.

Can also directly modify the Spot array

updateAll (*rt*, *send*)
Resends all information to RedTweezers

Only sends set options (leaves others at RedTweezers defaults) Does not send the shader.

If send is false, the command isn't sent. Default value for send is nargout == 0

PrismsAndLensesSpot

class otslm.utils.RedTweezers.**PrismsAndLensesSpot** (*varargin*)
Properties definition for a PrismsAndLenses spot. This class is for use with *PrismsAndLenses*.

Properties

- position – Position of spot [x; y; z]
- oam – Orbital angular momentum charge number (int)
- phase – Phase of the spot
- intensity – Intensity of the spot
- aperture – Aperture to define hologram within [x; y; radius]
- line – Line trap direction and phase [x; y; z; phase]

PrismsAndLensesSpot (*varargin*)
Declares a new spot for PrismsAndLenses

PrismsAndLensesSpot(position, ...) declares a new spot at the specified position [x, y, z].

Optional named parameters

- 'oam' int – Vortex charge
- 'phase' float – Phase offset for the spot
- 'intensity' float – Intensity for the spot
- 'aperture' [x, y, R] – Position and radius of aperture
- 'line' [x, y, z, phase] – Direction, length and phase of line

4.4.5 Base classes of showable and viewable objects

These abstract base classes define the interface expected by the various imaging, calibration and GUI functions/classes in the toolbox. You can not directly create instances of these classes, instead you must implement your own subclass or use one of the predefined subclasses, see *Physical devices* or *Non-physical devices*.

Showable

class otslm.utils.**Showable**(*varargin*)

Represents devices that can display a pattern. Inherits from `handle`.

Methods (abstract):

- showRaw(pattern) – Display the pattern on the device. The pattern is raw values from the device valueRange (i.e. colour mapping should already have been applied).

Methods:

- show(pattern) – Display the pattern on the device. The pattern type is determined from the patternType property.
- showComplex(pattern) – Display a complex pattern. The default behaviour is to call show after converting the pattern to the patternType of the device. Conversion is done by calling otslm.tools.finalize with for amplitude, phase target.
- showIndexed(pattern) – Display a pattern with integers describing entries in the lookup table.
- view(pattern) – Calculate the raw pattern.
- viewComplex(pattern) – Calculate the raw pattern from complex
- viewIndexed(pattern) – Calculate the raw pattern from indexed
- valueRangeNumel() – Total number of values device can display

Properties (abstract):

- valueRange – Values that the device patterns can contain. This should be a 1-d array, or cell array of 1-d arrays for each dimension of the raw pattern.
- patternType – Type of pattern, can be one of:
- 'phase' – Real pattern in range [0, 1]
- 'amplitude' – Real pattern in range [0, 1]

- 'complex' – Complex pattern, abs(value) <= 1
- size – Size of the device [rows, columns]
- lookupTable – Lookup table for show -> raw mapping

This is the interface that utility functions which request an image from the experiment/simulation use. For declaring a new display device, you should inherit from this class and define the abstract methods and properties described above. You can also override the other methods if needed.

See also Showable and otslm.utils.ScreenDevice.

Showable (*varargin*)
Constructor for Showable objects, provides options for default vals

Optional named arguments

- prescaledPatterns bool – Default value for prescaled argument in show. Default: false.

linearValueRange (*varargin*)
Generate an array of all possible device value combinations

linearValueRange('structured', true) generates a table with as many rows as valueRange has cells.

lienarValueRange() generates a table with a single column. values in each column of valueRange must be column unique.

show (*varargin*)
Method to show device type pattern

slm.show(...) with no pattern opens the window with an empty pattern.

slm.show(pattern, ...) displays the pattern after applying the color map. For phase based devices, the pattern should not be scalled by 2*pi (i.e. mod(pattern, 1)*2*pi should give the angle).

Optional named arguments:

– prescaled bool – If the pattern is already scalled by 2*pi. Default: false.

Additional arguments are passed to showRaw.

See also view, showRaw, showComplex.

showComplex (*pattern*, *varargin*)
Default function to display a complex pattern on a device

slm.showComplex(pattern, ...) Additional arguments are passed to showRaw.

showIndexed (*slm*, *pattern*, *varargin*)
Display a pattern described by linear indexes on the device

slm.showIndexed(pattern, ...) Additional arguments are passed to showRaw.

valueRangeSize (*idx*)
Calculate the size of the lookup table

view (*slm*, *pattern*)
Generate the raw pattern that is displayed on the device

im = slm.view(pattern) apply the modulo (if phase device) and apply the lookup table. Any nans remaining in the image are replaced by the first value in the lookup table.

viewComplex (*slm*, *pattern*)
Convert the complex pattern to a raw pattern

viewIndexed(*slm*, *pattern*)
Convert the indexed pattern to a raw pattern

Viewable

class otslm.utils.**Viewable**
Abstract representation of objects that can be viewed (cameras). Inherits from handle.

Methods (Abstract)

- view() Show an image from the device.

Methods

- viewTarget() Show an image of the target region from the device. The default behaviour is just to call view().
- crop(roi) Create a region of interest that is returned by viewTarget. Can have multiple regions.

