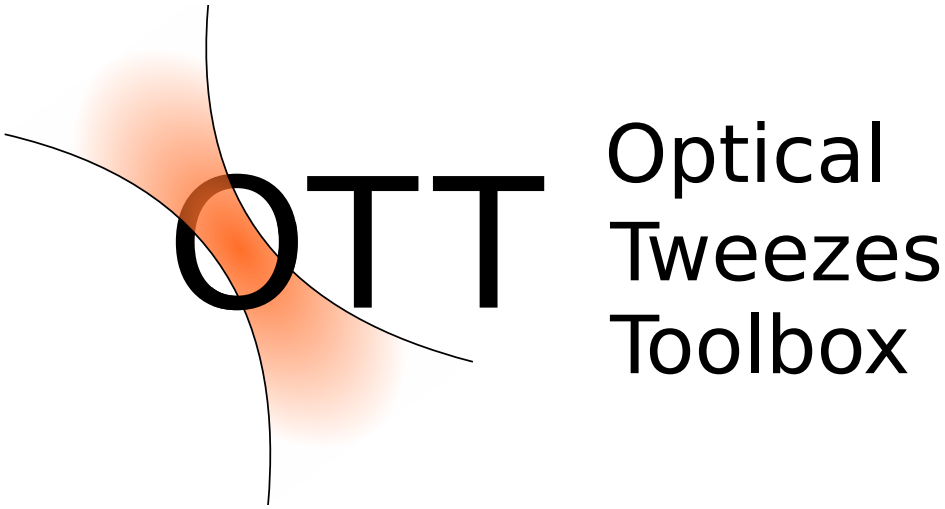


OTT Documentation

Isaac Lenton

Jun 28, 2022

TABLE OF CONTENTS

CHAPTER

ONE

INTRODUCTION

The optical tweezers toolbox can be used to calculate optical forces and torques of particles using the T-matrix formalism in a vector spherical wave basis. The toolbox includes codes for calculating T-matrices, beams described by vector spherical wave functions, functions for calculating forces and torques, simple codes for simulating dynamics and examples.

To get started using the toolbox, checkout the Getting Started page.

1.1 License

This version of the toolbox can be referenced by citing the following paper

> T. A. Nieminen, V. L. Y. Loke, A. B. Stilgoe, G. Knöner, A. M. Branczyk, N. R. Heckenberg, and H. Rubinsztein-Dunlop, "Optical tweezers computational toolbox", Journal of Optics A 9, S196-S203 (2007)

or by directly citing the toolbox

> I. C. D. Lenton, T. A. Nieminen, V. L. Y. Loke, A. B. Stilgoe, Y. Hu, G. Knöner, A. M. Branczyk, N. R. Heckenberg, and H. Rubinsztein-Dunlop, "Optical tweezers toolbox", https://github.com/ilent2/ott

and the respective Bibtex entry

```
@misc{Lenton2020,
  author = {Lenton, Isaac C. D. and Nieminen, Timo A. and Loke, Vincent L. Y. and␣
↪Stilgoe, Alexander B. and Y. Hu and Kn{\ifmmode\ddot{o}\else\"{o}\fi}ner, Gregor and␣
↪Bra{\ifmmode\acute{n}\else\'{n}\fi}czyk, Agata M. and Heckenberg, Norman R. and␣
↪Rubinsztein-Dunlop, Halina},
  title = {Optical Tweezers Toolbox},
  year = {2020},
  publisher = {GitHub},
  howpublished = {\url{https://github.com/ilent2/ott}},
  commit = {A specific commit or version (optional)}
}
```

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 OTT 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

TWO

GETTING STARTED

This section has information about getting started with the toolbox including information on *installation*, *using the GUIs* and *running the example files*. This section also contains a brief *overview of the toolbox*, further information can be found in the papers listed in *Further Reading*.

2.1 Installation

To use the toolbox, you will need a recent version of Matlab (at least R2016b; R2018a is needed for some features; Octave should also work) and the latest version of OTT. There are a couple of ways to get OTT. You can download one of the Matlab toolbox files (with the `.mltbx` file extension), or download one of the `.zip` archives containing the source code, or download the latest source directly from GitHub. If you are using Matlab, the easiest method is to install OTT via the Addons explorer. The following sub-sections detail each of these methods.

2.1.1 Installing via Matlab Addons Explorer

If using Matlab, the easiest method to install the toolbox is using the Matlab Addons explorer. Simply launch Matlab and navigate to **Home > Addons > Get-Addons** and search for "Optical Tweezers Toolbox". Then, simply click the *Add from GitHub* button to automatically download the package and add it to the path. You may need to logging to a Mathworks account to complete this step.

2.1.2 Using a `.mltbx` file

You can download the latest stable release of OTT from either the GitHub release page. 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.

2.1.3 Using a `.zip` or cloning the repository

The latest version of OTT can be downloaded from the OTT GitHub page. Simply click the *Clone or Download* button and select your preferred method of download. 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.

If you are cloning the repository you can checkout different tags to select the desired release. Alternatively, for a specific release, navigate to the release page and select the `.zip` file for the desired release.

To install OTT, 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`. To use the graphical user interface or add the toolbox to your own code, you will need to make Matlab aware of the toolbox path. To do this, simply run

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

Replace the path with the path you placed the downloaded toolbox in. The folder must contain the `+ott` 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 `ott-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.

2.1.4 Post installation

To check that `ott` installed correctly and can be found by Matlab, run the following command and verify it displays the contents of the `+ott/Contents.m` file

```
help ott
```

If you have multiple versions of `ott` installed, you may want to check which version is currently being used. The following command can be used to check the path of the toolbox currently being found

```
what ott
```

Further information about using the toolbox functions and graphical user interface can be found in subsequent sections.

The toolbox runs on recent versions of Matlab, most functionality should work on at least R2016b but the graphical user interface might need R2018a or newer (we have tested the toolbox on R2018a). Most functionality should work with [GNU Octave](https://www.gnu.org/software/octave/), however this has not been tested recently and performance is optimised for Matlab.

Some functionality may require additional dependences including additional Matlab products. We are currently working on a full list; feel free to get in contact if you encounter problems with missing dependencies. 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 Toolbox Overview

The toolbox includes a collection of functions and classes for calculating optical forces and torques for particles in various light fields. The core toolbox files are grouped into a Matlab package (a folder with a + prefix). Other components, including examples and documentation, are provided in separate folders in the directory where OTT was downloaded/installed. If you installed OTT by downloading a `.zip` or cloning the repository, the OTT path is the directory containing the `+ott` and `docs` directories. If you installed OTT with a Matlab package or via the Addons explorer, you can view the OTT directory containing the docs and examples by navigating to **Home > Addons > Manage addons**, find the toolbox and select **Options > Open folder**.

The following list provides a brief overview of the toolbox parts and the corresponding folders/file paths:

Examples (*<ott-path>/examples*)
: This directory contains examples of various features included in the toolbox. Most of these examples are described in the *Examples* part of the documentation and information on *running the example files* can be found bellow.

Graphical user interface (*+ott/+ui*)
: This sub-package contains the graphical user interface components. See bellow for information on *using the GUIs*.

BSC and T-matrix classes (*+ott/Bsc and *+ott/Tmatrix**)**
: The BSC and T-matrix classes represent beams and particles in the toolbox. In the vector spherical wave function (VSWF) basis, beams are represented by vectors describing a superposition of VSWF components and particles are represented by matrices which operate on beam-vectors to produce scattered beam-vectors. The BSC and T-matrix classes behave like regular Matlab vector and matrix classes but also provide additional functionality such as functions for visualising fields and beam related properties (wavelength, numerical aperture, etc.). Further details can be found in the *Bsc classes* and *Tmatrix classes* reference pages.

Functions operating on beams and particles (*+ott/ functions)**
: In addition to the BSC and T-matrix classes, the *+ott* package contains a range of other functions for calculating forces and locating traps. Further information can be found in *Other functions*.

Geometric Shapes (*+ott/+shapes*)
: This sub-package provides descriptions of Geometric shapes which are used mostly by the point-matching and DDA routines for generating T-matrices for particles. See *shapes Package* reference pages for more information.

Utility functions (*+ott/+utils*)
: This directory contains functions commonly used by other parts of the toolbox. Most users will probably not need to access these directly. See *utils Package* reference pages for more information.

Documentation (*<ott-path>/docs*)
: This directory contains the restructured text (ReST) used to generate this documentation. If you don't have or prefer not to use a web browser to view the documentation, you can open these files in most regular text editors.

Unit tests (*<ott-path>/tests*)
: This directory contains functions and scripts for testing the toolbox functionality. This is only included in the GitHub version and you should not need to interact with this directory unless you are contributing to OTT.

The toolbox doesn't use any particular units, although most examples will assume units of dimensionless force F_Q and torque τ_Q efficiencies. To convert to SI units:

$$F_{SI} = F_Q \frac{nP}{c}$$
$$\tau_{SI} = \tau_Q \frac{P}{\omega}$$

where n is the refractive index of the medium, P is the beam power, c is the speed of light in vacuum, and ω is the optical frequency. You should be able to use any units as long as you are consistent with defining parameters. However, this hasn't been thoroughly tested, if you encounter any inconsistencies, please let us know.

To learn more about how the toolbox calculates forces and torques, take a look at the original paper describing OTT

> T. A. Nieminen, V. L. Y. Loke, A. B. Stilgoe, G. Knöner, A. M. Branczyk, N. R. Heckenberg, and H. Rubinsztein-Dunlop, "Optical tweezers computational toolbox", Journal of Optics A 9, S196-S203 (2007)

2.3 Exploring the toolbox with the GUI

The toolbox includes a graphical user interface (GUI) for performing many of the basic tasks including generating beams, T-matrices and calculating force profiles. The user interface can be used to explore the basic functionalities 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 `+ott/+ui` directory and running `Launcher.mlapp`. Alternatively, you can launch the GUI from the command window with

```
ott.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.

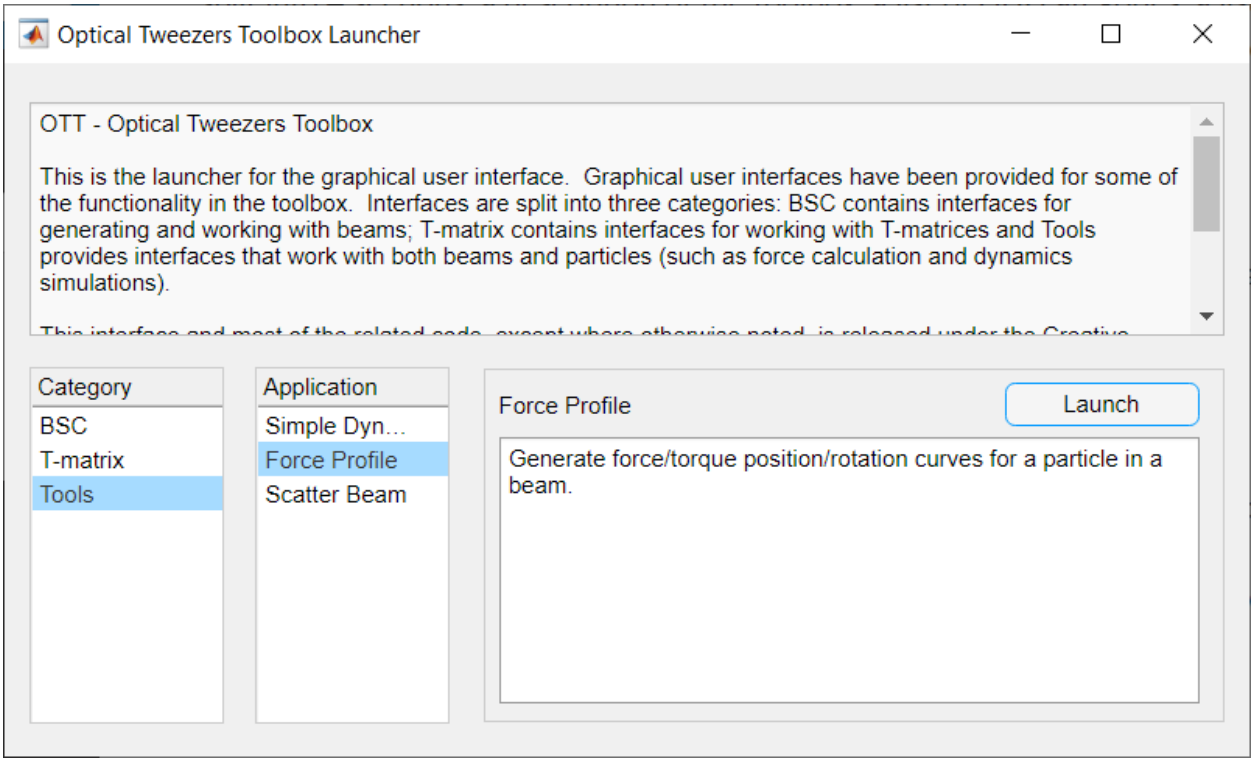


Fig. 2.1: Overview of the Launcher application.

BSC and T-matrix generation function need to specify a variable name. This variable name is used when assigning the object data to the Matlab workspace. Other GUIs which support output can also specify a variable name. Output from one GUI can be used as input to another GUI by specifying the corresponding variable name as the input.

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

For a complete example showing how to use the GUI, see *Calculating forces with the GUI*

2.4 Running the examples

To run the examples, navigate to the examples directory, either following the instructions above or using the `what` command:

```
what_result = what('ott');
ott_path = fileparts(what_result(end).path);
cd([a, '/examples']);
```

To run an example, open the script and run it (either the full file or section-by-section). The first line in most script files is `addpath('../')`, this line ensure OTT is added to the path. If you have already added OTT to the path or installed OTT as an Add-on, this line is unnecessary. If you copy the example to another directory, you will need to adjust the `addpath` command accordingly.

Further documentation and example output for specific examples can be found in *Examples*.

CHAPTER

THREE

EXAMPLES

This section provides detailed examples to help you get started using the toolbox. Further examples scripts and LiveScripts can be found in the `examples` directory.

3.1 Calculating forces on a spherical particle

This section is a companion for the `examples/example_sphere.m` script and will guide you through the core functionality of the toolbox. The script sets up the variables for calculating forces on a spherical particle in a Gaussian beam. The particle is created using the *`ott.Tmatrix.simple()`* method, which for a spherical particle calls the *`ott.TmatrixMie`* class constructor. The beam is constructed using the *`ott.BscPmGauss`* class which can also be used for Laguerre-Gaussian and Hermite-Gaussian beam modes. And, forces are calculated using *`ott.forcetorque()`*. A similar example is described in *Calculating forces with the GUI*.

Contents

3.1.1 Setting up the Matlab workspace

The `example_sphere.m` script starts by setting up the Matlab workspace to work with OTT. The first step is to add OTT to the Matlab path, this step is not required if OTT is already on the path (for instance, if you installed OTT via the add-ons menu, OTT should already be on the path).

```
addpath('../');   % Change this to your OTT path if required
```

The next step is clearing all existing variables and configuring OTT specific warnings. Some OTT functions can trigger many warnings, to reduce the verbosity of the output, OTT can be asked to only warn once about issues during this Matlab session. Several OTT functions are also likely to move in a future release, we can turn off warnings related to these changes here too.

```
close all;
ott.warning('once');
ott.change_warnings('off');
```

Next, we declare our material properties, wavelength, sphere radius and numerical aperture for the objective.

```
n_medium = 1.33;         % Water
n_particle = 1.59;       % Polystyrene
wavelength0 = 1064e-9;   % Vacuum wavelength
wavelength_medium = wavelength0 / n_medium;
radius = 1.0*wavelength_medium;
NA = 1.02;               % Numerical aperture of beam
```

3.1.2 Generating the beam shape coefficients

To create a beam, we use *ott.BscPmGauss*. The constructor for this class accepts several positional and named arguments, in this example we set the numerical aperture, polarisation, refractive index and vacuum wavelength.

```
beam = ott.BscPmGauss('NA', NA, 'polarisation', [ 1 1i ], ...
     'index_medium', n_medium, 'wavelength0', wavelength0);
```

The class can also be used for generating Laguerre-Gaussian beams and other type of beams by adding additional parameters. For example, the following would generate an LG(0, 3) beam

```
beam = ott.BscPmGauss('lg', [ 0 3 ], ...
     'polarisation', [ 1 1i ], 'NA', NA, ...
     'index_medium', n_medium, 'wavelength0', wavelength0);
```

We may also want to set or normalise the beam power, this can be done at any time by setting the `power` property, for example

```
beam.power = 1.0;
```

Regardless of the type of beam we are using, we are now able to visualise the beam. The *ott.Bsc* base class (which *ott.BscPmGauss* inherits from) defines several visualisation function. To visualise the field around the focus, we can use *ott.Bsc.visualise()*. Before calling the function we need to specify the vector spherical wave function basis to use, for near-field visualisation this should be set to *regular*.