Properties (Abstract)

- size Size of the device [rows, columns]
- roisize Size of the target region [rows, columns]

This is the interface that utility functions which request an image from the experiment/simulation use. For declaring a new camera, you should inherit from this class and define the view method.

crop(*cam*, *roi*)
Crop the image to a specified ROI

cam.crop([]) resets the roi to the full screen.

cam.crop(rect) creates a single region of interest described by the rect [xmin ymin width height].

cam.crop({rect1, rect2, ...}) creates multiple regions of interest described by separate rects.

viewTarget(*varargin*)
View the target, applies a ROI to the result of view()

im = viewTarget(...) acquire one or more target regions and return them to an cell array of images. Specify target regions using the crop function. If no output is requested, the image will be displayed in the current axes.

Optional named arguments

- roi array Specified which roi to return

See also otslm.Viewable.crop.

4.4.6 Physical devices

These classes are used to interact with hardware, including cameras, screens and photo-diodes.

- *ScreenDevice*
- *GigeCamera*
- *WebcamCamera*
- *ImaqCamera*

ScreenDevice

Represents a device controlled by a window on the screen. Devices including some digital micro-mirror devices and spatial light modulators can be connected as additional monitors to the computer and can be controlled by displaying an image on the screen. This class provides an interface for controlling a Matlab figure, making sure the window has the correct size, and ensures the window is positioned above other windows on the screen.

To use the *ScreenDevice* class, you need to specify which screen to place the window on and how large the screen should be. The *ScreenDevice* can either occupy the whole screen, for example

```
slm = otslm.utils.ScreenDevice(2, 'pattern_type', 'phase');
```

would create a full-screen window on display number 2; or the class can create a window of a specified size, for example

```
slm = otslm.utils.ScreenDevice(1, 'pattern_type', 'amplitude', ...
      'size', [512, 512], 'offset', [100, 100]);
```

creates a 512x512 window positioned 100 pixels from the bottom and 100 pixels from the left of screen 1. Size must always be positive numbers, while offset can be negative to indicate a position relative to the right/top of the screen, for details, see figure Fig. 4.24.

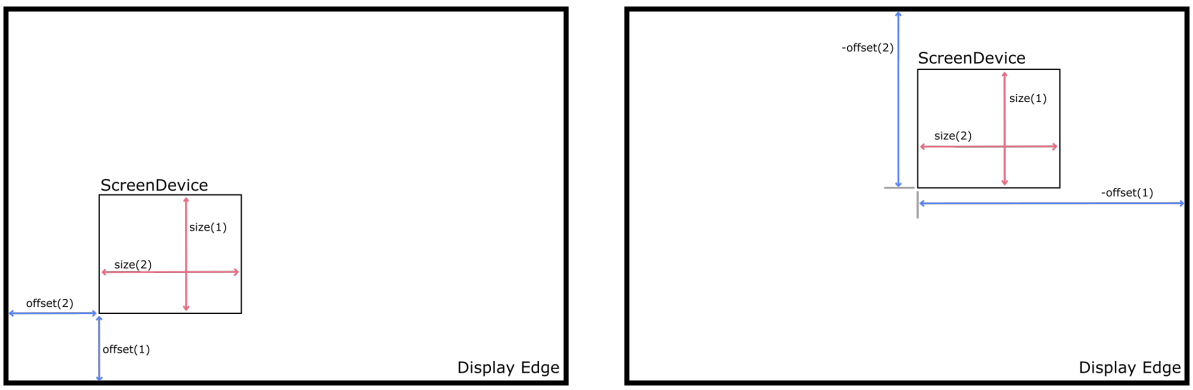

Fig. 4.24: Position and size of the screen device and display showing how they relate to the *ScreenDevice* `offset` and `size` input parameters. (left) a *ScreenDevice* positioned relative to the bottom left corner, `offset` is positive. (right) a *ScreenDevice* positioned relative to the top right corner, `offset` is negative.

The `pattern_type` argument specifies if the input pattern to the `show` methods should be a phase, amplitude or complex amplitude pattern. To create a window that is not full-screen, we can simply pass `false` as the full-screen argument and set the corresponding target window size and position offset.

To display a pattern on the device for 10 seconds, we can use

```
pattern = otslm.simple.linear(slm.size, 50);
slm.show(pattern);
pause(10);
slm.close();
```

This configuration assumes the pattern has not yet been passed to the finalize function (i.e. for a linear grating with a spacing of 50 pixels, the pattern should be in the range 0 to 1 and not 0 to 2pi). If you are using pre-scaled patterns (in the range 0 to 2pi), you can set the `prescaledPatterns` optional parameter in the constructor for the *ScreenDevice* to `true`:

```
slm = otslm.utils.ScreenDevice(scid, 'pattern_type', 'phase', ...
  'prescaledPatterns', true);
```

To display a sequence of frames on the device, you can use multiple calls to `ScreenDevice.show()`. This will apply the colour-map during the animation, which can be time consuming and reduce the frame rate. An alternative is to pre-calculate the animation frames. To do this, we generate a struct which can be passed to the `movie` function:

```
% Generate images first
patterns = struct('cdata', {}, 'colormap', {});
for ii = 1:100
  patterns(ii) = im2frame(otslm.tools.finalize(otslm.simple.linear(slm.size, ii), ...
      'colormap', slm.lookupTable));
end

% Then display the animation
slm.showRaw(patterns, 'framerate', 100);
slm.close();
```

Showable classes have multiple methods for showing patterns on the device. The `showRaw` method takes patterns that are already in the range of values suitable for the device. The `show` function converts the specified pattern into the device value range (by applying, for example, a colour-map or modulo to the pattern). The type of input to the show function should match the `patternType` property, for *`ScreenDevice`* objects, `patternType` is set from the `pattern_type` parameter in the constructor. If `patternType` is amplitude, the input to show is assumed to be a real amplitude pattern, if `patternType` is phase, the input is assumed to be a phase pattern. The `showComplex` function uses *`otslm.tools.finalize()`* to convert the complex amplitude to a phase or amplitude pattern (depending on the value for `patternType`), before calling `show` to display the pattern on the device. Further details can be found in the documentation for the *`Showable`* base class.