```
beam.basis = 'regular';

figure();
subplot(1, 2, 1);
beam.visualise('axis', 'y');
subplot(1, 2, 2);
beam.visualise('axis', 'z');
```

The above code should produce something similar to figure Fig. 3.1. The *axis* parameter specifies which axis should be normal to our visualisation slice.

We can also visualise the far-field of the beam. For this we set the basis to *incoming* and use the *ott.Bsc.visualiseFarfield()* function.

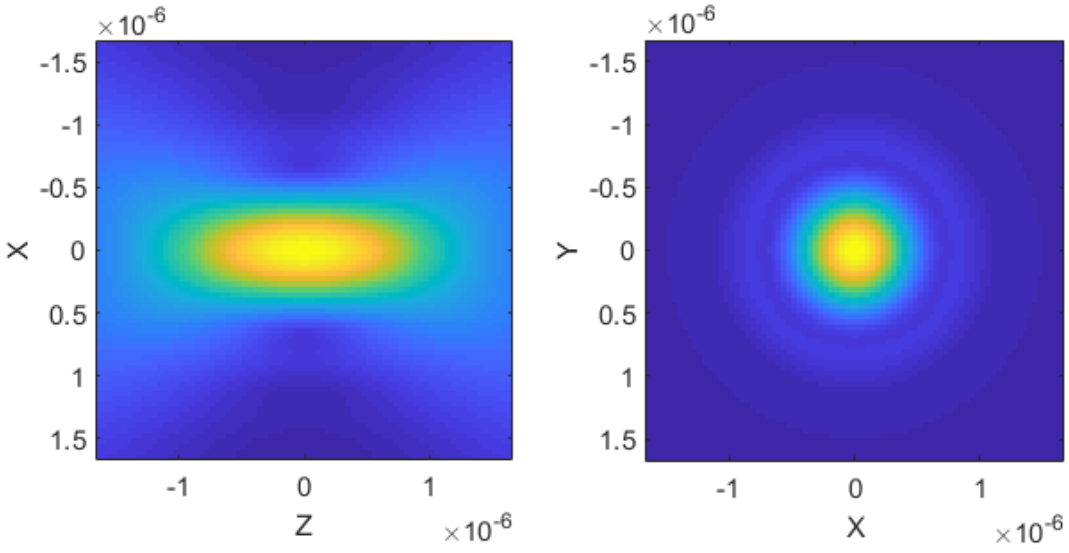


Fig. 3.1: Visualisation of the incident beam near-field.

```
beam.basis = 'incoming';

figure();
beam.visualiseFarfield('dir', 'neg');
```

The above should produce something similar to figure Fig. 3.2. The *dir* parameter specifies which hemisphere we want to look at, in this case we look at the negative (backward) hemisphere. Depending on the beam and the chosen basis, either the forward or backward hemisphere may have very little power, if you are unsure about the direction of your beam it is a good idea to look in both directions.

3.1.3 Generating the T-matrix

In this simulation we use a T-matrix for a spherical particle. The T-matrix is diagonal and the elements along the diagonal are the Mie coefficients for a sphere. To calculate the T-matrix we use the *ott.Tmatrix.simple()* method, we specify the shape as a sphere and the method automatically selects the best method for this shape, in this case *ott.TmatrixMie*. The *ott.Tmatrix.simple()* method takes various named parameters for the particle size, shape and refractive index.

```
T = ott.Tmatrix.simple('sphere', radius, 'wavelength0', wavelength0, ...
   'index_medium', n_medium, 'index_particle', n_particle);
```

For a sphere, this should only take a couple of seconds to evaluate.

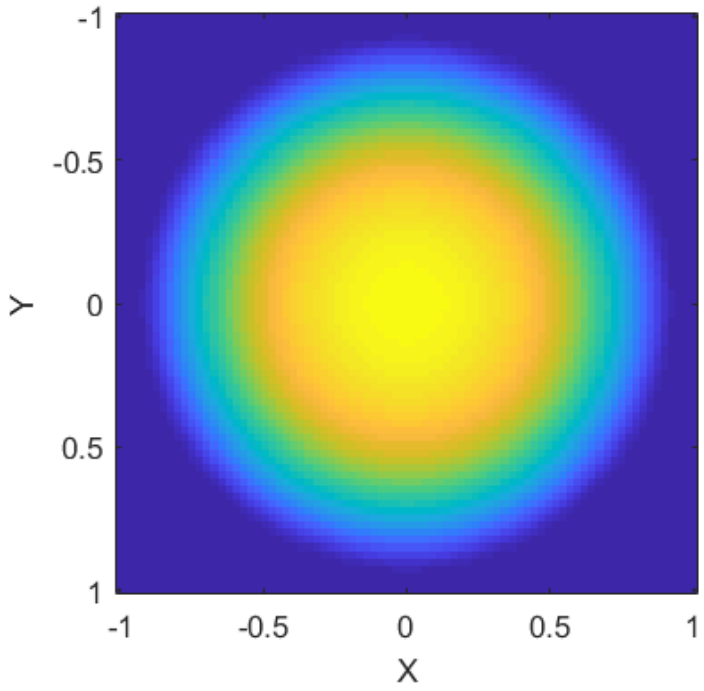


Fig. 3.2: Visualisation of the incident beam far-field.

3.1.4 Calculate the scattered field

The T-matrix and beam objects encapsulate the data for the T-matrix and beam shape coefficients (a matrix and vector respectively). We can view this data by accessing the `data` attribute of these objects, for example

```
disp(T.data);
```

In the T-matrix method, a T-matrix describes how a particle scatters light. It is a linear matrix which relates each incident mode to each scattered mode, mathematically this is

$$\begin{pmatrix} p \\ q \end{pmatrix} = T \begin{pmatrix} a \\ b \end{pmatrix}$$

where T is the T-matrix, and (a, b), (p, q) are the incident and scattered beam shape coefficients. To implement this in OTT, we can simply write

```
sbeam = T * beam;
```

This is equivalent to directly multiplying the `T.data` and `beam.data` matrix and vector objects to calculate the resulting scattered beam shape coefficients, and encapsulating the result in a *`ott.Bsc`* object.

As with the incident beam, we are able to generate various visualisations of the fields. The following example shows a visualisation of the scattered field and the total field (incident + scattered) around the beam focus, for the sphere and Gaussian beam described above, the results are shown in Fig. 3.3.

```
figure();
subplot(1, 2, 1);
sbeam.basis = 'outgoing';
```

(continues on next page)

(continued from previous page)

```
sbeam.visualise('axis', 'y', ...
   'mask', @(xyz) vecnorm(xyz) < radius, 'range', [1,1]*2e-6)
title('Scattered field');

subplot(1, 2, 2);
tbeam = sbeam.totalField(beam);
tbeam.basis = 'regular';
tbeam.visualise('axis', 'y', ...
   'mask', @(xyz) vecnorm(xyz) < radius, 'range', [1,1]*2e-6)
title('Total field');
```

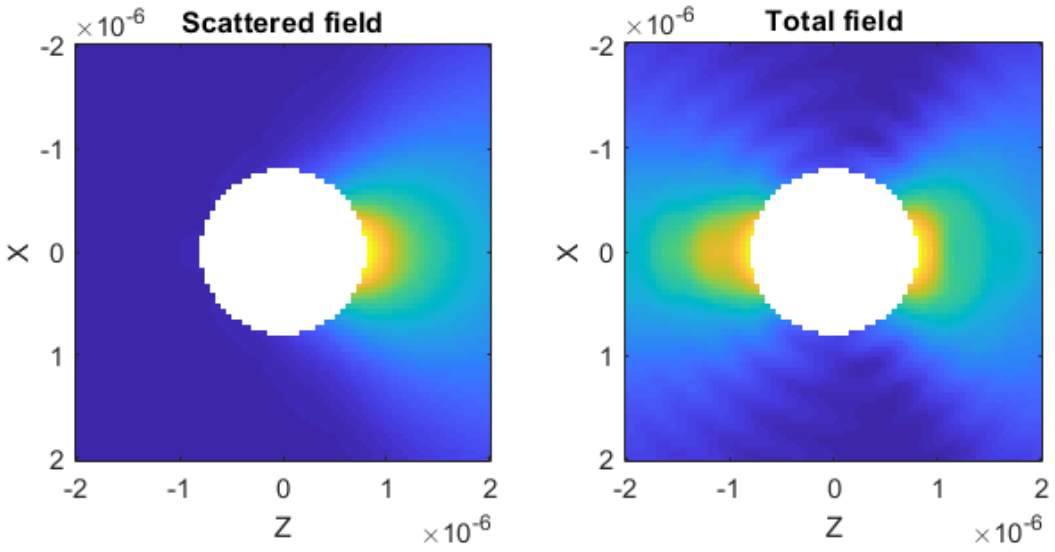


Fig. 3.3: The total and scattered field visualised for a spherical particle at the focus of a Gaussian beam. This slice is along the beam axis, the region corresponding to the particle has been removed.

The T-matrix in this example only gives the fields outside the particle, we use the `mask` parameter to remove the region inside the particle. To visualise the fields inside the particle we would need to calculate a internal T-matrix instead.

3.1.5 Calculating optical forces

Now that we have a scattered beam, we are able to calculate the change in momentum between the incident beam and the particle; and, therefore, infer the force acting on the particle. The main function for calculating forces is *ott.forcetorque()*, this function can operate on beams or beams and T-matrix. When both the inputs are beams, the function calculates forces and torques using various summations over the beam shape coefficients.

```
[force, torque] = ott.forcetorque(beam, sbeam);
```

In this example, the force would be `0.0135` and the torque would be `1e-16` which is on the order of round-off error (i.e. numerically equivalent to zero). The units depend on the units used for the beam, in this example we can convert to SI units (Newtons) using

```
nPc = 0.001 .* index_medium / 3e8;  % 0.001 W * n / vacuum_speed
force_SI = force .* nPc
```

3.1.6 Beam translations

Being able to calculate optical forces is only useful if we can translate either the beam or the particle to different locations. For this, we can use the beam *ott.Bsc.translateXyz()* function. The behaviour of this function depends on the current beam basis: if the beam was generated using one of the `ott.Bsc*` functions, the basis should typically be set to *regular*; if the beam was generated from scattering by another particle, the basis should be *outgoing*. For example, in this example we could translate the beam along the x-axis with

```
beam.basis = 'regular';
x = 1.0e-6; y = 0.0e-6; z = 0.0e-6;
offset_beam = beam.translateXyz([x; y; z]);
```

We could translate the beam and calculate the forces multiple times with the above method; however, OTT provides a more convinient method using *ott.forcetorque()*.

3.1.7 Calculate multiple forces with `ott.forcetorque`

Instead of passing two beam objects to *ott.forcetorque()*, we could instead pass a beam and T-matrix object and the various positions we want to translate the beam to. As with other translations, it is important to set the basis of the incident beam before calling the method. The following code calculates the force along the beam axis (the z-axis), output is shown in Fig. 3.4.

```
xyz = [0;0;1] .* linspace(-4, 4, 100).*1e-6;
fxyz = ott.forcetorque(beam, T, 'position', xyz);

figure();
nPc = n_medium / 3e8;  % n / vacuum_speed
plot(xyz(3, :), fxyz .* nPc);
xlabel('Z position [m]');
ylabel('Force [N/W]');
legend({'Fx', 'Fy', 'Fz'});
```

3.2 Simulating vaterite with DDA

This section is a companion for the `examples/dda_vaterite.m` script and will guide you through simulating an inhomogeneous particle using the discrete dipole approximation. The script simulates the vaterite particles described in

> Highly birefringent vaterite microspheres: production, characterization and applications for optical micromanipulation. S. Parkin, et al., Optics Express Vol. 17, Issue 24, pp. 21944-21955 (2009) https://doi.org/10.1364/OE.17.021944

These particles are made of a uniaxial birefringent material. The crystal axis is aligned according to a sheaf-of-wheat structure, shown in figure Fig. 3.5. In order to simulate this particle, we need to specify the polarizability of each dipole (computational unit cell).

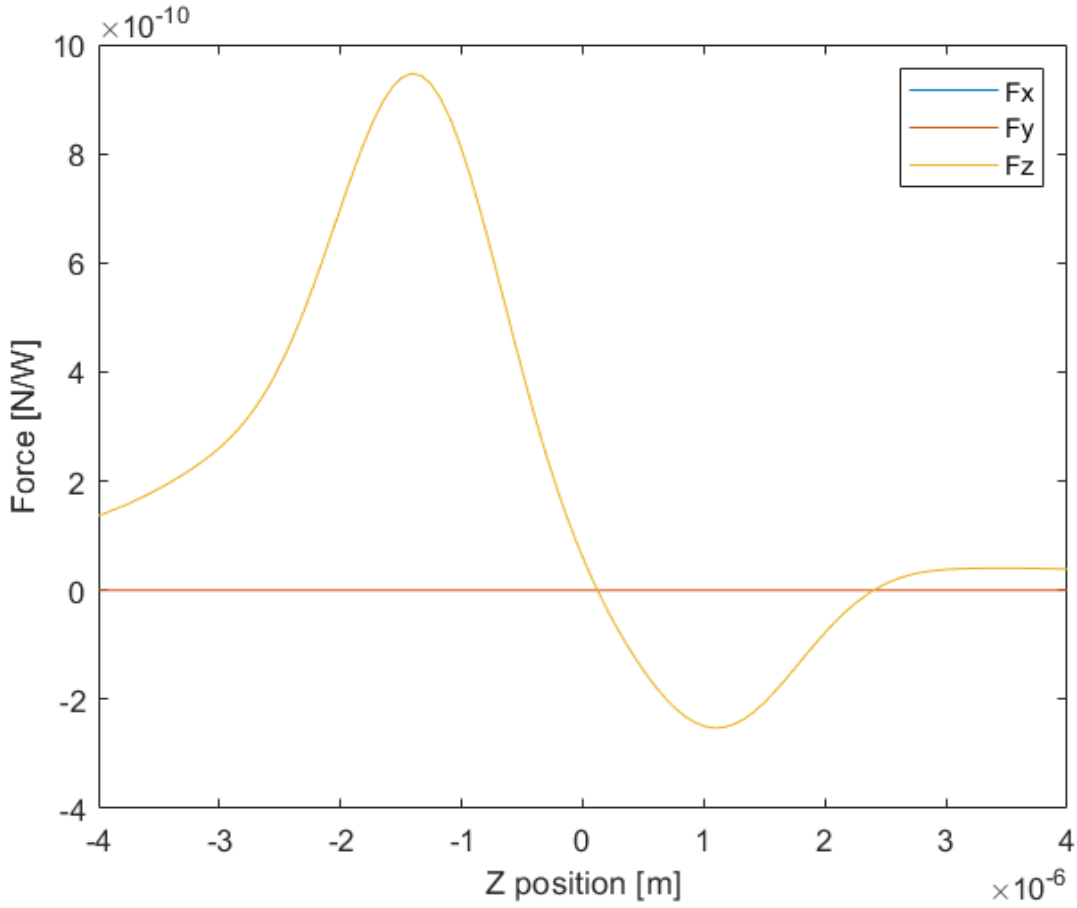


Fig. 3.4: The force on a spherical particle positioned at different locations along the beam axis. The transverse components of the force are approximately zero. The axial force displays the well known profile of an optically traped particle.

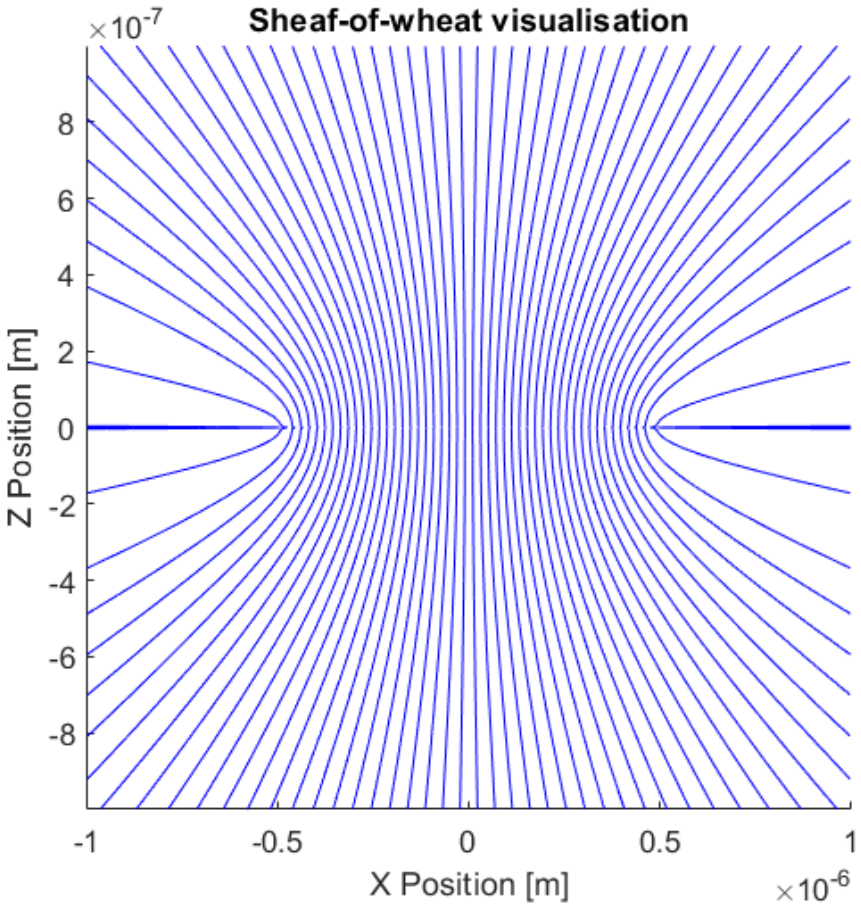


Fig. 3.5: Illustration of the vaterite sheaf of wheat structure. The image shows a slice through the XZ plane. The particle is rotaionly symmetry about the Z axis and mirror symmetric about the XY plane.

3.2.1 Describing the vaterite properties

To describe the vaterite shape, we use *ott.shapes.Sphere*. DDA requires the shape to be specified using voxels, for this we used `ott.shapes.Sphere.voxels()` to calculate voxels on a grid with a regular spacing.