To setup the lookup table which is applied by `show`, we can load a lookup table from a file and pass it in on construction. If you don't yet have a lookup table, you can use one of the calibration functions, see *calibration*. As an example, to load a lookup table specified by a filename `fname` we could use the following:

```
lookup_table = otslm.utils.LookupTable.load(fname, ...
  'channels', [2, 2, 0], 'phase', [], 'format', @uint16, ...
  'mask', [hex2dec('00ff'), hex2dec('ff00')], 'morder',  1:8);
```

This assumes the file has 2 columns, we ignore the first and split the second into the lower 8 bits and upper 8 bits. The lookup table has 3 channels, the first two channels have values from the second column in the file, the third channel is all zeros. The format for the input is `uint16`, we apply a `mask` to this input for each column and we specify the order of the bits from least significant to most significant (`morder`). The phase isn't specified in this lookup table, so we assume it is linear from 0 to 2pi. For further details, see *`LookupTable`*.

To use this lookup table for the *`ScreenDevice`*, we simply pass it into the constructor:

```
slm = otslm.utils.ScreenDevice(1, ...
    'lookup_table', lookup_table, ...
    'pattern_type', 'phase');
```

class otslm.utils.**ScreenDevice**(*varargin*)

Represents a device controlled by a window on the screen Inherits from *`Showable`*.

Useful for displaying images on SLMs and DMDs connected as a screen on the computer running Matlab.

The actual target device size may be smaller than the size reported by the device.

See also ScreenDevice, show, otslm.utils.Showable.

ScreenDevice(*varargin*)

ScreenDevice Construct a new instance of the screen device.

screen = ScreenDeivce(device_number, ...) creates a new screen device for the specified physical device. Patterns are assumed to be amplitude based, value range is RGB.

Optional named parameters:

- 'size' [r,c] – Size of the device within the window Default: *[]*, (i.e. *slm.device_size*)
- 'offset' [r,c] – Offset within the window. Negative values are offset from the top of the screen. Default: *[0, 0]*
- 'lookup_table' table – Lookup table to use for device Default lookup table is value_range{1} repeated for each channel.
- 'value_range' table – Cell array of value ranges Default is 256x3 for a RGB screen
- 'pattern_type' type – Type of pattern the device displays. Default is amplitude.
- 'fullscreen' bool – Default value for showRaw/fullscreen
- 'prescaledPatterns' bool If the pattern is already pre-scaled.

close()
Close the window used to control the device

showRaw(*varargin*)
Show the window and (optionally) display an image

showRaw() display a blank screen.

showRaw(img) display an image on the screen. The image should have 1 or 3 channels.

showRaw(frames) displays frames using the movie command. frames should be an array of frames generated by im2frame. showRaw(frame, 'framerate', rate) specifies the frame rate. showRaw(frame, 'play', times) specifies time to play (see movie).

GigeCamera

Showable wrapper for cameras using the `gigecam` interface. This class uses the `snapshot` function to get an image from the device. The `gigecam` device is stored in the `device` property of the class.

class otslm.utils.**GigeCamera**(*varargin*)
Connect to a gige camera connected to the computer Inherits from *Viewable*.

Properties

- device – gige camera object
- size – size of the camera output image
- Exposure – camera exposure setting

See also GigeCamera

GigeCamera(*varargin*)
Connect to the camera

cam = GigeCamera(device_id) connects to the specified GIGE camera. device_id is passed to the gigecam function.

WebcamCamera

Showable wrapper for windows web-cameras. Uses the `videoinput('winvideo', ...)` function to connect to the device. This class uses the `getsnapshot` function to get an image from the device. The `videoinput` device is stored in the `device` property of the class.

Note: This class currently doesn't inherit from *ImaqCamera* but is likely to in a future release of OTSLM.

class otslm.utils.**WebcamCamera**(*varargin*)

Connect to a webcam camera connected to the computer. Inherits from *Viewable*.

Properties

- size – resolution of the device
- device – videoinput device for the camera

This call can be used to create a otslm.utils.Viewable instance for a videoinput source. This requires the Image Acquisition Toolbox.

See also WebcamCamera, *GigeCamera* and *ImaqCamera*.

WebcamCamera(*varargin*)

Connect to the camera

cam = WebcamCamera(device_id) connect to the specified webcam camera. For the device id, imaqhwinfo.

If the device you are interested in is not listed, try resetting/clearing all devices with imaqreset.

ImaqCamera

Showable wrapper for image acquisition toolbox cameras. Uses the `videoinput(...)` function to connect to the device. This class uses the `getsnapshot` function to get an image from the device. The `videoinput` device is stored in the `device` property of the class.

class otslm.utils.**ImaqCamera**(*varargin*)

Connect to a image acquisition toolbox (imaq) camera Inherits from *Viewable*

This call can be used to create a otslm.utils.Viewable instance for a videoinput source. This requires the Image Acquisition Toolbox.

Properties

- device – imaq object for the camera

See also ImaqCamera.

ImaqCamera(*varargin*)

Connect to the camera

cam = ImaqCamera(adaptor, id, ...) conntect to the specified webcam camera. For the device id, imaqhwinfo.

For cameras that support multiple formats, a 'format' named argument can be supplied with the format to use.

4.4.7 Non-physical devices

The `utils` package defines several non-physical devices which can be used to test calibration or imaging algorithms. The *`TestDmd`* and *`TestSlm`* classes are *`Showable`* devices which can be combined with the *`TestFarfield`* or *`TestMichelson`* Viewable devices. These *`Showable`* devices implement the same functions as their physical counter-parts, except they store their output to a `pattern` property. The *`Viewable`* devices require a valid *`TestShowable`* instance and implement a view function which retrieves the `pattern` property from the *`Showable`* and simulates the expected output.

For example usage, see the examples in the *imaging* section.

TestDmd

class otslm.utils.**TestDmd**(*varargin*)

Non-physical dmd-like device for testing code. Inherits from *`TestShowable`*.

This class can be used as a non-physical Showable device in simulating a binary amplitude device, such as a digital micro-mirror device.

When `showRaw` is called, the function calculates the pattern by applying `rpack` using the `otslm.tools.finalize()` method and sets the `pattern` property with the computed pattern. The incident illumination is added to the output. To change the incident illumination, either set a different pattern on construction or change the property value.

Properties

- incident (complex) – incident illumination profile. Must be the same size as the device.
- pattern (comples) – pattern generated by the `showRaw` method. This pattern is is the complex amplitude after multiplying by the incident illumination and applying `rpack`. The `rpack` operation means that this pattern is larger than the device, with extra padding added to the corners.

Constant properties

- size (size) – device resolution (pixels) [rows, columns]
- valueRange – range of raw device values (fixed: `{[0, 1]}`)
- patternType – type of pattern for device (fixed: `'amplitude'`)
- lookupTable – mapping between gray-scale and binary values. (fixed)

See also TestDmd, *`TestSlm`* and *`TestFarfield`*.

TestDmd(*varargin*)

Create a new virtual DMD object for testing

Usage slm = TestDmd(...) create a virtual binary amplitude device.

Optional named arguments

- size [row, col] – Size of the device (default: [512,512])
- incident im – Incident illumination (default: [])

TestSlm

class otslm.utils.**TestSlm**(*varargin*)

Non-physical slm-like device for testing code. Inherits from *TestShowable*.

The `showRaw` function applies the inverse of the lookup table, converts from phase to a complex amplitude and assigns the result to the `pattern` property.

Properties

- pattern (complex) – pattern generated by the `showRaw` method. This pattern is the complex amplitude after multiplying by the incident illumination and is the same size as the device.
- incident (complex) – incident illumination.

Constant properties

- size (size) – device resolution in pixels [rows, columns]
- valueRange – range of raw device values (default: `{0:255}`)
- patternType – type of pattern for device (fixed: `'phase'`)
- lookupTable – Lookup table for the device. Defaults to a linear mapping of 0 to 2pi to the discrete colour levels of the device.

See also TestSlm, *TestDmd* and *TestFarfield*.

TestSlm(*varargin*)

Create a new virtual SLM object for testing

Usage slm = TestSlm(...) creates a device with a linear phase lookup table from 0 to 2*pi.

Optional named arguments

- size (size) – size of the device (`[rows, cols]`).
- incident (complex) – incident illumination
- lookup_table – lookup table for colormap
- value_range (cell) – cell array with channel values for raw pattern. Default: `{0:255}`. Use `{0:255, 0:255, 0:255}` for three channel device with 256 levels on each channel.

TestFarfield

class otslm.utils.**TestFarfield**(*varargin*)

Non-physical camera for viewing TestShowable objects Inherits from *Viewable*.

Calculates the paraxial far-field of the *TestShowable* object. The view method calls `otslm.tools.visualise()` and calculates the intensity of the resulting image (`abs(U)^2`).

Note: This class may change in future versions to use a propagator instead of `otslm.tools.visualise()`.

Properties

- size – size of the output image
- showable – the Showable object that this class is linked to
- NA – numerical aperture of the lens (passed to `otslm.tools.visualise()`).
- offset – offset from the focal plane of the lens (passed to `otslm.tools.visualise()`).

Inherited properties

- roisize – (Viewable) size of the regions of interest
- roioffset – (Viewable) offsets for the regions of interest
- numroi – (Viewable) number of regions of interest

See also TestFarfield, *TestMichelson*, *TestSlm*.

TestFarfield(*varargin*)

Construct a new TestFarfield looking at a TestShowable object

Usage obj = TestFarfield(showable, ...)

Parameters

- showable (*TestShowable*) – linked showable device.

Optional named arguments

- NA (numeric) – Numerical aperture of lens (default: 1.0)
- offset (numeric) – Offset from focal plane of lens (default: 0.0)

TestMichelson

class otslm.utils.**TestMichelson**(*showable*, *varargin*)

Non-physical representation of Michelson interferometer. Inherits from *Viewable*.