```
spacing = wavelength0 / 10;
radius = wavelength0;
shape = ott.shapes.Sphere(radius);
voxels = shape.voxels(spacing, 'even_range', true);
```

The `even_range` parameter tells the voxels function to use an even number of points for the size of each voxel grid dimension. This means that the voxel grid will not place voxels at the origin making it easier to use mirror and rotational symmetry options for DDA.

To calculate the polarizability for a unit cell we use the method based on the lattice dispersion relation from *ott.utils.polarizability*. Vaterite is a uniaxial crystal with an ordinary and extraordinary refractive index, `index_o` and `index_e` respectively.

```
upol = ott.utils.polarizability.LDR(spacing ./ wavelength0, ...
   [index_o; index_o; index_e] ./ index_medium);
```

To generate the sheaf of wheat structure we used the `sheafOfWheat` function defined in the script to calculate the orientation direction of each unit cell.

```
dirs = sheafOfWheat(voxels, 0.5*radius);
```

Finally, we rotate the polarisability and refractive index so the `z` direction aligns with the sheaf-of-wheat direction

```
index_relative = ott.utils.rotate_3x3tensor(...
    diag([index_o, index_o, index_e] ./ index_medium), 'dir', dirs);
polarizabilities = ott.utils.rotate_3x3tensor(...
    diag(upol), 'dir', dirs);
```

3.2.2 Calculating the T-matrix with DDA

Once all the properties have been described, we construct the T-matrix using *ott.TmatrixDda*. We use the class constructor rather than *ott.TmatrixDda.simple()* in order to specify the polarizability of each voxel. If we wanted to simulate a homogeneous sphere we could use the simple method.

```
Tmatrix = ott.TmatrixDda(voxels, ...
     'polarizability', polarizabilities, ...
     'index_relative', index_relative, ...
     'index_medium', index_medium, ...
     'spacing', spacing, ...
     'z_rotational_symmetry', 4, ...
     'z_mirror_symmetry', true, ...
     'wavelength0', wavelength0, ...
     'low_memory', low_memory);
```

Most of the arguments are fairly intuitive: we specify the material properties and voxel locations with `voxels` and `polarizabilities`. When `polarizabilities` are specified explicitly, we still need to specify `index_relative` in order to ensure correct scaling of the resulting T-matrix. The `z_rotational_symmetry` argument tells the DDA implmenetation to use fourth order rotational symmetry about the z-axis and the `z_mirror_symmetry` says to use mirror symmetry about the XY plane. The low memory option can be used with rotational symmetry or mirror symmetry

to reduce the amount of memory required at a slight reduction to computational efficiency. `spacing` is currently only used for the `Nmax` estimation. If the polarizabilities were not specified, `spacing` would also be used for calculating the polarizabilities from `index_relative`. `wavelength0` and `index_medium` specify the units for distance, i.e. the scaling that should be applied to `voxels`.

Depending on the size of the particle and the number of dipoles, this could take anywhere from a couple of minutes to several hours to run. An alternative is to only calculate modes which are present in the illuminating beam, this can be achieved using the `modes` option, for examples, to a beam with only `m = 1` modes you could do

```
Nmax = 5;
m = 1;
n = max(1, abs(m)):Nmax;
modes = [n(:), repmat(m, size(n(:)))];

Tmatrix = ott.TmatrixDda(voxels, ...
   'modes', modes);
```

The above could also be used to calculate the T-matrix in parallel by specifying a different mode for each worker to calculate and then combining the T-matrix columns after calculation.

If you are calculating large T-matrices, you will probably want to save them to a file so you can use them again later without needing to rerun DDA. For this, simply use the `save` command

```
save('output.mat', 'Tmatrix')
```

To load the T-matrix back into Matlab, first make sure OTT is on the matlab path and then either double click on the `.mat` file or run

```
load('output.mat')
```

3.2.3 Calculating torque on the particle

The calculated T-matrix can be used like any other T-matrix object. For example, we can create a beam and calculate the torque on the particle at different locations in the beam:

```
beam = ott.BscPmGauss('NA', 1.1, 'index_medium', index_medium, ...
   'power', 1.0, 'wavelength0', wavelength0, 'polarisation', [1, -1i]);

[~, tz] = ott.forcetorque(beam, Tmatrix, ...
    'position', [0;0;0], 'rotation', eye(3));
```

3.3 Combining Beams

The toolbox can calculate the force on a particle from multiple beams either coherently or incoherently. This page describes the different beam combination methods used in the `multiple_beams.m` example. For both coherent and incoherent beams, the order which beams are combined and translated can affect accuracy and computation time. This page assumes the beams can be exactly represented, as is the case for focused beams which pass through a finite aperture. For plane waves and other beams requiring an infinite VSWF expansion, it is better to calculate the beams at the new location.

For this example, we calculate the forces on a spherical partice, the T-matrix for this particle is given by

```
% Wavelength in medium/vacuum [m]
wavelength = 1064.0e-9;

T = ott.Tmatrix.simple('sphere', wavelength, 'index_medium', 1.0, ...
    'index_particle', 1.2, 'wavelength0', wavelength);
```

We combine two copies of the same beam displaced along the x axis:

```
beam = ott.BscPmGauss('polarisation', [1 1i], 'angle_deg', 50, ...
    'index_medium', 1.0, 'wavelength0', wavelength, 'power', 0.5);

% Displacement of beams [wavelength_medium]
displacement = 0.2*wavelength;
```

The original beam is centered around the coordinate origin. When we use these beams we need to translate them to the particle origin. For example, if the particle is displaced a distance `x` from the centre of the two beams, we can translate the beams to this location using by translating each beam separately:

```
beam1 = beam.translateXyz([x+displacement; 0; 0]);
beam2 = beam.translateXyz([x-displacement; 0; 0]);
```

3.3.1 Combining coherent beams

For coherent beams, we need to combine the `a` and `b` coefficients before doing the final force calculation. We can either: * translate both beams from the coordinate origin to the displaced locations, combine the beams and then translate the combined beam to the particle location * translate each beam to the particle location and combine the beams before calculating the force

Depending on the size of the beams, the separation and the size of the particle, these methods will take different amounts of time. If the particle is significantly smaller than the beam Nmax or the separation between the two beams, it is most likely faster to combine the beams after translating the individual beams. If many translations/rotations are needed, it is most likely faster to create a single beam and apply the translations to that beam.

Creating a single beam

In order to create a single beam, we first need to translate the two beams from the origin to their displaced locations. In order to be able to translate a beam multiple times, we need to keep the higher order multipole terms after the first translation. To calculate the Nmax we need for this translation, we convert the old Nmax to a radius and add the displacement before converting back to a Nmax. We then request that `translateXyz` produce a beam with the new Nmax.

```
% Calculate new Nmax
Nmax = ott.utils.ka2nmax(ott.utils.nmax2ka(beam.Nmax) ...
    + displacement*T.k_medium);

% Change the Nmax and create the two beams
beam1 = beam.translateXyz([-displacement; 0; 0], 'Nmax', Nmax);
beam2 = beam.translateXyz([displacement; 0; 0], 'Nmax', Nmax);
```

To combine the beams, we simply add them (which adds the `a` and `b` coefficients of each beam). We can change the relative phase between the two beams by simply multiplying a complex phase term by one of the beams.

```
% Add the beams
nbeam = beam1 + beam2 * phase;
```

The force can then be calculated from this combined beam:

```
% Calculate the force along the x-axis
fx1 = ott.forcetorque(nbeam, T, 'position', [1;0;0] * x);
```

Combining after translations/rotations

Instead of applying multiple translations to each beam, it is also possible to apply only a single translation to each beam. There is not currently any automated method for doing this in the toolbox, the easiest way is to add the translations and force calculation to a for loop:

```
for ii = 1:length(x)

  % Translate and add the beams
  beam1 = beam.translateXyz([x(ii)+displacement; 0; 0]);
  beam2 = beam.translateXyz([x(ii)-displacement; 0; 0]);
  tbeam = beam1 + beam2 * phase;

  % Scatter the beam and calculate the force
  sbeam = T * tbeam;
  fx2(:, ii) = ott.forcetorque(tbeam, sbeam);
end
```

3.3.2 Combining incoherent beams

For incoherent beams we just need to sum the force from each individual beam. Similarly to coherent beams, we can apply the translation to the individual beams or to a combined incoherent beam object.

Operations on individual beams

For incoherent beams, we can use the `ott.forcetorque` method to do the translations and force calculation. Unlike coherent beams, we don't need to combine the beams after translating the beam.

```
fx3 = ott.forcetorque(beam, T, ...
    'position', [1;0;0] * x + [displacement; 0; 0]);
fx3 = fx3 + ott.forcetorque(beam * phase, T, ...
    'position', [1;0;0] * x - [displacement; 0; 0]);
```

Applying the same operations on both beams

As with coherent beams, combining both beams requires translating the beam and keeping the higher Nmax terms. We can then combine the beams into a single `ott.Bsc` object and use the `ott.forcetorque` method to apply the translations and calculate the forces. When `ott.forcetorque` is called with a `ott.Bsc` object containing multiple beams, it produces a 3-Dimensional matrix with the third dimension corresponding to the force from each beam in `ott.Bsc`. To calculate the incoherent force, we simply need to sum over the third dimension of this matrix.

```
% Calculate new Nmax
Nmax = ott.utils.ka2nmax(ott.utils.nmax2ka(beam.Nmax) ...
    + displacement*T.k_medium);

% Change the Nmax and create the two beams
beam1 = beam.translateXyz([-displacement; 0; 0], 'Nmax', Nmax);
beam2 = beam.translateXyz([displacement; 0; 0], 'Nmax', Nmax);

beamc = beam1.append(beam2);

fx4 = ott.forcetorque(beamc, T, 'position', [1;0;0] * x);
fx4 = sum(fx4, 3);
```

3.4 Creating a custom beam

There are multiple ways to create your own beam in the toolbox. If your beam can be described as a mixture of other beams already in the toolbox, you can create the beam by simply adding these beams, see the combining beams page. If you already have the beam shape coefficients, the easiest way is to create a new instance of `ott.Bsc`. If you don't know the coefficients but you know the near- or far-field representation of the beam then you can use the static functions in `ott.BscPointmatch`.

A more complicated way to create your own beam is to create your own class which inherits either from `ott.Bsc` or `ott.BscPointmatch`. This offers greater flexibility and allows you to easily add additional checks on user input.

3.4.1 Using `ott.Bsc` directly

The `ott.Bsc` class can be instantiated with arrays of the beam shape coefficients, this can be useful if you want to create a beam with particular multipole components or if you are using another method to calculate the beam shape coefficients. For example, to create a incoming beam with only quadrapole coefficients:

```
a = [0, 0, 0, 0, 1, 0, 0, 0];  % [3 dipole, 5 quadrapole]
b = 1i * a;
basis = 'incoming';
type = 'incident';
beam = ott.Bsc(a, b, basis, type);
```

The `ott.Bsc` object can then be used with `Tmatrix` objects or for visualisation of the beam:

```
figure();
beam.visualiseFarfieldSphere('type', '3dpolar', 'field', 'E2')
```

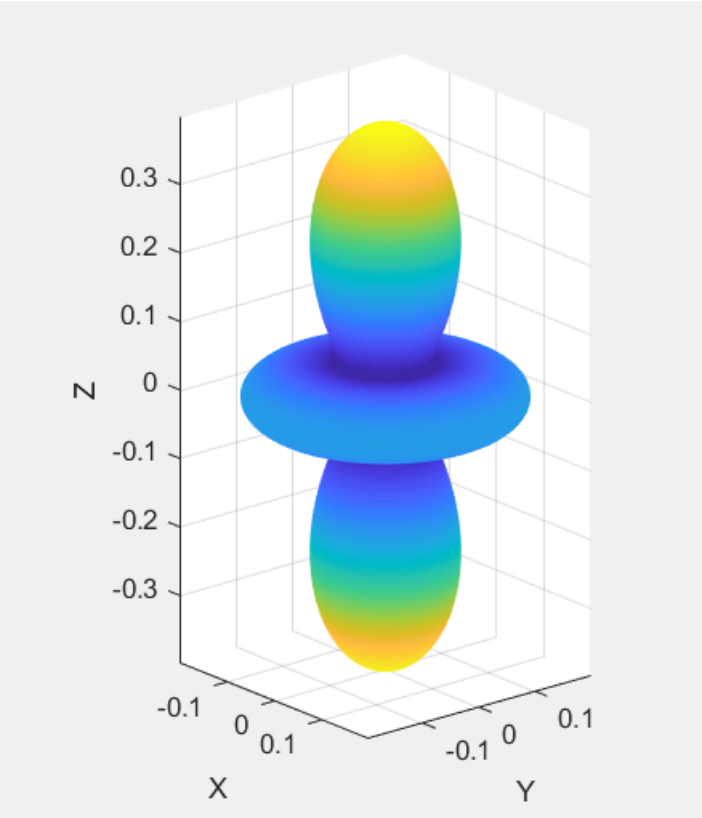


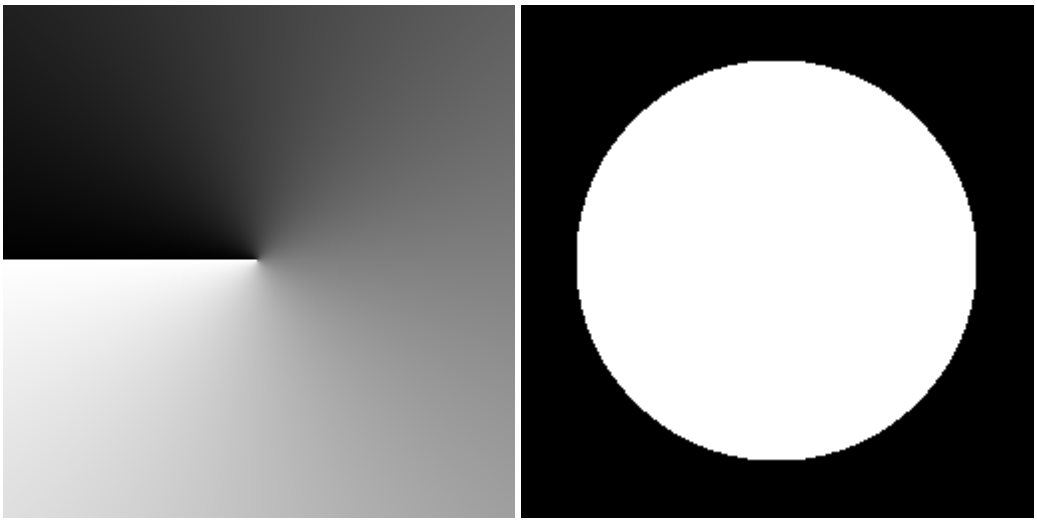
Fig. 3.6: dipole beam visualization

3.4.2 Creating a beam with point matching

If you don't know the VSWF expansion of your beam but you are able to calculate or measure the phase and amplitude in the near-field or far-field, you can use `ott.BscPointmatch` or `ott.BscPmParaxial`. `ott.BscPointmatch` contains two static methods for calculating the beam shape coefficients from the near-fields or far-fields. `ott.BscPmParaxial` uses the far-field method from `ott.BscPointmatch` and provides the code to calculate the far-field coordinates for a 2-D image of the fields at the back focal plane of the objective.

`ott.BscPmParaxial` in far-field

This class can be used to create beam shape coefficients from images at the back-aperture of the microscope objective. For this example, we will use the following images for the phase and amplitude:



These images may work better if spatial filtering is applied to remove higher frequency components from the images, this can be achieved using `imgaussfilt`. In this example, we are going to generate a circularly polarised beam. To do this, we first load our images and assemble them into a complex E field matrix:

```
% read, scale and convert to double (might need rgb2gray depending on file)
imPhase = double(imread('phase.png')) ./ 255.0;
imAmplitude = double(imread('amplitude.png')) ./ 255.0;

E = imAmplitude;    % x polarisation
E(:, :, 2) = 1i * imAmplitude;  % y polarisation
E = E .* exp(1i * 2 * pi * imPhase);
```

For a vector beam, we could instead use separate images for the x and y polarisation of each pixel. To use the `BscPmParaxial` class, we need to provide the complex far-field matrix, the numerical aperture of the beam, properties of the beam (such as wavelength and frequency), and a mapping function describing how the image coordinates are mapped to the spherical coordinates for the far-field of the beam.

The E-field matrix describes the field at the back aperture of the objective. The matrix is mapped onto a hemisphere, with a maximum angle defined by the numerical aperture of the objective. The centre of the hemisphere corresponds to the centre of the images. The radial coordinate extends from the centre of the image to the edge of the image (the corners of the image do not contribute to the beam). `BscPmParaxial` supports the following mapping functions

- `'sintheta'` (default) image radius proportional to `sin(theta)`
- `'tantheta'` image radius proportional to `tan(theta)`

- 'theta' image radius proportional to theta where theta is the polar coordinate on the hemisphere.