The interferometer consists of two arms, a reference arm with a mirror and a device arm with a `Showable` device such as a SLM or DMD The TestMichelson simulates a *TestShowable* device placed in one arm, and calculates the interference pattern between the reference arm and the test arm.

The `view` function gets the current `pattern` from the `TestShowable` device and calculates the interference. The device supports adding a tilt between the reference arm and the Showable arm. This can be useful for testing `calibration.smichelson()`.

Properties

- tilt – Angle to tilt the showable device with respect to the reference beam. Applies a exp(2*pi*i*linear*tilt) grating.

Properties (read-only)

- size – Size of the output image (same as Showable)
- showable – The TestShowable device

See also TestMichelson, *TestShowable* and *TestFarfield*.

TestMichelson(*showable*, *varargin*)

Create the new interferometer-like device

Usage obj = TestMichelson(showable, ...) construct a new Michelson interferometer to view the TestShowable device.

Parameters

- showable (*TestShowable*) – the device to link

Optional named arguments

- tilt (numeric) – tilt factor (default: 0.0)

TestShowable

class otslm.utils.**TestShowable**

Non-physical showable device for testing implementation. Inherits from *Showable*.

This is an abstract class defining a single abstract property, `pattern`, containing the pattern currently displayed on the device. For implementations see *TestDmd* and *TestSlm*.

Properties (Abstract)

- pattern – complex valued pattern currently displayed on the device.

See also *Showable*, *TestSlm* and `TestCamera`.

4.5 *ui* Package

The UI sub-package contains graphical user interfaces for exploring the toolbox functionality. The sub-package contains the `Launcher` GUI which provides a list of components and a brief description of their function. The rest of the sub-package is split between `simple`, `utils`, `tools`, `iter`, and `examples` sub-packages providing interfaces to the OTSLM core packages and examples of how the UI can be combined. The UI sub-package also contains a *otslm.ui.support* sub-package with common code used by the GUIs.

This section provides information required to extend the Launcher, or other GUI windows as well as a brief *overview of the other GUI components*. For details on the functions the GUIs represent, see the corresponding package documentation: *iter Package*, *tools Package*, *simple Package* or *utils Package*. For details on how to use the GUIs, see *Getting Started* and *Examples*.

Contents

4.5.1 Launcher

otslm.ui.**Launcher**

The launcher consists of two layers: the category list and the application list. The application list is populated when the user selects a category. Details about the programs are specified in the `CategoryListBoxValueChanged` function and `*Data` functions.

Specifying application names

Application names are specified in the `CategoryListBoxValueChanged` function. To add a new application, extend the `ItemsData` and `Items` fields of the `ApplicationListBox` for the category you wish to place the app in. The `ItemsData` field is used in the `*Data` function (see below) to get the application name and description.

Program name, description and launch command

Information about each of the programs is defined in the `*Data` functions, one function for each sub-package: `ExampleData`, `IterativeData`, `ToolsData`, `UtilitiesData` and `SimpleData`.

These functions return a struct with the fields `Name`, `Description` and `AppName` for the user-readable name, description and the Matlab application name to launch. The value returned depends on the current value for the `ApplicationListBox` list box. In order to extend the applications list, simply add a new case to the switch for the new application and set the corresponding values in the `data` struct, for example:

```
data.Name = 'Mixing Two Beams';
data.Description = ['This example shows how to generate a phase only diffraction ' ...
    'grating to split a beam into two independently controllable spots.'];
data.AppName = 'otslm.ui.examples.MixingTwoBeams';
```

4.5.2 Simple GUI overview

Most GUIs are split into 4 main sections, shown in Fig. 4.25

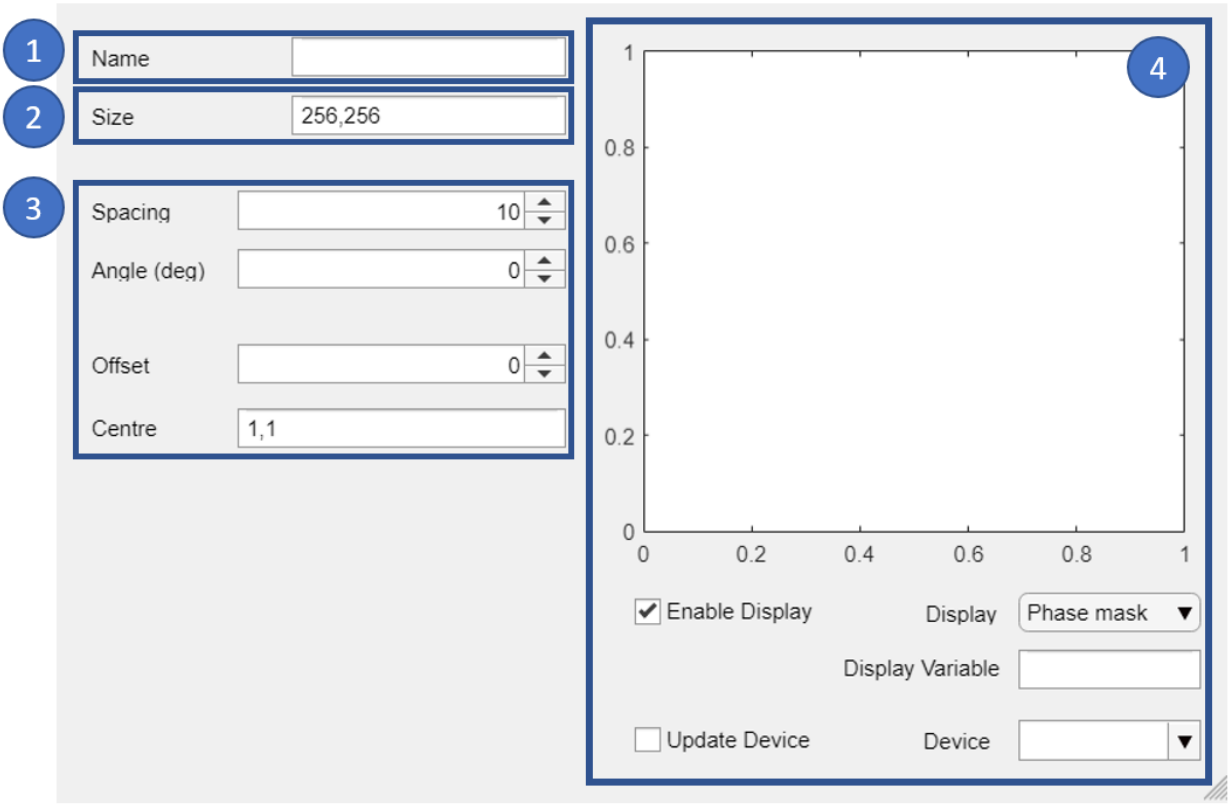

Fig. 4.25: Overview of *ui.simple.linear* graphical user interface. The layout consists of: (1) Output variable name; (2) Size of pattern (mostly used on `ui.simple.*` GUIs); (3) Controls for the method; and (4) Pattern preview window.

When the window launches it will search the base workspace for variables names and *`otslm.utils.Showable`* devices which can be used for displaying the pattern (see *`otslm.ui.support.populateDeviceList()`*).

Methods which updated as soon as the user changes a value will have most of the implementation contained in a callback function. For `ui.simple.linear`, this is done in `otslm.ui.support.patternValueChanged()`.

The content of this function involves first getting the inputs from the user and converting the strings to variables:

```
% Get the UI fileds for generating the pattern
name = app.NameEditField.Value;
sz = evalin('base', ['[', app.SizeEditField.Value, ']']);
spacing = app.SpacingSpinner.Value;
angle_deg = app.AngledegSpinner.Value;
offset = app.OffsetSpinner.Value;
centre = evalin('base', ['[', app.CentreEditField.Value, ']']);
```

The function then calls the OTSLM method:

```
% Generate the pattern
pattern = otslm.simple.linear(sz, spacing, ...
    'centre', centre, 'angle_deg', angle_deg);
pattern = pattern + offset;
```

And finally, calls the *otslm.ui.support.simplePatternValueChanged()* helper handle updating the preview window, saving the result to the workspace and updating the device.

```
% Offload to the base class (sort of...)
otslm.ui.support.simplePatternValueChanged(name, pattern, ...
    app.DeviceDropDown.Value, app.UpdateDeviceCheckBox.Value, ...
    app.EnableDisplayCheckBox.Value, app.UIAxes, ...
    app.DisplayDropDown.Value, app.DisplayVariableEditField.Value);
```

Most functions will have a public `updateView` function which can be used by other GUI windows to force an update to window after values have changed.

4.5.3 Support sub-package

The support sub-package contains common code and functions used by the GUI components. These support functions can be used to design additional user interfaces using the toolbox. This section briefly describes these functions and how they are used by the existing GUI components.

Warning: Some of these functions should really be part of a custom GUI component layout class. To the best of our knowledge, this is currently not supported for Matlab Apps in R2018a. If this changes in a future Matlab release, much of this code will likely move/change.

Contents

calculateImageSliceFreq

`otslm.ui.support.`**`calculateImageSliceFreq`**(*img*, *theta*, *offset*, *swidth*)

Calculate the frequency spectrum of an image slice

Usage [fvals, freqs] = calculateImageSliceFreq(img, theta, offset, swidth) calculates the frequency spectrum of a slice through an image.

Parameters

- img – Real valued image to calculate spectrum from
- theta – Angle of slice (radians)
- offset – Offset of slice (pixels)
- swidth – width of slice (pixels) to average over

Returns

- fvals – Calculated amplitudes
- freqs – Corresponding frequencies

This function is used for the power spectrum plots in the calibration functions. The function samples a slice of pixels from an image. Arguments control the slice position, width and angle. The function returns the spatial frequencies and complex amplitudes. For example usage, see `ui.utils.CalibrationStepFarfield`.

checkImagesChanged

`otslm.ui.support.`**`checkImagesChanged`**(*oldImages*, *newImages*)

Compare two cell arrays of images for changes

Usage changed = checkImagesChanged(oldImage, newImages) compares each image in the two cell arrays for differences. If the cell arrays are different, returns true.