```
index_medium = 1.0;
wavelength0= 1064e-9;
omega = 3e8 / wavelength0 * 2 * pi;
NA = -1.0;   % sign of NA determines beam direction
Nmax = 30;   % higher spatial frequencies require higher NA
beam = ott.BscPmParaxial(NA, E, ...
    'index_medium', index_medium, ...
    'Nmax', Nmax, ...
    'wavelength0', wavelength0, ...
    'omega', omega);
beam.basis = 'regular';
figure();
beam.visualise('axis', 'y');
```

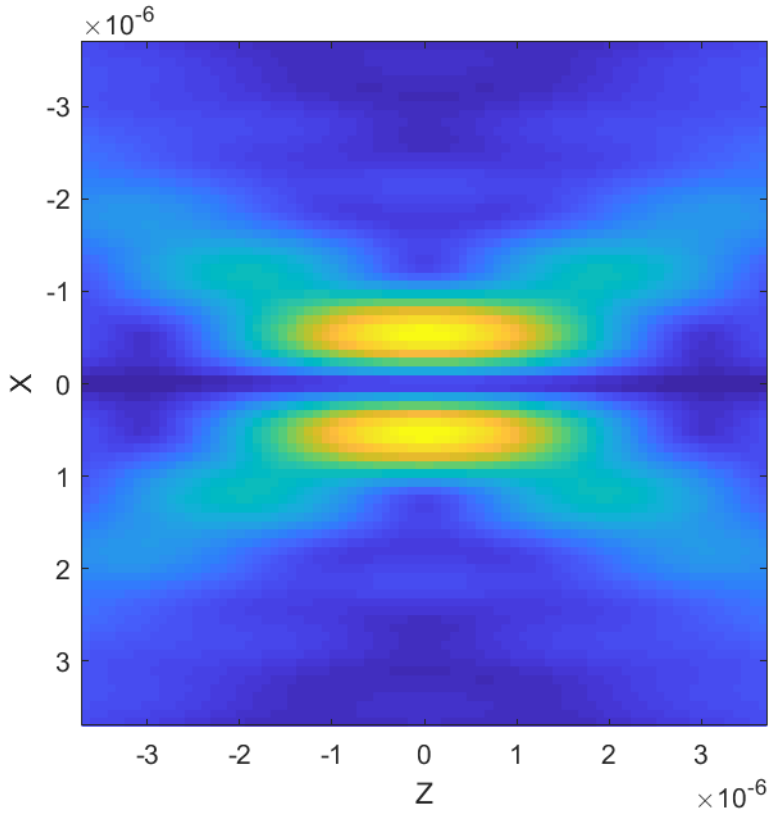


Fig. 3.7: output beam

This method can be slow since the coefficient matrix for point matching is calculated each time. To speed up the method for multiple beam calculation, BscPmParaxial supports keeping the coefficient matrix.

```
beam1 = ott.BscPmParaxial(..., 'keep_coefficient_matrix', true);
beam2 = ott.BscPmParaxial(..., 'beamData', beam1);
```

Far-field

`ott.BscPointmatch/bsc_farfield` can be used to calculate the beam shape coefficients from the mode indices, coordinates and E-field. The resulting BSC can be wrapped in an `ott.Bsc` object (see above).

```
% Calculate mode indices
mode_indexes=[1:Nmax*(Nmax+2)].';
[nn,mm]=ott.utils.combined_index(mode_indexes);

% Calculate e_field in theta/phi coordinates
[theta,phi]=ott.utils.angulargrid(2*(Nmax+1),2*(Nmax+1));
e_field = ...;

[a, b] = ott.BscPointmatch.bsc_farfield(nn, mm, e_field(:), theta(:), phi(:));
```

Near-field

`ott.BscPointmatch/bsc_focalplane` calculates the beam shape coefficients in a Cartesian coordinate system centred around the focal plane. To use the method, you must specify the mode indices, field locations and field vectors in Cartesian coordinates.

```
% Calculate mode indices
mode_indexes=[1:Nmax*(Nmax+2)].';
[nn,mm]=ott.utils.combined_index(mode_indexes);

% Calculate e_field
[xx, yy, zz] = meshgrid(linspace(-1, 1), linspace(-1, 1), linspace(-1, 1));
[r, theta, phi] = ott.utils.xyz2rtp(xx(:), yy(:), zz(:));
kr = r .* 2 * pi / lambda;
e_field = [Ex(:); Ey(:); Ez(:)];

[a, b] = ott.BscPointmatch.bsc_focalplane(nn, mm, e_field, kr, theta, phi);
```

Custom `ott.BscPointmatch` class

Although the `bsc_focalplane` and `bsc_pointmatch` functions can be used directly, their use is rather cumbersome for regular use. In order to offer a simplified interface for these objects you can inherit from `ott.BscPointmatch`. This allows you to define all the methods needed to create the beam within the class, directly set the beam shape coefficients and provide a user interface which provides only physically motivated parameters.

In this section we will go through an example of creating a point-matching method for annular beams. For other examples, look at the `ott.BscPm*` class implementations.

All beam classes should inherit from `ott.Bsc`. Point-matching beams should implement from `ott.BscPointmatch` which inherits from `ott.Bsc`. For our annular class we inherit from `ott.BscPointmatch`. The outline for our class is shown bellow:

```
classdef BscPmAnnular < ott.BscPointmatch
  % Documentation...

  properties (SetAccess=protected)
    % Beam properties...
```

(continues on next page)

(continued from previous page)

```
  end

  methods (Static)
    % Methods which can't access properties...
  end

  methods
    % Methods which can access properties
  end
end
```

We declare the properties as `SetAccess=protected`, this means that the properties can only be set by functions defined in the class method blocks. For annular beams, we define one property, the numerical aperture describing the inner and outer radius of the annular.

```
properties (SetAccess=protected)
  NA     % Numerical aperture [r1, r2]
end
```

To calculate the beam profile, we will implement a static method which takes as input the two NA and outputs zeros or ones for the amplitude of the beam:

```
methods (Static)
  function im = generatePattern(r1, r2)

    [xx, yy] = meshgrid(linspace(-1, 1), linspace(-1, 1));
    rr = sqrt(xx.^2 + yy.^2);

    im = double(rr > r1 & rr < r2);
  end
end
```

The main method the user will use to interact with the beam is the constructor. The constructor will include the numerical aperture and optional named arguments. We use an `inputParser` to handle the named arguments. For the beam wavenumber, we can use the `ott.Bsc/parser_k_medium` function.

```
methods
  function beam = BscPmAnnular(NA, varargin)

    % Call base class constructor
    beam = beam@ott.BscPointmatch();

    p = inputParser();
    p.addParameter('Nmax', 30);

    % Parameters for frequency and wavenumber
    p.addParameter('omega', 2*pi);
    p.addParameter('wavelength0', 1);
    p.addParameter('k_medium', []);
    p.addParameter('index_medium', []);
    p.addParameter('wavelength_medium', []);
    p.parse(varargin{:});
```

(continues on next page)

(continued from previous page)

```
    % Store/get parameters
    Nmax = p.Results.Nmax;
    beam.k_medium = ott.Bsc.parser_k_medium(p, 2*pi);
    beam.omega = p.Results.omega;
    beam.NA = NA;

    if isempty(p.Results.index_medium)
      nMedium = 1.0;
    else
      nMedium = p.Results.index_medium;
    end

    % Calculate the radius from NA
    NAonm = NA/nMedium;

    % Calculate the pattern
    im = beam.generatePattern(NAonm(1), NAonm(2));

    % Calculate the coordinates in the far-field
    [xx, yy] = meshgrid(linspace(-1, 1), linspace(-1, 1));
    rr = sqrt(xx.^2 + yy.^2);
    phi = atan2(yy, xx);
    theta = asin(rr);

    % Remove points outside NA=1
    phi = phi(rr < 1);
    theta = theta(rr < 1);
    im = im(rr < 1);

    % Transform im into e_field
    Et = sign(cos(theta)).*cos(phi).*im;
    Ep = -sin(phi).*im;
    e_field=[Et(:); Ep(:)];

    % Calculate mode indices
    mode_indexes=[1:Nmax*(Nmax+2)].';
    [nn,mm]=ott.utils.combined_index(mode_indexes);

    % Calculate BSC
    [beam.a, beam.b] = ott.BscPointmatch.bsc_farfield(nn, mm, e_field(:), theta(:),␣
↪phi(:));

    % Set other BSC properties
    beam.type = 'incident';
    beam.basis = 'regular';
  end
end
```

This class doesn't implement exactly the same functionality as the `ott.BscPmAnnular` class, but it shows how a class could be implemented to wrap the `bsc_farfield` method.

Full class definition

```
classdef BscPmAnnular < ott.BscPointmatch
  % Documentation...

  properties (SetAccess=protected)
    NA     % Numerical aperture [r1, r2]
  end

  methods (Static)
    function im = generatePattern(r1, r2)

      [xx, yy] = meshgrid(linspace(-1, 1), linspace(-1, 1));
      rr = sqrt(xx.^2 + yy.^2);

      im = double(rr > r1 & rr < r2);
    end
  end

  methods
    function beam = BscPmAnnular(NA, varargin)

      % Call base class constructor
      beam = beam@ott.BscPointmatch();

      p = inputParser();
      p.addParameter('Nmax', 20);

      % Parameters for frequency and wavenumber
      p.addParameter('omega', 2*pi);
      p.addParameter('wavelength0', 1);
      p.addParameter('k_medium', []);
      p.addParameter('index_medium', []);
      p.addParameter('wavelength_medium', []);
      p.parse(varargin{:});

      % Store/get parameters
      Nmax = p.Results.Nmax;
      beam.k_medium = ott.Bsc.parser_k_medium(p, 2*pi);
      beam.omega = p.Results.omega;
      beam.NA = NA;

      if isempty(p.Results.index_medium)
        nMedium = 1.0;
      else
        nMedium = p.Results.index_medium;
      end

      % Calculate the radius from NA
      NAonm = NA/nMedium;

      % Calculate the pattern
      im = beam.generatePattern(NAonm(1), NAonm(2));

      % Calculate the coordinates in the far-field
```

(continues on next page)

(continued from previous page)

```
      [xx, yy] = meshgrid(linspace(-1, 1), linspace(-1, 1));
      rr = sqrt(xx.^2 + yy.^2);
      phi = atan2(yy, xx);
      theta = asin(rr);

      % Remove points outside NA=1
      phi = phi(rr < 1);
      theta = theta(rr < 1);
      im = im(rr < 1);

      % Transform im into e_field
      Et = sign(cos(theta)).*cos(phi).*im;
      Ep = -sin(phi).*im;
      e_field=[Et(:); Ep(:)];

      % Calculate mode indices
      mode_indexes=[1:Nmax*(Nmax+2)].';
      [nn,mm]=ott.utils.combined_index(mode_indexes);

      % Calculate BSC
      [beam.a, beam.b] = ott.BscPointmatch.bsc_farfield(nn, mm, e_field(:), theta(:),␣
↪phi(:));

      % Set other BSC properties
      beam.type = 'incident';
      beam.basis = 'regular';
    end
  end
end
```

3.4.3 Creating a custom `Bsc` class

For other beam shape coefficient definitions, it is possible to create a custom class which inherits from `ott.Bsc`. The implemention for this class will be very similar to the `BscPmAnnular` class shown above. For examples, see `ott.BscBessel` and `ott.BscPlane`.

```
classdef BscCustomClass < ott.Bsc
  % Documentation...

  properties (SetAccess=protected)
    % Beam properties...
  end

  methods (Static)
    % Methods which can't access properties...
  end

  methods
    % Methods which can access properties

    function beam = BscCustomClass()
```

(continues on next page)

(continued from previous page)

```
      % Call the base class constructor
      beam = beam@ott.Bsc();