Parameters

- oldImages – first cell array of images to compare
- newImages – second cell array of images to compare

Returns

- changed (logical) – if the images have changed

This function is used by most methods which have an input image, including `ui.tools.Visualise`, `ui.tools.Finalize` and `ui.tools.Dither`. The two inputs contain cell arrays of matrices to be compared. If either the length of the cell arrays, size or type of the images, or the image data are different, the function returns true. This can be a expensive comparison. We look for changes between the old and new images rather

than watching for a change event on variables, this is to allow the user to enter constants or procedural functions into the GUI inputs.

cleanTimer

otslm.ui.support.**cleanTimer**(*tmr*)

Cleans up the timer when the app is about to finish

Usage cleanTimer(tmr) attempts to delete the specified timer.

Parameters

- tmr – The matlab timer to clean up

This function shouldn't intentionally raise any warnings.

Function attempts to stop and delete the given timer. The function avoids raising errors, making it safe to use in a GUI clean-up method. Timers are mainly used to watch for changes to input variables, such as image inputs to `ui.tools.Visualise`, `ui.tools.Finalize` and `ui.tools.Dither`.

complexPatternValueChanged

otslm.ui.support.**complexPatternValueChanged**(*name*, *phase*, *amplitude*, *ptype*, *device_name*, *enable_update*, *enable_display*, *display_ax*, *display_type*, *display_name*)

Common code for simple update uis with ptype

Usage complexPatternValueChanged(name, phase, amplitude, ptype, device_name, enable_update, enable_display, display_ax, display_type, display_name)

Parameters

- name – Variable name to save pattern in base workspace.
- phase – Phase part of pattern.
- amplitude – Amplitude part of pattern
- ptype – type of pattern. Must be 'phase', 'amplitude' or 'complex'.
- device_name – Name of Showable device to display pattern on
- enable_update – If showable device should be updated
- enable_display – If preview should be displayed
- display_ax – Axis for preview
- display_type – Type argument for display, see *updateComplexDisplay()* for options.
- display_name – Output variable name for preview data

This should realy be part of the base class, but we don't seem to be able to package apps with a custom base class. Maybe in future MATLAB versions this might be possible.

As per *simplePatternValueChanged()* but with complex patterns and an additional `ptype` argument.

See also *iterPatternValueChanged()* and *updateComplexDisplay()*.

findTabUserdata

otslm.ui.support.**findTabUserdata**(*tab*, *tag*)
Find entries with the specific user-data tag and returns a struct

Usage userdata = findTabUserdata(tab, tag)

Parameters

- tab – An object which can be passed to `findall`
- tag – Cell array of values for UserData property to search for

Returns `struct` with fields corresponding to `tag` values

This function uses `findall` to search the given `Tab` for entries whose `UserData` attribute is set to one of the specified strings. `tag` should be a cell array of character vectors for the tags to search for. Example usage (based on `ui.tools.SampleRegion`):

getDeviceFromBase

otslm.ui.support.**getDeviceFromBase**(*sname*)
Get an showable object from the base workspace

Usage slm = getDeviceFromBase(sname)

Parameters

- sname – string for device variable name in base workspace

Returns Returns the Showable device or an empty list.

This function attempts to get the variable specified by `sname` from the base workspace. If `sname` is empty, the function returns an empty matrix. If `sname` is not a variable name, the function raises a warning. Otherwise, the function gets the variable and checks to see if it is valid using `isvalid`. For example usage see *simplePatternValueChanged()*.

getImageOrNone

otslm.ui.support.**getImageOrNone**(*name*, *silent*)
Get the image from the base workspace or an empty array

Usage im = getImageOrNone(name) gets the variable name from base or None if any error occurs.

im = getImageOrNone(name, silent) as above but if `silent=true` does not retrow the error to the console, just silently ignores it.

Parameters

- name – variable name for image in base workspace
- silent (logical) – True if the method should not print warnings

Attempts to evaluate the given string in the base workspace with `evalin`. The string can either be a variable name or valid matlab code which can be evaluated in the users base workspace.

If an error occurs, the function prints the error to the console and returns a empty matrix. If the silent argument is set to true, the function does not print to the console (useful for methods which frequently check for the existence of a variable, such as *checkImagesChanged()*. For example usage, see `ui.tools.Visualise`, `ui.tools.finalize` and `ui.tools.dither`.

iterPatternValueChanged

otslm.ui.support.**iterPatternValueChanged**(*name*, *pattern*, *device_name*, *enable_update*, *enable_display*, *display_ax*, *display_type*, *display_name*, *fitness_method*)

Common code for iter update uis

Usage iterPatternValueChanged(name, pattern, ... device_name, enable_update, enable_display, ... display_ax, display_type, display_name, fitness_method)

Parameters

- name – Variable name to save pattern in base workspace.
- pattern – Pattern to display/preview
- device_name – Name of Showable device to display pattern on
- enable_update – If showable device should be updated
- enable_display – If preview should be displayed
- display_ax – Axis for preview
- display_type – Type argument for display, see *updateIterDisplay()* for options.
- display_name – Output variable name for preview data
- fitness_method – Function handle for fitness graph

This should realy be part of the base class, but we don't seem to be able to package apps with a custom base class. Maybe in future MATLAB versions this might be possible.

Function is similar to *simplePatternValueChanged()* but with a function handle for plotting fitness scores.

See also See also *complexPatternValueChanged()* and *updateIterDisplay()*.

populateDeviceList

otslm.ui.support.**populateDeviceList**(*list*, *type_name*)

Populates the device list with Showable devices

Usage populateDeviceList(list) populates the device drop down list with the `otslm.utils.Showable` devices in the base workspace.

populateDeviceList(list, type_name) specify types of devices to populate list with.

Parameters

- list (uidropdown) – List handle to add items to
- type_name – Name of type to filter variables by (optional, default: `otslm.utils.Showable`)

This function is used to populate the contents of a `uidropdown` widget. The function takes a handle to the `uidropdown` widget, an optional Matlab class name and searches the base workspace for variables with the specified type. If no class name is specified, the method populates the list with `Showable` object names. For example usage, see `ui.simple.linear`.

saveVariableToBase

otslm.ui.support.**saveVariableToBase**(*name*, *pattern*, *warn_prefix*)

Saves the variables to the base workspace

Usage saveVariableToBase(name, pattern, warn_prefix) Saves the variable pattern to the base workspace with variable name *name*. If name is invalid, prefixes warning with *warn_prefix*.

Parameters

- name – variable name to save pattern to
- pattern – pattern to be saved
- warn_prefix – prefix to add to warnings

If the name is empty, aborts the operation. If the names is an invalid variable name, raises a warning.

This function is used by most GUIs for saving computed patterns into the base workspace, for example usage see *simplePatternValueChanged()*.

simplePatternValueChanged

otslm.ui.support.**simplePatternValueChanged**(*name*, *pattern*, *device_name*, *enable_update*, *enable_display*, *display_ax*, *display_type*, *display_name*)

Common code for simple update uis

Usage simplePatternValueChanged(name, pattern, ... device_name, enable_update, enable_display, ... display_ax, display_type, display_name)

Parameters

- name – Variable name to save pattern in base workspace.
- pattern – Pattern to save/preview/display
- device_name – Name of Showable device to display pattern on
- enable_update – If showable device should be updated
- enable_display – If preview should be displayed
- display_ax – Axis for preview
- display_type – Type argument for display, see *updateSimpleDisplay()* for options.
- display_name – Output variable name for preview data

This should realy be part of the base class, but we don't seem to be able to package apps with a custom base class. Maybe in future MATLAB versions this might be possible.

This function is used by most of the simple GUIs including `ui.simple.linear`, `ui.simple.random`, and `ui.tools.combine`. The function takes as input values from the various GUI components as well as the generated pattern. The function saves the pattern to the workspace, displays the pattern on the device, and updates the pattern preview (if the appropriate values are set).

See also *iterPatternValueChanged()* and *complexPatternValueChanged()*.

updateComplexDisplay

otslm.ui.support.**updateComplexDisplay**(*pattern*, *slm*, *ptype*, *display_type*, *ax*, *output_name*)
Helper for the display on simple uis with ptype

Usage updateComplexDisplay(pattern, slm, ptype, display_type, ax, output_name) Generates the display pattern, updates the axis and outputs to base.

Parameters

- pattern – pattern to be displayed
- slm – showable device displaying pattern (or `[]`)
- ptype – type of pattern. Must be 'phase', 'amplitude' or 'complex'.
- display_type – mode for the preview window. can be 'phase', 'raw', 'device', or 'farfield'.
- ax – axis to place the preview in
- output_name – output variable name in base workspace (or `[]`)

As per *updateSimpleDisplay()* but with complex patterns and an additional `ptype` argument.

See also *updateIterDisplay()* and *complexPatternValueChanged()*

updateIterDisplay

otslm.ui.support.**updateIterDisplay**(*pattern*, *slm*, *display_type*, *ax*, *output_name*, *fitness_method*)
Helper for updating the display on iterative uis.

Usage updateIterDisplay(pattern, slm, display_type, ax, output_name, fitness_method) generates the display pattern, updates the axis and outputs to base.

Parameters

- pattern – pattern to be displayed
- slm – showable device displaying pattern (or `[]`)
- display_type – mode for the preview window. can be 'phase', 'error', 'device', or 'farfield'
- ax – axis to place the preview in
- output_name – output variable name in base workspace (or `[]`)
- fitness_method – function to plot fitness

Similar to *updateSimpleDisplay()* but displays either the phase pattern, error function, simulated farfield or device pattern in the preview window.

This function generates the pattern to display in the preview axis. If `output_name` is not empty, the function also writes the pattern to the specified variable name.

See also *updateComplexDisplay()* and *iterPatternValueChanged()*.

updateSimpleDisplay

otslm.ui.support.**updateSimpleDisplay**(*pattern*, *slm*, *display_type*, *ax*, *output_name*)
Helper for updating the display on simple uis.

Usage updateSimpleDisplay(pattern, slm, display_type, ax, output_name) Generates the display pattern, updates the axis and outputs to the base workspace.

Parameters

- pattern – pattern to be displayed
- slm – showable device displaying pattern (or `[]`)
- display_type – mode for the preview window. can be 'Phase mask', 'Raw phase mask', 'Device image', or 'Simulated farfield'
- ax – axis to place the preview in
- output_name – output variable name in base workspace (or `[]`)

This function generates the pattern to display in the preview axis. If `output_name` is not empty, the function also writes the pattern to the specified variable name.

This function is used by most of the simple GUIs including `ui.simple.linear`, `ui.simple.random`, and `ui.tools.combine`. For example usage, see *simplePatternValueChanged()*.

See also *updateComplexDisplay()* and *updateIterDisplay()*.

APPENDIX A

Documentation terms of use