      % Implementation...
    end
  end
end
```

3.5 Creating a custom T-matrix

Contents

Information on creating a custom T-matrix code.

3.6 Calculating forces with the GUI

In this example we calculate the forces on a spherical particle for different axial and radial displacements in a Gaussian beam. We use the GUI for calculating the forces, generating the beam shape coefficients for the beam and calculating the T-matrix for the particle. This example produces similar output to the spherical particle example script (`examples/example_sphere.m`) without needing to write a single line of Matlab code.

Before starting this example, ensure you have OTT installed and are able to launch the Launcher GUI, see *Getting Started* for details.

In this example, we generate the beam shape coefficients for a Gaussian beam, calculate the T-matrix for a spherical particle and

Contents

3.6.1 Generating a Gaussian beam

To generate a LG beam, we will use *LG beam* application, which uses `ott.BscPmGauss` to generate Gaussian and LG beams. Open the Launcher and select **BSC > LG beam > Launch** to open the *LG beam* application. The window shown in Fig. 3.8 should display.

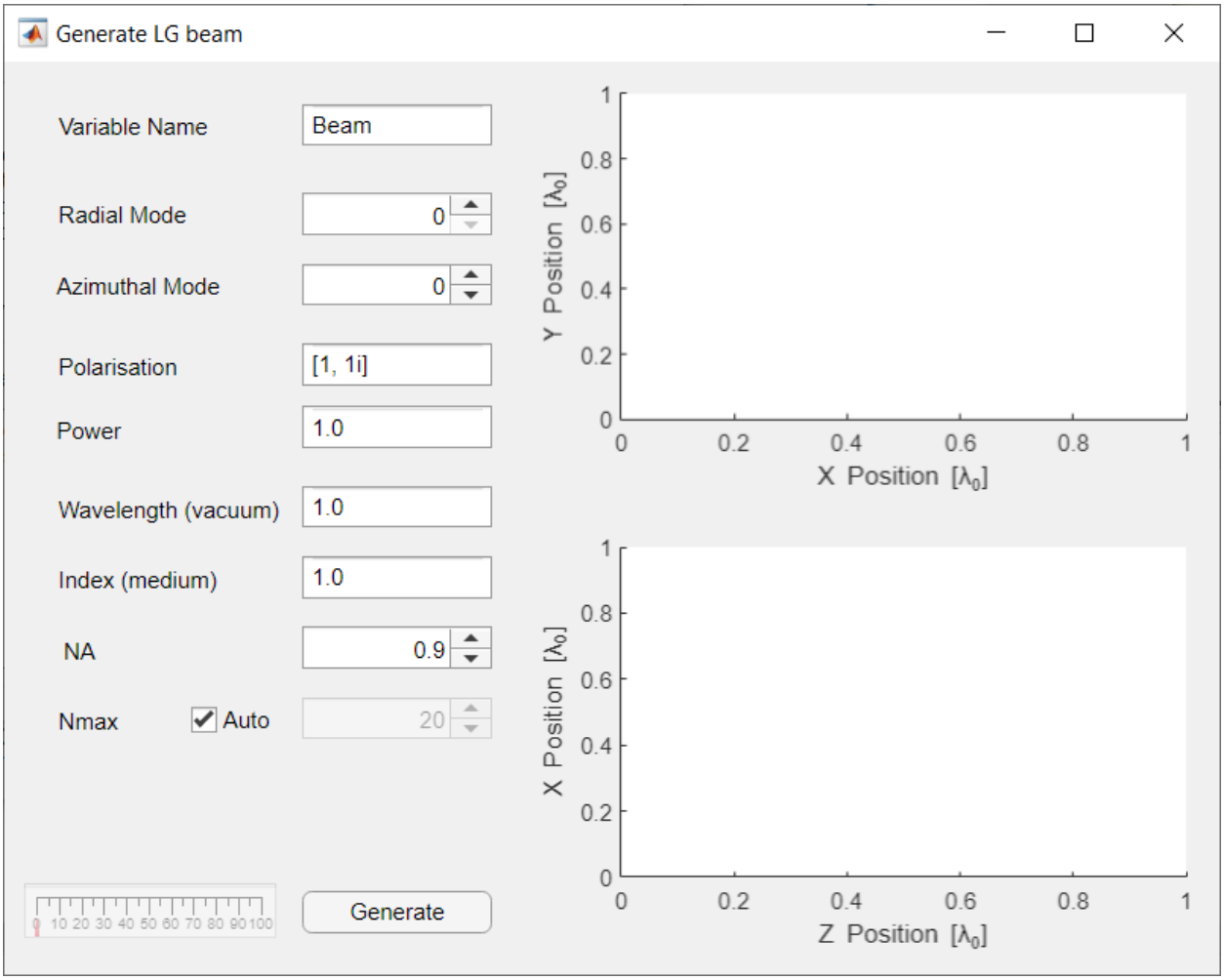


Fig. 3.8: The Generate LG Beam GUI with all the default parameters.

For a Gaussian beam, we want to set the radial and azimuthal modes to 0 (the default). For this example we will use a circularly polarised beam, so we set the polarisation to `[1, 1i]` (the default). For the vacuum wavelength we will enter `1064.0e-9`, corresponding to 1064nm, and we will use the refractive index of water for the medium (enter `1.33` into the *Index (medium)* field). Finally, for the NA we will use `1.02`.

For most Gaussian and LG beam we do not need to explicitly set N_{max}. A general rule of thumb is Nmax should be large enough to surround the beam focus, so most of the beam power goes through a circle of radius $N_{max}k_{medium}$ where k_{medium} is the wavenumber in the medium.

Once all the parameters have been set, click **Generate**. Depending on you computer this may take a couple of seconds or a few minutes. The resulting output is shown in figure Fig. 3.9. Additionally, a variable should be created in the Matlab workspace for our new beam (we will use this variable later).

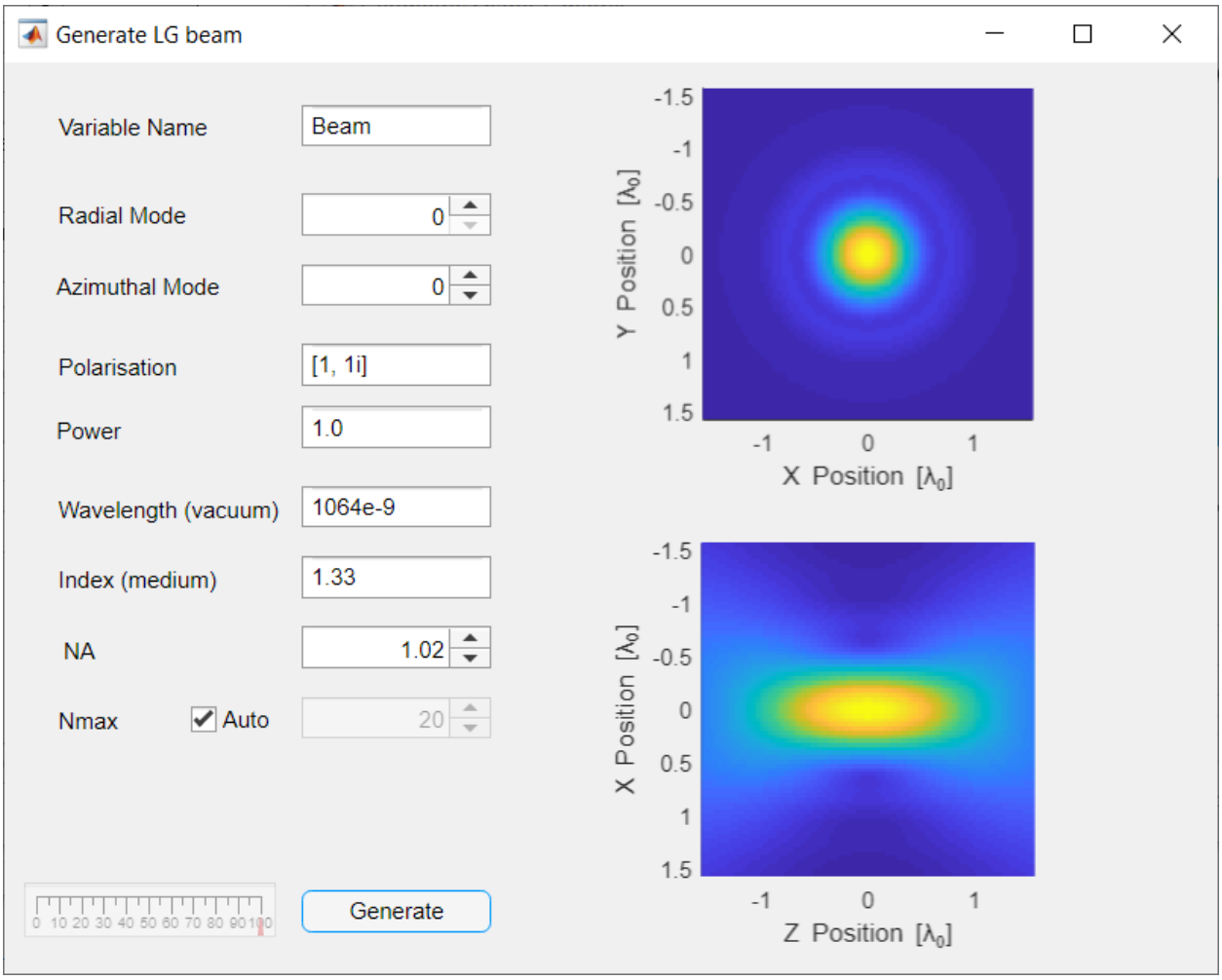


Fig. 3.9: The Generate LG Beam GUI after clicking the Generate button should now display the beam transverse and axial field distributions.

3.6.2 Generating the T-matrix

To generate the T-matrix representing the scattering by a spherical particle we can use the *Geometric shape* GUI. This GUI attempts to use the appropriate T-matrix method for the given geometric shape. To launch the GUI, open the launcher and select **T-matrix > Geometric shape > Launch**.

For this example, we want to simulate a spherical polystyrene particle with refractive index 1.59. For the relative refractive index field we enter `1.59/1.33`, this expression is evaluated in the Matlab workspace and should produce about 1.2. It is also possible to enter a variable name, for instance, if we had a variable called `index_medium` we could have written `1.59/index_medium`. For the wavelength we enter `1064.0e-9/1.33`. Make sure the sphere option is selected and set the radius to 500 nm (i.e., *5e-7*). For a spherical particle, we leave the Nmax and Method options with their default values.

Finally click Generate. The progress bar should change and a T-matrix object should be added to the Matlab workspace. For spherical particles this shouldn't take very long, however for other shapes this could take hours depending on the shape and chosen method. The progress bar is approximate and not supported by all methods. Figure Fig. 3.10 shows the GUI after clicking generate.

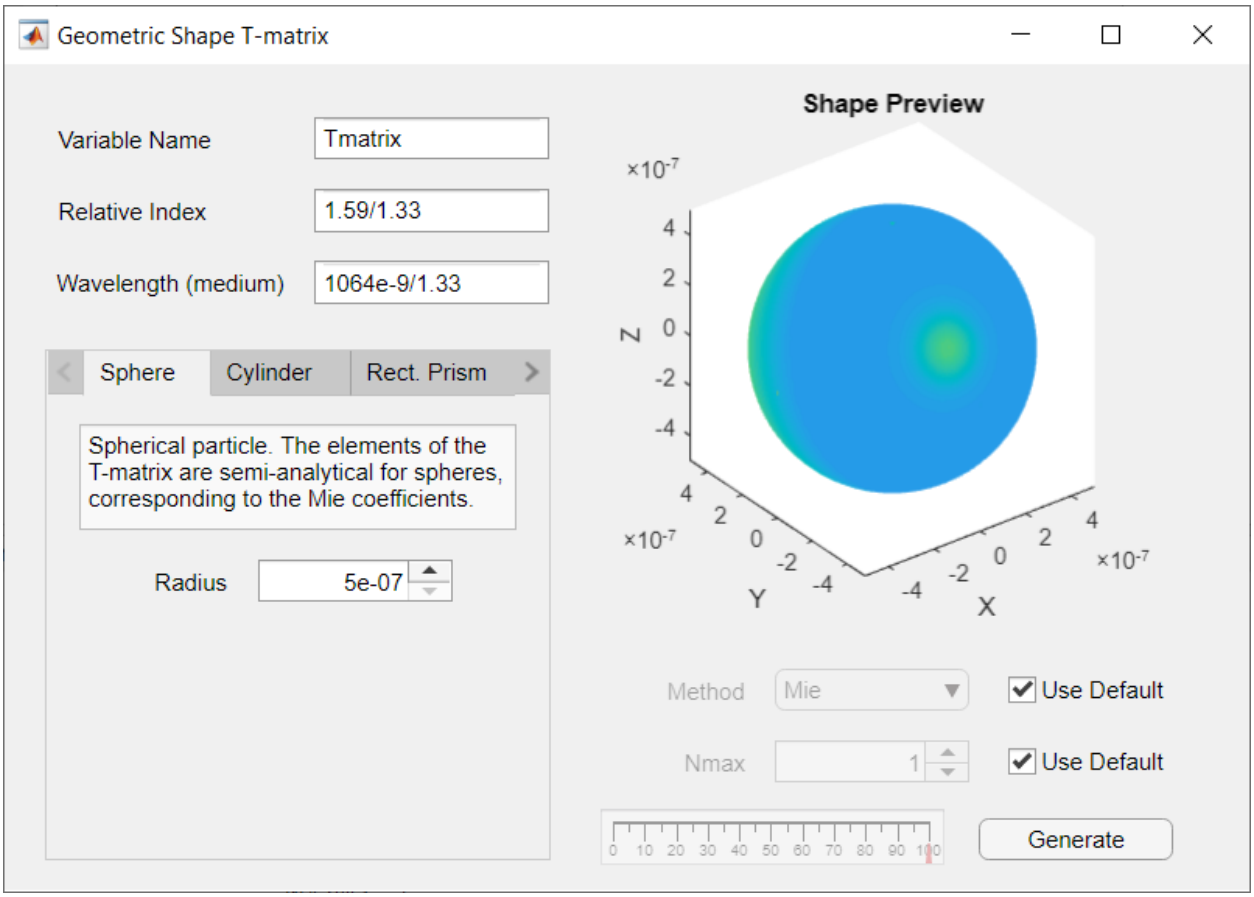


Fig. 3.10: The Generate Shape T-matrix GUI after clicking generate looks almost the same as before clicking generate. The shape preview is provided when the shape proeprties are set.

3.6.3 Calculating forces

The final part of this example is calculating the force for different axial and radial displacements. To do this, we need `Beam` and `Tmatrix` variables in Matlabs workspace, these can be generated by following the above instructions or by directly calling the appropriate functions/classes. To translate the beam and calculate the forces we use the *Calculate Force/Torque Profiles* GUI. From the Launcher select **Tools > Force Profile > Launch**.

The GUI has two drop down boxes for selecting the Beam and T-matrix variables. These fields are only updated when you launch the GUI: If you created your T-matrix or Beam after launching this GUI, you can type in the beam and T-matrix names manually. For this example, we select the `Beam` and `Tmatrix` variables created in the previous steps.

Optionally, we can specify an output variables. This variable name is used to save the generated force/torque data in the Matlab workspace, useful if you would like to save the data or generate your own plots with the raw data.

The remaining options are for specifying the location and translation/rotation. The units for the range values depend on the type of direction, for translations the units are beam wavelengths. For rotations, the units are radians.

Once you have specified your desired range, click generate to calculate the forces and generate a graph. Example output is shown in figure Fig. 3.11 for translation along the axial direction.

The units for the force and torque depend on the units chosen for the beam power. In this example, the beam power was left at its default value (1.0) and the units for the force are the dimensionless trapping efficiency, which can be converted to Newtons by multiplying with nP/c where n is the refractive index of the medium, P is the power and c is the speed of light in vacuum.

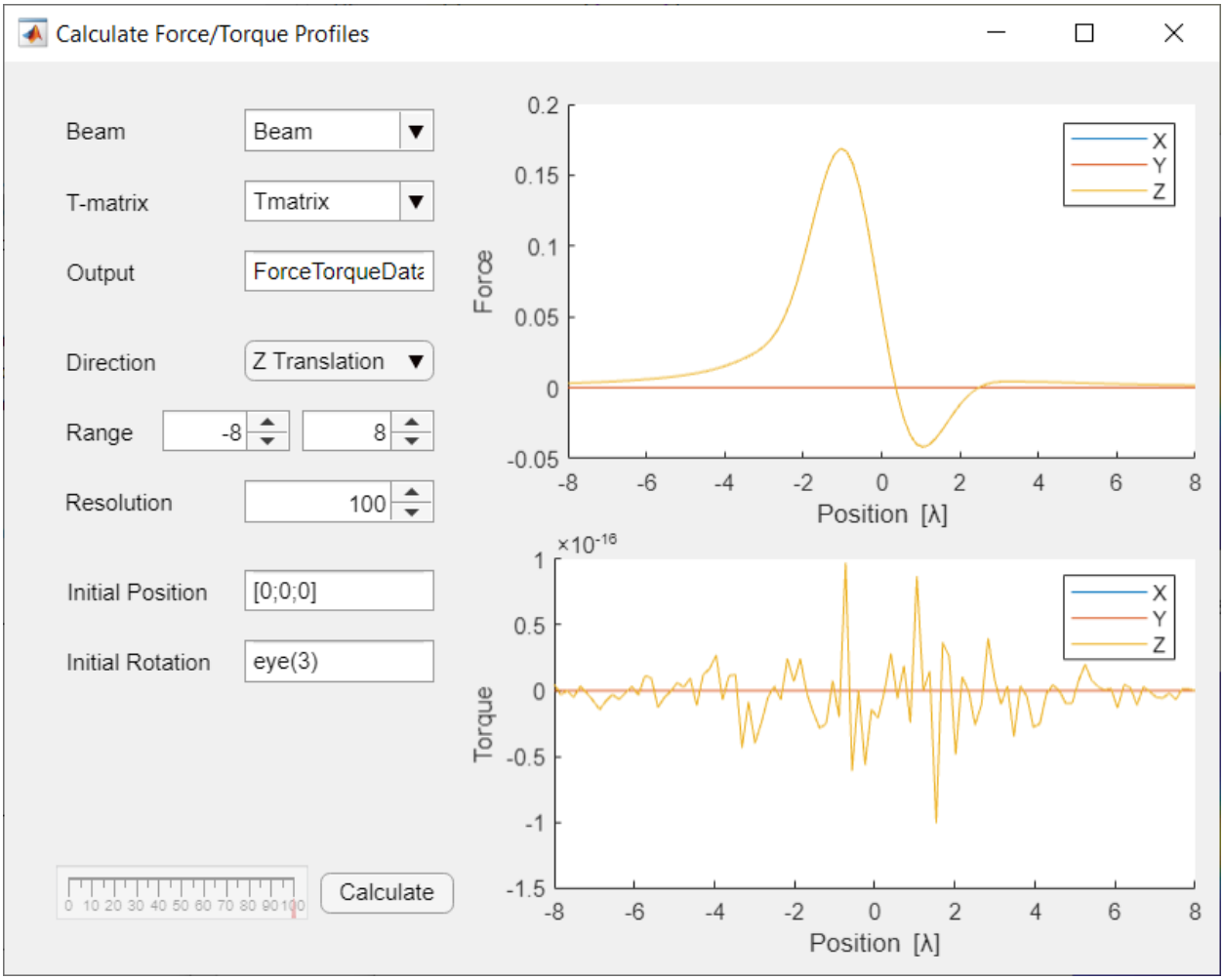


Fig. 3.11: The force profile for a spherical particle in a Gaussian beam when translated along the beam axis. There is no torque on this particle and the displayed torque is noise from the numerical calculation.

CHAPTER

FOUR

REFERENCE

This section contains information about the different functions and packages contained in the toolbox.

4.1 *Bsc* classes

This section contains information about the beam shape coefficient classes (Bsc) currently implemented in the toolbox. These classes can be used to describe optical tweezers beams in a basis of vector spherical wave functions. The classes provide functions for translating beams, visualising beams and overloads for adding beams. Most of the core functionality is provided in the base class `ott.Bsc`. Classes inheriting from this class typically only need to define the beam creation code specific to that type of beam.

Contents

4.1.1 Bsc

`ott.Bsc` is the base class for objects representing beam shape coefficients (BSC) including `BscPmGauss` and `BscPmParaxial`. The class can also be used directly to package a set of existing BSC for use with other functions in the toolbox, for example

```
a = [1; 0; 0]; b = 1i.*a;
basis = 'incoming';
type = 'incident';
beam = ott.Bsc(a, b, basis, type);
```

would create a new beam with *Nmax = 1* (i.e. 3 coefficients for *a* and *b*) with the incoming vector spherical wave function basis, representing a incident beam. For further information about creating custom beams, see the *Creating a custom beam* example.

class ott.**Bsc**(*a*, *b*, *basis*, *type*, *varargin*)

Bsc abstract class representing beam shape coefficients

Most quantities have SI dimensions. Any SI units can be used for these quantities (for example m or microns) as long as the units are consistent.

Beam power (i.e., the Bsc.power property) can be converted to SI units by multiplying by *1/(2*Z*k^2)* where Z is the medium impedance and k is the medium wave number (this does not affect force calculation). Changing the beam power is not recommended in this or earlier versions of the toolbox since this effects both accuracy and run-time. This behaviour will be changed (fixed) in a future release.

Properties

- a – Beam shape coefficients a vector
- b – Beam shape coefficients b vector
- type – Beam type (incident, scattered, total)
- basis – VSWF beam basis (incoming, outgoing or regular)
- Nmax – Truncation number for VSWF coefficients
- power – Power of the beam [M*L^2/S^3]
- Nbeams – Number of beams in this Bsc object
- wavelength – Wavelength of beam [L]
- speed – Speed of beam in medium [L/T]
- omega – Angular frequency of beam [2*pi/T]
- k_medium – Wavenumber in medium [2*pi/L]
- dz – Absolute cumulative distance the beam has moved

Methods

- append – Joins two beam objects together
- sum – Merge the BSCs for the beams contained in this object
- translateZ – Translates the beam along the z axis
- translateXyz – Translation to xyz using rotations and z translations
- translateRtp – Translation to rtp using rotations and z translations
- farfield – Calculate fields in farfield
- emFieldXyz – Calculate field values in cartesian coordinates
- emFieldRtp – Calculate field values in spherical coordinates
- getCoefficients – Get the beam coefficients [a, b]
- getModeIndices – Get the mode indices [n, m]
- totalField – Calculate the total field reprsentation of the beam
- scatteredField – Calcualte the scattered field representation of the beam
- visualise – Generate a visualisation of the beam near-field
- visualiseFarfield – Generate a visualisation of the beam far-field
- visualiseFarfieldSlice – Generate scattering slice at specific angle
- visualiseFarfieldSphere – Generate spherical surface visualisation
- intensityMoment – Calculate moment of beam intensity in the far-field

Static methods:

- make_beam_vector – Convert output of bsc_* functions to beam coefficients

See also Bsc, ott.BscPmGauss, ott.BscPlane.

`Bsc`(*a*, *b*, *basis*, *type*, *varargin*)

BSC construct a new beam object

beam = Bsc(a, b, basis, type, ...) constructs a new beam vector. Useful if you have a specific set of a/b coefficients that you want to wrap in a beam object.

Basis: incoming, outgoing or regular Type: incident, scattered, total, internal

Optional named arguments:

k_medium n Wavenumber in medium (default: 2*pi) omega n Angular frequency (default: 2*pi) dz n Initial displacement of the beam (default: 0) like beam Construct this beam to be like another beam

`static GetVisualisationData`(*field_type*, *xyz*, *rtp*, *vxyz*, *vrtp*)

Helper to generate the visualisation data output. This function is not intended to be called directly, instead see `visualise()` or `visualiseFarfield()`.

Usage

GetVisualisationData(field_type, xyz, rtp, vxyz, vrtp) Takes a field_type string, the coordinates (either xyz or rtp), and the data values (either vxyz or vrtp).

Parameters

- xyz, rtp, vxyz, vrtp – (Nx3 numeric) Coordinates in a suitable form to be passed to *ott.utils.xyz2rtp* and similar functions. Pass empty arrays for unused values.
- field_type – (enum) Type of field to calculate. Supported types include:
  - 'irradiance' – $\sqrt{|Ex|^2 + |Ey|^2 + |Ez|^2}$
  - 'E2' – $|Ex|^2 + |Ey|^2 + |Ez|^2$
  - 'Sum(Abs(E))' – $|Ex| + |Ey| + |Ez|$
  - Re(Er), Re(Et), Re(Ep), Re(Ex), Re(Ey), Re(Ez)
  - Abs(Er), Abs(Et), Abs(Ep), Abs(Ex), Abs(Ey), Abs(Ez)
  - Arg(Er), Arg(Et), Arg(Ep), Arg(Ex), Arg(Ey), Arg(Ez)

`translateXyz`(*beam*, *varargin*)

Translate the beam given Cartesian coordinates.

Units for the coordinates should be consistent with the beam wave number (i.e., if the beam was created by specifying wavelength in units of meters, distances here should also be in units of meters).

Usage

tbeam = beam.translateXyz(xyz) translate the beam to locations given by the `xyz` coordinates, where `xyz` is a 3xN matrix of coordinates.

tbeam = beam.translateXyz(Az, Bz, D) Translate the beam using z-translation and rotation matrices.

[tbeam, Az, Bz, D] = beam.translateXyz(...) returns the z-translation matrices `Az, Bz`, the rotation matrix `D`, and the translated beam `tbeam`.

[tbeam, A, B] = beam.translateXyz(...) returns the translation matrices `A, B` and the translated beam.

[tbeam, AB] = beam.translateXyz(...) returns the translation matrices `A, B` packaged so they can be directly applied to a beam using `tbeam = AB * beam`.

tbeam = beam.translateXyz(..., 'Nmax', Nmax) specifies the output beam `Nmax`. Takes advantage of not needing to calculate a full translation matrix.

`translateZ`(*beam*, *varargin*)

Translate a beam along the z-axis.

Units for the coordinates should be consistent with the beam wave number (i.e., if the beam was created by specifying wavelength in units of meters, distances here should also be in units of meters).

Usage

tbeam = beam.translateZ(z) translates by a distance `z` along the z axis.

[tbeam, A, B] = beam.translateZ(z) returns the translation matrices and the translated beam. See also `ott.Bsc.translate()`.

[tbeam, AB] = beam.translateZ(z) returns the `A, B` matrices packed so they can be directly applied to a beam: `tbeam = AB * beam`.

[...] = beam.translateZ(..., 'Nmax', Nmax) specifies the output beam `Nmax`. Takes advantage of not needing to calculate a full translation matrix.

`visualise`(*beam*, *varargin*)

Create a visualisation of the beam

Usage

visualise(...) displays an image of the beam in the current figure window.

im = visualise(...) returns a image of the beam. If the beam object contains multiple beams, returns images for each beam.

Optional named arguments

- 'size' [x, y] Width and height of image
- 'field' type Type of field to calculate
- 'axis' ax Axis to visualise ('x', 'y', 'z') or a cell array with 2 or 3 unit vectors for x, y, [z].
- 'offset' offset Plane offset along axis (default: 0.0)
- '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].
- 'mask' func(xyz) Mask function for regions to keep in vis
- 'combine' (enum) If multiple beams should be treated as 'coherent' or 'incoherent' beams and their outputs added. incoherent may only makes sense if the field is an intensity. Default: [].

`visualiseFarfield`(*beam*, *varargin*)

Create a 2-D visualisation of the farfield of the beam

visualiseFarfield(...) displays an image of the farfield in the current figure window.

im = visualiseFarfield(...) returns a 2-D image of the farfield.

[im, data] = visualiseFarfield(..., 'saveData', true) returns the saved data that can be used for repeated calculation.

TODO: Should the data object instead be a callable object?

This would make the interface simpler.

Optional named arguments:

'size' [x, y] Size of the image 'direction' dir Hemisphere string ('pos' or 'neg'),

2-vector (theta, phi) or 3x3 rotation matrix.

'field' type Type of field to calculate 'mapping' map Mapping from sphere to plane ('sin', 'tan') 'range' [x, y] Range of points to visualise

'saveData' bool save data for repeated calculation (default: false) 'data' data data saved for repeated calculation. 'thetaMax' num maximum theta angle to include in image 'showVisualisation' bool show the visualisation in the

current figure (default: nargout == 0).

`visualiseFarfieldSlice`(*beam*, *phi*, *varargin*)

Generate a 2-D scattering plot of the far-field.

Usage:

beam.visualiseFarfieldSlice(phi, ...) Generates a slice/polar plot of the far-field. In this version the slice is always aligned to the z-axis. *phi* specifies the rotation of the plane about the z-axis.

[theta, I] = beam.visualiseFarfieldSlice(phi, ...) calculate data for the visualisation but don't generate a visualisation unless *showVisualisation* is true.

Optional named arguments:

- field (enum) – The field type to visualise. Defaults to 'irradiance'. Not all field types support visualisation.
- normalise (logical) – If true, the calculated fields are normalised by the maximum calculated field value.
- ntheta (numeric) – Number of angular points to use. Defaults to 100.
- showVisualisation (logical) – If the visualisation should be shown. Defaults to *nargout == 0*.

`visualiseFarfieldSphere`(*beam*, *varargin*)

Generate a spherical surface visualisation of the far-field

beam.visualiseFarfieldSphere(phi)

Optional named arguments:

npts num Number of points to use for sphere surface normalise bool If intensity values should be normalised to 1 type str Type of visualisation to produce.

sphere (default) draw a sphere with intensity as color 3dpolar scale the radius by the intensity

4.1.2 BscPlane

Representation of a plane wave in VSWF coefficients

`class ott.BscPlane`(*theta*, *phi*, *varargin*)

BscPlane representation of a plane wave in VSWF coefficients

BscPlane properties:

theta Beam direction (polar angle) phi Beam direction (azimuthal angle) polarisation Beam polarisation [Etheta Ephi]

BscPlane methods:

translateZ Translates the beam and checks within beam range

Based on bsc_plane.m from ottv1.

See also BscPlane and ott.Bsc.

This file is part of the optical tweezers toolbox. See LICENSE.md for information about using/distributing this file.

4.1.3 BscPmGauss

Provides HG, LG and IG beams using point matching method

class ott.**BscPmGauss**(*varargin*)

BscPmGauss provides HG, LG and IG beams using point matching method

Properties

- gtype – Type of beam ('gaussian', 'lg', 'hg', or 'ig')
- mode – Beam modes (2 or 4 element vector)
- polarisation – Beam polarisation
- truncation_angle – Truncation angle for beam [rad]
- offset – Offset for original beam calculation
- angle – Angle of incoming beam waist
- angular_scaling – Angular scaling function (tantheta | sintheta)

See *ott.Bsc* for inherited properties.

This class is based on `bsc_pointmatch_farfield.m` and `bsc_pointmatch_focalplane.m` from OTT (version 1).

See also BscPmGauss.

BscPmGauss(*varargin*)

Construct a new IG, HG, LG or Gaussian beam.

Usage

BscPmGauss(...) constructs a new Gassian beam (LG00).

BscPmGauss(type, mode, ...) constructs a new beam with the given type. Supported types [mode]:

- 'lg' – Laguarre-Gauss [radial azimuthal]
- 'hg' – Hermite-Gauss [m n]
- 'ig' – Ince-Gauss [paraxial azimuthal parity elipticity]

Optional named parameters

- 'Nmax' – Truncation number for beam shape coefficients. If omitted, Nmax is initially set to 100, the beam is calculated and Nmax is reduced so that the power does not drop significantly.
- 'zero_rejection_level' – Level used to determine non-zero beam coefficients in far-field point matching. Default: 1e-8.
- 'NA' – Numerical aperture of objective
- 'polarisation' – Polarisation of the beam
- 'power' – Rescale the power of the beam (default: [])
- 'omega' – Optical angular frequency (default: 2*pi)
- 'k_medium' – Wave number in medium
- 'index_medium' – Refractive index of medium

- 'wavelength_medium' – Wavelength in medium
- 'wavelength0' – Wavelength in vacuum
- 'offset' – Offset of the beam from origin
- translation_method – Method to use when calculating translations. Can either be 'Default' or 'NewBeamOffset', the latter calculates new beam shape coefficients for every new position.
- angular_scaling (enum) – Angular scaling function. For a discussion of this parameter, see Documentation (*Point-matching and angle projections*).
  - 'sintheta' – angular scaling function is the same as the one present in standard microscope objectives. Preserves high order mode shape!
  - 'tantheta' – default angular scaling function, "small angle approximation" which is valid for thin lenses ONLY. Does not preserve high order mode shape at large angles.
- truncation_angle (numeric) – Adds a hard edge to the beam, this can be useful for simulating the back-aperture of a microscope objective. Default: `pi/2` (i.e. no edge).
- truncation_angle_deg – Same as *truncation_angle* but with degrees instead of radians.

4.1.4 BscPmParaxial

Calculate representation from farfield/paraxial beam

class ott.**BscPmParaxial**(*NA*, *E_ff*, *varargin*)

BscPmParaxial calculate representation from farfield/paraxial beam

Properties:

a (Bsc) Beam shape coefficients a vector b (Bsc) Beam shape coefficients b vector type (Bsc) Beam type (incoming, outgoing or scattered)

Methods:

translateZ (Bsc) Translates the beam along the z axis translateXyz (Bsc) Translation to xyz using rotations and z translations translateRtp (Bsc) Translation to rtp using rotations and z translations farfield (Bsc) Calculate fields in farfield emFieldXyz (Bsc) Calculate fields at specified locations set.Nmax (Bsc) Resize the beam shape coefficient vectors get.Nmax (Bsc) Get the current size of the beam shape vectors getCoefficients (Bsc) Get the beam coefficients [a, b] getModeIndices (Bsc) Get the mode indices [n, m] power (Bsc) Calculate the power of the beam

Based on paraxial_to_bsc from ottv1.

See also BscPmParaxial, ott.Bsc and examples/slm_to_focalplane

This file is part of the optical tweezers toolbox. See LICENSE.md for information about using/distributing this file.

4.1.5 BscPointmatch

Base class for BSC generated using point matching

ott.**BscPointMatch**

4.2 *Tmatrix* classes

This section describes the `Tmatrix` classes currently implemented in the toolbox.

4.2.1 Tmatrix

class ott.**Tmatrix**(*data*, *type*)

Class representing the T-matrix of a scattering particle or lens. This class can either be instantiated directly or used as a base class for defining custom T-matrix types.

This class is the base class for all other T-matrix object, you should inherit from this class when defining your own T-matrix creation methods. This class doesn't inherit from `double` or `single`, instead the internal array type can be set at creation allowing the use of different data types such as `sparse` or `gpuArray`.

This class is not a handle class, therefore, when using the class methods you need to store the resulting T-matrix output, for example:

```
tmatrix = ott.Tmatrix();
new_tmatrix = tmatrix.scattered();
```

Properties

- data – The T-matrix this class encapsulates
- type (enum) – Type of T-matrix (total or scattered)

Methods

- total() – Convert to a total-field T-matrix
- scattered() – Convert to a scattered-field T-matrix
- real – Extract real part of T-matrix
- imag – Extract imaginary part of T-matrix

Static methods

- simple() – Construct a simple particle T-matrix

See also Tmatrix, simple, *ott.TmatrixMie*.

Tmatrix(*data*, *type*)

Construct a new T-matrix object.

Usage

TMATRIX() leaves the data uninitialised.

TMATRIX(data, type) initializes the data with the matrix data.

Parameters

- data (numeric) – The T-matrix data. Typically a sparse or full matrix.
- type (enum) – Type of T-matrix. Must be 'internal', 'scattered' or 'total'.

Example

The following example creates an identity T-matrix which represents a particle which doesn't scatter light:

```
data = eye(16);
tmatrix = ott.Tmatrix(data, 'total');
```

static simple(*shape*, *varargin*)

Constructs a T-matrix for different simple particle shapes. This method creates an instance of one of the other T-matrix classes and is here only as a helper method.

Usage

SIMPLE(shape) constructs a new simple T-matrix for the given *ott.shapes.Shape* object.

SIMPLE(name, parameters) constructs a new T-matrix for the shape described by the name and parameters.

Supported shape names [parameters]

- 'sphere' – Spherical (or layered sphere) [radius]
- 'cylinder' – z-axis aligned cylinder [radius height]
- 'ellipsoid' – Ellipsoid [a b c]
- 'superellipsoid' – Superellipsoid [a b c e n]
- 'cone-tipped-cylinder' – [radius height cone_height]
- 'cube' – Cube [width]
- 'axisym' – Axis-symetric particle [rho(:) z(:)]

Optional named arguments

- method (enum) – Allows you to choose the preferred method to use for T-matrix calculation. Supported methods are 'mie', smarties', 'dda', 'ebcm', and 'pm'. Default: ''.
- method_tol (numeric) – Specifies the error tolerances, a number between (0, 1] to use for method selection. Smaller values correspond to more accurate methods. Default: *[]*.

Example

The following example creates a T-matrix for a cube with side length of 1 micron using a guess at the best available method. Illumination wavelength is 1064 nm, relative index 1.5/1.33:

```
tmatrix = ott.Tmatrix.simple('cube', 1.0e-6, ...
  'wavelength0', 1064e-9, ...
  'index_medium', 1.33, 'index_particle', 1.5);
```

4.2.2 TmatrixEbcm

Constructs a T-matrix using extended boundary conditions method.

class ott.**TmatrixEbcm**(*rtp*, *normals*, *ds*, *varargin*)

Constructs a T-matrix using extended boundary conditions method. Inherits from *ott.Tmatrix*.

TmatrixEbcm properties:
k_medium Wavenumber in the surrounding medium k_particle Wavenumber of the particle

This class is based on tmatrix_ebcm_axisym.m from ottv1.

See also TmatrixEbcm

4.2.3 TmatrixMie

Construct T-matrix from Mie scattering coefficients.

class ott.**TmatrixMie**(*radius*, *varargin*)

TmatrixMie construct T-matrix from Mie scattering coefficients

TmatrixMie properties:
radius The radius of the sphere the T-matrix represents k_medium Wavenumber in the trapping medium k_particle Wavenumber of the particle

This class is based on tmatrix_mie.m and tmatrix_mie_layered.m from ottv1.

4.2.4 TmatrixPm

Constructs a T-matrix using the point matching method.

class ott.**TmatrixPm**(*rtp*, *normals*, *varargin*)

TmatrixPm constructs a T-matrix using the point matching method

TmatrixPm properties:
k_medium Wavenumber in the surrounding medium k_particle Wavenumber of the particle

TmatrixPm methods:
getInternal Get the internal T-matrix

This class is based on tmatrix_pm.m from ottv1.

4.2.5 TmatrixSmarties

class ott.**TmatrixSmarties**(*a*, *c*, *varargin*)

Constructs a T-matrix using SMARTIES. Inherits from *ott.Tmatrix*

SMARTIES is a method for calculating T-matrices for spheroids.

This class requires the SMARTIES toolbox has been loaded onto the path. Details about the toolbox can be found in:

> Somerville, Auguié, Le Ru. JQSRT, Volume 174, May 2016, Pages 39-55. https://doi.org/10.1016/j.jqsrt.2016.01.005

See also TmatrixSmarties

4.2.6 TmatrixDda

class ott.**TmatrixDda**(*xyz*, *varargin*)

Constructs a T-matrix using discrete dipole approximation. Inherits from *ott.Tmatrix*.

To construct a T-matrix with DDA, either the simple interface or the class constructor can be used. Using the simple interface, the following should produce something similar to `TmatrixMie`:

```
Tmatrix = ott.TmatrixDda.simple('sphere', 0.1, 'index_relative', 1.2);
```

The DDA method requires a lot of memory to calculate the T-matrix. Most small desktop computers will be unable to calculate T-matrices for large particles (i.e., particles larger than a couple of wavelengths in diameter using 20 dipoles per wavelength). For these particles, consider using Geometric Optics or Finite Difference Time Domain method.

See also TmatrixDda, simple.

TmatrixDda(*xyz*, *varargin*)

Calculates T-matrix using discrete dipole approximation.

Usage

TmatrixDda(xyz, ...) calculates the T-matrix for the particle described by voxels xyz. xyz is a 3xN matrix of coordinates for each voxel.

The method supports homogenous and inhomogenous particles. For homogeneous particles, specify the material as a scalar, 3x1 vector or 3x3 polarizability matrix. For inhomogeneous particles use a N, 3xN or 3x3N vector/matrix.

Optional named parameters

- Nmax [r,c] Size of the T-matrix to generate. Default: `ott.utils.ka2nmax(max_radius*k_medium)`
- k_medium (numeric) – Wavenumber in medium
- wavelength_medium (numeric) – Wavelength in medium
- index_medium (numeric) – Refractive index in medium. Default: `k_medium = 2*pi`
- k_particle (numeric) – Wavenumber in particle
- wavelength_particle (numeric) – Wavelength in particle
- index_particle (numeric) – Refractive index in particle. Default: `k_particle = 2*pi*index_relative`
- polarizability (enum|numeric) – Polarizability or method name to use to calculate from relative refractive index. Default: 'LDR'. Supported methods: 'LDR', 'FCD', 'CM'.
- index_relative (numeric) – Relative refractive index. Default: 1.0
- wavelength0 (numeric) – Wavelength in vacuum. Default: 1.0
- spacing (numeric) – spacing for estimating Nmax and calculating the polarizability. Only required when polarizability is non-numeric. Default: `[]`
- z_mirror_symmetry (logical) – If z-mirror symmetry should be used. All voxels less than 0 are ignored. Default: `false`.
- z_rotational_symmetry (numeric) – z-rotational symmetry. Degree of rotational symmetry. Objects with no rotational symetry should set this to 1. Default: 1.

- low_memory (logical) – If true, the DDA implementation favours additional calculations over additional memory use allowing simualtion of larger particles. Default: false. Only applicable with z_mirror_symmetry and z_rotational_symmetry.
- modes (numeric) – Specifies the modes to include in the calculated T-matrix. Can either be a Nx2 matrix or a N element vector with the (n, m) modes or combined index modes respectively. Default: `[]`.
- use_nearfield (logical) – If true, uses near-field point matching. Default: *false*.
- use_iterative (logical) – If true, uses an iterative solver. Default: `false`.
- verbose (logical) – Display additional information. Doesn't affect the display of the progress callback. Default: false.

static simple(*shape*, *varargin*)

Construct a T-matrix using DDA for simple shapes.

Usage

simple(shape, ...) constructs a new simple T-matrix for the given *ott.shapes.Shape* object.

simple(name, parameters, ...) constructs a new T-matrix for the shape described by the name and parameters. For supported shape names, see `ott.shapes.Shape.simple`.

Optional named arguments:

- spacing (numeric) – Spacing between dipoles. Default: `wavelength_particle/20`

For other named parameters, see `TmatrixDda()`.

4.3 *shapes* Package

This section provides an overview of the shapes currently in the toolbox.

4.3.1 Base classes

class ott.shapes.**Shape**

Shape abstract class for optical tweezers toolbox shapes

Properties

maxRadius maximum distance from shape origin volume volume of shape position Location of shape `[x, y, z]`

Methods (abstract):

inside(shape, ...) determine if spherical point is inside shape

Methods:

writeWavefrontObj(shape, ...) write shape to Wavefront OBJ file
only implemented if shape supports this action.

insideXyz(shape, ...) determine if Cartesian point is inside shape
requires inside(shape, ...) to be implemented.

simple(...) simplified constructor for shape-like objects.

See also simple, ott.shapes.Cube, ott.shapes.TriangularMesh.

class ott.shapes.**AxisymShape**

AxisymShape abstract class for axisymetric particles

Methods

- boundarypoints calculate boudary points for surface integral

Abstract methods

- radii Calculates the particle radii for angular coordinates
- normals Calculates the particle normals for angular coorindates
- axialSymmetry Returns x, y, z rotational symmetry (0 for infinite)

class ott.shapes.**StarShape**

StarShape abstract class for star shaped particles

Abstract methods:
radii Calculates the particle radii for angular coordinates normals Calculates the particle normals for angular coorindates axialSymmetry Returns x, y, z rotational symmetry (0 for infinite) mirrorSymmetry Returns x, y, z mirror symmetry

4.3.2 Geometric shapes

class ott.shapes.**Cube**(*width*)

Cube a simple cube shape

properties:
width % Width of the cube

class ott.shapes.**RectangularPrism**(*x*, *y*, *z*)

Cube a simple cube shape

properties:
x Size of prism in x direction y Size of prism in y direction z Size of prism in z direction

See also RectangularPrism and ott.shapes.Cube.

class ott.shapes.**Cylinder**(*radius*, *height*)

Cylinder a simple cylinder shape

properties:
radius % Radius of the cylinder height % Height of the cylinder

class ott.shapes.**Ellipsoid**(*a*, *b*, *c*)

Ellipsoid a simple ellipsoid shape

properties:
a % x-axis scaling b % y-axis scaling c % z-axis scaling

class ott.shapes.**Sphere**(*radius*, *position*)

Sphere a simple sphere shape

properties:

radius % Radius of the sphere

class ott.shapes.**Superellipsoid**(*a*, *b*, *c*, *ew*, *ns*)

Superellipsoid a simple superellipsoid shape

properties:

a % x-axis scaling b % y-axis scaling c % z-axis scaling ew % East-West smoothness (ew = 1 for ellipsoid) ns % North-South smoothness (sw = 1 for ellipsoid)

4.3.3 Sets of shapes

These classes can be used to create shapes by combining simple geometric shapes or other shape objects. For instance, the union class can be used to create a union of two spheres:

```
shape1 = ott.shapes.Sphere(1.0, [0, 0, -2]);
shape2 = ott.shapes.Sphere(1.0, [0, 0,  2]);
union = ott.shapes.Union([shape1, shape2]);
```

class ott.shapes.**Union**(*shapes*)

Represents union between two shapes. Inherits from `ott.shapes.Shape`.

A point is considered to be inside the union if the point is inside any of the shapes in the union.

Methods

inside – Determine if point is inside any contained shape.

Properties

shapes – Shapes contained in this union volume – Estimate of shape volume from sum of shapes in set maxRadius – Estimate of shape maximum radius

See also Union

Union(*shapes*)

Construct a new union of shapes.

Usage

shape = Union([shape1, shape2, ...])

Todo: We will probably add other sets in future including differences or exclusions.

4.3.4 Procedural shapes

class ott.shapes.**AxisymLerp**(*rho*, *z*)

AxisymLerp a axisymmetric particle with lerping between points Inherits from ott.shapes.StarShape and ott.shapes.AxisymShape.

properties:

rho % Radial position of defining points (cylindrical coords) z % z position of defining points (cylindrical coords)

See also AxisymLerp

class ott.shapes.**TriangularMesh**(*verts*, *faces*)

TriangularMesh base class for triangular mesh objects (such as file loaders)

Properties (read-only):
verts 3xN matrix of vertex locations faces 3xN matrix of vertex indices describing faces

Faces vertices should be ordered so normals face outwards for volume and inside functions to work correctly.

4.3.5 File loaders

class ott.shapes.**StlLoader**(*filename*)

StlLoader load a shape from a STL file

Properties:
filename name of the file this object loaded verts (TriangularMesh) 3xN matrix of vertex locations faces (TriangularMesh) 3xN matrix of vertex indices describing faces maxRadius (Shape) maximum distance from shape origin volume (Shape) volume of shape

Inherited methods:
writeWavefrontObj(shape, ...) write shape to Wavefront OBJ file. insideXyz(shape, ...) determine if Cartesian point is inside shape. voxels(shape, ...) xyz coordinates for voxels inside the shape. surf(shape, ...) shape surface representation.

See also StlLoader, ott.shapes.TriangularMesh, ott.shapes.WavefrontObj.

This class uses a 3rd party STL file reader:
https://au.mathworks.com/matlabcentral/fileexchange/22409-stl-file-reader

See tplicenses/stl_EricJohnson.txt for information about licensing.

class ott.shapes.**WavefrontObj**(*filename*)

WavefrontObj load a shape from a Wavefront OBJ file

The file format is described on the Wikipedia page.

4.4 *ui* Package

Contents

4.5 *utils* Package

This package contains functions used by other methods in the toolbox. Most of these functions are from version 1 of the toolbox with some minor modifications and bug fixes.

Warning: These functions are likely to move in future releases and are not very well documented.

4.5.1 Special functions

`ott.utils.`**`sbesselh1`**`(`*n*, *kr*, *varargin*`)`

SBESSELH1 spherical hankel function hn(kr) of the first kind, hn(kr) = sqrt(pi/2kr) Hn+0.5(kr).

hn = SBESSELH1(n,z) calculates spherical hankel function of first kind.

[hn,dzhn] = SBESSELH1(n,z) additionally, calculates the derivatives of the appropriate Ricatti-Bessel function divided by z.

See also besselj and bessely.

`ott.utils.`**`sbesselh2`**`(`*n*, *kr*, *varargin*`)`

SBESSELH2 spherical hankel function hn(kr) of the second kind hn(kr) = sqrt(pi/2kr) Hn+0.5(kr)

hn = SBESSELH2(n,z) calculates the hankel function of the second kind.

[hn,dzhn] = SBESSELH2(n,z) additionally, calculates the derivative of the appropriate Ricatti-Bessel function divided by z.

See also besselj and bessely.

`ott.utils.`**`sbesselh`**`(`*n*, *htype*, *kr*`)`

SBESSELH spherical hankel function hn(kr) of the first or second kind hn(kr) = sqrt(pi/2kr) Hn+0.5(kr)

hn = SBESSELH(n,htype,z) computes the spherical hankel function of degree, n, of kind, htype, and argument, z.

[hn,dzhn] = SBESSELH(n,htype,z) additionally, calculates the derivative of the appropriate Ricatti-Bessel function divided by z.

See also besselj and bessely.

`ott.utils.`**`sbesselj`**`(`*n*, *kr*`)`

SBESSELJ spherical bessel function jn(kr) jn(kr) = sqrt(pi/2kr) Jn+0.5(kr)

jn = SBESSEL(n,z) calculates the spherical bessel function.

[jn,dzjn] = sbessel(n,z) additionally, calculates the derivative of the appropriate Ricatti-Bessel function divided by z.

See also besselj.

`ott.utils.`**`spharm`**`(`*n*, *m*, *theta*, *phi*`)`

SPHARM scalar spherical harmonics and angular partial derivatives.

Y = SPHARM(n,m,theta,phi) calculates scalar spherical harmonics.

[Y,Ytheta,Yphi] = SPHARM(n,m,theta,phi) additionally, calculates the angular partial derivatives dY/dtheta and 1/sin(theta)*dY/dphi.

SPHARM(n,theta,phi) as above but for all m.

Scalar n for the moment.

If scalar m is used Y is a vector of length(theta,phi) and is completely compatible with previous versions of the toolbox. If vector m is present the output will be a matrix with rows of length(theta,phi) for m columns.

"Out of range" n and m result in return of Y = 0

ott.utils.**vsh**(*n*, *m*, *theta*, *phi*)

VSH calculate vector spherical harmonics

[B,C,P] = VSH(n,m,theta,phi) calculates vector spherical harmonics for the locations theta, phi. Vector m allowed. Scalar n for the moment.

[B,C,P] = VSH(n,theta,phi) outputs for all possible m.

If scalar m: B,C,P are arrays of size length(theta,phi) x 3 If vector m: B,C,P are arrays of size length((theta,phi),m) x 3 theta and phi can be vectors (of equal length) or scalar.

The three components of each vector are [r,theta,phi]

"Out of range" n and m result in return of [0 0 0]

ott.utils.**vswf**(*n*, *m*, *kr*, *theta*, *phi*, *htype*)

VSWF vector spherical wavefunctions: M_k, N_k.

[M1,N1,M2,N2,M3,N3] = VSWF(n,m,kr,theta,phi) calculates the outgoing M1,N1, incomming M2,N2 and regular M3,N3 VSWF. kr, theta, phi are vectors of equal length, or scalar.

[M,N] = VSWF(n,m,kr,theta,phi,type) calculates only the requested VSWF, where type is

1 -> outgoing solution - h(1) 2 -> incoming solution - h(2) 3 -> regular solution - j (ie RgM, RgN)

VSWF(n, kr, theta, phi) if m is omitted, will calculate for all m.

M,N are arrays of size length(vector_input,m) x 3

The three components of each vector are [r,theta,phi].

"Out of range" n and m result in return of [0 0 0]

ott.utils.**vswfcart**(*n*, *m*, *kr*, *theta*, *phi*, *htype*)

VSWFCART vector spherical harmonics spherical coordinate input, cartesian output.

[M1,N1,M2,N2,M3,N3] = VSWFCART(n,m,kr,theta,phi) calculates the outgoing M1,N1, incomming M2,N2 and regular M3,N3 VSWF. kr, theta, phi are vectors of equal length, or scalar.

[M,N] = VSWFCART(n,m,kr,theta,phi,type) calculates only the requested VSWF, where type is

1 -> outgoing solution - h(1) 2 -> incoming solution - h(2) 3 -> regular solution - j (ie RgM, RgN)

Scalar n,m for the moment. M,N are arrays of size length(vector_input) x 3

The three components of each input vector are [kr,theta,phi] The three components of each output vector are [x,y,z]

"Out of range" n and m result in return of [0 0 0]

At the coordinate origin (kr == 0) we use only theta/phi.

4.5.2 Coordinate transformations

ott.utils.**rtp2xyz**(*r*, *theta*, *phi*)

RTP2XYZ coordinate transformation from spherical to cartesian
r radial distance [0, Inf) theta polar angle, measured from +z axis [0, pi] phi azimuthal angle, measured from +x towards +y axes [0, 2*pi)

[x,y,z] = RTP2XYZ(r,theta,phi) takes vectors or scalars, outputs the spherical coordinates as vectors/scalars of the same size.

[x,y,z] = RTP2XYZ(r) same as above but with the coordinate packed into the vector/matrix r = [r theta phi].

x = RTP2XYZ(...) same as above with the result packed into the vector/matrix x = [x y z].

ott.utils.**rtpv2xyzv**(*rv*, *thetav*, *phiv*, *r*, *theta*, *phi*)

RTPV2XYZV spherical to cartiesn vector field conversion

[xv,yv,zv,x,y,z] = RTPV2XYZV(rv,thetav,phiv,r,theta,phi)

[vec_cart,pos_cart] = rtpv2xyzv(vec,pos)

Inputs must be column vectors or Nx3 matrices.

See also rtp2xyz and xyzv2rtpv.

ott.utils.**xyz2rtp**(*x*, *y*, *z*)

XYZ2RTP coordinate transformation from cartesian to spherical
r radial distance [0, Inf) theta polar angle, measured from +z axis [0, pi] phi azimuthal angle, measured from +x towards +y axes [0, 2*pi)

[r,theta,phi] = XYZ2RTP(x,y,z) takes vectors or scalars outputs the spherical coordinates as vectors/scalars of the same size.

[r,theta,phi] = XYZ2RTP(x) same as above but with the coordinate packed into the vector/matrix x = [x y z].

r = XYZ2RTP(...) same as above with the result packed into the vector/matrix r = [r theta phi].

ott.utils.**xyzv2rtpv**(*xv*, *yv*, *zv*, *x*, *y*, *z*)

XYZV2RTPV cartiesian to spherical vector field conversion

[rv,thetav,phiv,r,theta,phi] = XYZV2RTPV(xv,yv,zv,x,y,z)

[vec_sph,pos_sph] = XYZV2RTPV(vec_cart,pos_cart)

See also rtpv2xyzv and xyz2rtp.

4.5.3 Translations and rotations

ott.utils.**translate_z**(*nmax*, *z*, *varargin*)

Calculates translation matrices for translation of VSWFs along z axis.

Usage
[A,B] = translate_z(nmax,z) calculates translation matrices. The matrices are use as:

$$M' = AM + BN$$
$$N' = BM + AN$$

[A,B,C] = ott.utils.translate_z(nmax,z) additionally, calculates C, the scalar SWF translation coefficients in 3d packed form.

Parameters

- nmax (int) – Determines the number of multipole terms to include in the translation matrices (multipole order). Can be a single integer or two integers for the `[row, column]` nmax. If the row, column indices don't match, A and B will not be square.
- z (numeric) – Translation distance.

A and B are sparse matrices, since only m' = m VSWFs couple

If z is a vector/matrix only A's and B's will be outputted. A and B will be cells of matrices the length of the number of elements in z. To save time only use unique values of z.

Time *may* be saved by taking the conjugate transpose instead of calculating translations in the positive or negative direction.

Optional named parameters

- 'type' (enum) – Type of translation matrix to generate.
- 'method' (enum) – Method to calculate translation matrix.

Translation matrix types

- 'sbesselj' regular to regular. Default. Used for most particle or beam translations.
- 'sbesselh1' outgoing to regular. Can be useful for doing multiple scattering calculations.
- 'sbesselh2' incoming to regular
- 'sbesselh1farfield' outgoing to regular far-field limit
- 'sbesselh2farfield' incoming to regular far-field limit

Methods

- 'gumerov' – Use Gumerov method (default, recommended)
- 'videen' – Use Videen method. (not recommended for large translations of the beam, unstable)

Example usage

The following example calculates the A and B translation matrices and applies them to a Gaussian beam. A procedure similar to this is done when calling `Bsc.translateZ` with a distance:

```
beam = ott.BscPmGauss('NA', 0.9, 'index_medium', 1.0, ...
    'polarisation', [1, 0], 'wavelength0', 1064e-9);

z = 1.0e-6 ./ beam.wavelength;
[A, B] = translate_z([beam.Nmax, beam.Nmax], z);

% Apply translation matrices
new_beam = beam.translate(A, B);
```

`ott.utils.`**`rotx`**(*angle_deg*, *varargin*)

Create a 3x3 rotation matrix for rotation about x axis

R = rotx(angle_deg) calculate the rotation matrix for rotations from the +z towards +y axis.

R = rotx([a1, a2, a3, ...]) returns a 3xN matrix of rotation matrices for each angle in the input.

Optional named arguments:

usecell bool True to output as cell array instead of 3xN matrix.
Default: false. The cell array has the same shape as angle_deg.

Replacement/extension to Matlab rotx function provided in the Phased Array System Toolbox.

ott.utils.**roty**(*angle_deg*, *varargin*)

Create a 3x3 rotation matrix for rotation about y axis

R = roty(angle_deg) calculate the rotation matrix for rotations from the +z towards +x axis.

R = roty([a1, a2, a3, ...]) returns a 3xN matrix of rotation matrices for each angle in the input.

Optional named arguments:

usecell bool True to output as cell array instead of 3xN matrix.
Default: false. The cell array has the same shape as angle_deg.

Replacement/extension to Matlab roty function provided in the Phased Array System Toolbox.

ott.utils.**rotz**(*angle_deg*, *varargin*)

Create a 3x3 rotation matrix for rotation about z axis

R = rotz(angle_deg) calculate the rotation matrix for rotations from the +x towards +y axis.

R = rotz([a1, a2, a3, ...]) returns a 3xN matrix of rotation matrices for each angle in the input.

Optional named arguments:

usecell bool True to output as cell array instead of 3xN matrix.
Default: false. The cell array has the same shape as angle_deg.

Replacement/extension to Matlab rotz function provided in the Phased Array System Toolbox.

ott.utils.**rotation_matrix**(*rot_axis*, *rot_angle*)

ROTATION_MATRIX calculates rotation matrix using Euler-Rodrigues formula.

R = rotation_matrix(axis, angle) calculates the rotation about axis by angle (in radians).

R = rotation_matrix(axis_angle) calculates the rotation about vector axis_angle, the angle is specified as the length of the vector (in radians).

ott.utils.**wigner_rotation_matrix**(*nmax*, *R*)

WIGNER_ROTATION_MATRIX rotation matrix for rotation of spherical harmonics or T-matrices.

D = WIGNER_ROTATION_MATRIX(nmax,R) calculates the rotation matrix for the VSH given a 3x3 coordinate rotation matrix R. Usage: a' = D a.

This method from Choi et al., J. Chem. Phys. 111: 8825-8831 (1999) Note change in notation - here, use x' = Rx (x is row vector), a' = Da (a is column vector) etc.

4.5.4 Helper functions

ott.utils.**matchsize**(*varargin*)

Checks that all vector inputs have the same number of rows.

Usage

[A,B,...] = matchsize(A,B,...) checks inputs have same number of rows, and expands single-row inputs by repetition to match the input row number.

Parameters

- A,B,... (numeric) – Numeric arrays whose number of rows are to be matched.

Example

The following example shows has two inputs, a scalar and a row vector. The scalar is duplicated to match the length of the row vector:

```
A = 5;
B = [1; 2; 3];

[A,B] = matchsize(A, B);
disp(A)  % -> [5; 5; 5]
disp(B)  % -> [1; 2; 3]
```

ott.utils.**threewide**(*a*)

THREEWIDE creates colum vector with input repeated in 3 columns
the function can take a column of row vector input, the output will be a matrix with three columns.

You might find this useful for multiplying a vector of scalars with a column vector of 3-vectors.

ott.utils.**iseven**(*input*)

ISEVEN determines if an integer is even Outputs a matrix of the same size as input with 1 for even and 0 for odd entries.

Warning: Plays up if the the integer is of the order 10^16

ott.utils.**isodd**(*input*)

ISODD determines if an integer is odd Outputs a matrix the same size as input with 1 for odd and 0 for even entries.

Warning: Plays up if the the integer is of the order 10^16

ott.utils.**rotate_3x3tensor**(*ualpha*, *varargin*)

Apply a set of rotations to a 3x3 tensor

alpha = rotate_3x3tensor(ualpha, R, ...) applies the operation alpha = R*ualpha*inv(R) if R is a 3x3N matrix of rotation matrices.

alpha = rotate_3x3tensor(ualpha, 'direction', dir, ...) computes the appropriate rotation rotation matrix to rotate from the z-axis to the 3xN matrix of directions. This is useful for uniaxial materials. 'direction' is the [x; y; z] Cartesian coordinate. 'sphdirection' is the [phi; theta] Spherical coordinate.

Optional named parameters:

'inverse' bool When true, returns the inverse polarisability.
Default: false.

ott.utils.**ka2nmax**(*ka*)

Finds a reasonable Nmax to truncate at for given size parameter

Usage
Nmax = ka2nmax(ka) calculates reasonable maximum order, Nmax, to truncate beam beam coefficients/T-matrix at for a given size parameter.

Returns $Nmax = |ka| + 3(|ka|)^(1/3)$

ott.utils.**nmax2ka**(*Nmax*)

NMAX2KA finds size parameter ka corresponding to Nmax

ka = NMAX2KA(Nmax) finds size parameter for maximum order, Nmax, which spherical expansions are truncated.

Truncation order is given by Nmax = ka + 3 (ka)^(1/3)

4.5.5 Geometry functions

ott.utils.**angulargrid**(*ntheta*, *nphi*, *behaviour*)

ANGULARGRID makes a angular grid of points over a sphere

[theta,phi] = ANGULARGRID(N) generates two column N^2-by-1 matrices with theta (polar) and phi (azimuthal) angle pairs for N discrete evenly spaced polar and azimuthal angles.

[theta,phi] = ANGULARGRID(ntheta, nphi) specifies the number of evenly spaced points to use in the theta and phi direction.

[theta,phi] = ANGULARGRID(..., behaviour) uses behaviour to control the output type:

0 | column vectors of all points 1 | vectors of all theta and phi values 2 | ntheta x nphi matrix of all points

Note that the output data values are the same for behaviours 0 and 2; they're just arranged differently. To convert from one format to another: 2 -> 0: theta = theta(:); phi = phi(:); 0 -> 2: theta = reshape(theta,ntheta,nphi);

phi = reshape(phi,ntheta,nphi);

ott.utils.**perpcomponent**(*A*, *n*)

PERPCOMPONENT finds perpendicular (and optionally) parallel components of a vector relative to a reference vector.

perp_component = PERPCOMPONENT(A,n) calculates perpendicular component of row vector A or Nx3 matrix A of vectors. n is the reference vector.

[perp_component,parallel_component] = PERPCOMPONENT(A,n) calculates perpendicular and parallel components.

ott.utils.**inpolyhedron**(*varargin*)

INPOLYHEDRON Tests if points are inside a 3D triangulated (faces/vertices) surface

IN = INPOLYHEDRON(FV,QPTS) tests if the query points (QPTS) are inside the patch/surface/polyhedron defined by FV (a structure with fields 'vertices' and 'faces'). QPTS is an N-by-3 set of XYZ coordinates. IN is an N-by-1 logical vector which will be TRUE for each query point inside the surface. By convention, surface normals point OUT from the object (see FLIPNORMALS option below if to reverse this convention).

INPOLYHEDRON(FACES,VERTICES,...) takes faces/vertices separately, rather than in an FV structure.

IN = INPOLYHEDRON(..., X, Y, Z) voxelises a mask of 3D gridded query points rather than an N-by-3 array of points. X, Y, and Z coordinates of the grid supplied in XVEC, YVEC, and ZVEC respectively. IN will return as a 3D logical volume with SIZE(IN) = [LENGTH(YVEC) LENGTH(XVEC) LENGTH(ZVEC)], equivalent to syntax used by MESHGRID. INPOLYHEDRON handles this input faster and with a lower memory footprint than using MESHGRID to make full X, Y, Z query points matrices.

INPOLYHEDRON(...,'PropertyName',VALUE,'PropertyName',VALUE,...) tests query points using the following optional property values:

TOL - Tolerance on the tests for "inside" the surface. You can think of tol as the distance a point may possibly lie above/below the surface, and still be perceived as on the surface. Due to numerical rounding nothing can ever be done exactly here. Defaults to ZERO. Note that in the current implementation TOL only affects points lying above/below a surface triangle (in the Z-direction). Points coincident with a vertex in the XY plane are considered INside the surface. More formal rules can be implemented with input/feedback from users.

GRIDSIZE - Internally, INPOLYHEDRON uses a divide-and-conquer algorithm to split all faces into a chessboard-like grid of GRIDSIZE-by-GRIDSIZE regions. Performance will be a tradeoff between a small GRIDSIZE (few iterations, more data per iteration) and a large GRIDSIZE (many iterations of small data calculations). The sweet-spot has been experimentally determined (on a win64 system) to be correlated with the number of faces/vertices. You can overwrite this automatically computed choice by specifying a GRIDSIZE parameter.

FACENORMALS - By default, the normals to the FACE triangles are computed as the cross-product of the first two triangle edges. You may optionally specify face normals here if they have been pre-computed.

FLIPNORMALS - (Defaults FALSE). To match a wider convention, triangle face normals are presumed to point OUT from the object's surface. If your surface normals are defined pointing IN, then you should set the FLIPNORMALS option to TRUE to use the reverse of this convention.

Example:

tmpvol = zeros(20,20,20); % Empty voxel volume tmpvol(5:15,8:12,8:12) = 1; % Turn some voxels on tmpvol(8:12,5:15,8:12) = 1; tmpvol(8:12,8:12,5:15) = 1; fv = isosurface(tmpvol, 0.99); % Create the patch object fv.faces = fliplr(fv.faces); % Ensure normals point OUT % Test SCATTERED query points pts = rand(200,3)*12 + 4; % Make some query points in = inpolyhedron(fv, pts); % Test which are inside the patch figure, hold on, view(3) % Display the result patch(fv,'FaceColor','g','FaceAlpha',0.2) plot3(pts(in,1),pts(in,2),pts(in,3),'bo','MarkerFaceColor','b') plot3(pts(~in,1),pts(~in,2),pts(~in,3),'ro'), axis image % Test STRUCTURED GRID of query points gridLocs = 3:2.1:19; [x,y,z] = meshgrid(gridLocs,gridLocs,gridLocs); in = inpolyhedron(fv, gridLocs,gridLocs,gridLocs); figure, hold on, view(3) % Display the result patch(fv,'FaceColor','g','FaceAlpha',0.2) plot3(x(in), y(in), z(in),'bo','MarkerFaceColor','b') plot3(x(~in),y(~in),z(~in),'ro'), axis image

See also: UNIFYMESHNORMALS (on the file exchange)

This is a third party function, not part of OTTv2, see tplicenses/stl_Sven.txt for information about licensing.

4.5.6 Unclassified

Todo: These functions should be moved to other categories

ott.utils.**paraxial_transformation_matrix**(*paraxial_order*, *basis_in*, *basis_out*, *normal_mode*)

PARAXIAL_TRANSFORMATION_MATRIX produces paraxial beam mode conversion in a particular order.

[modeweights, col_modes, row_modes] = ...
PARAXIAL_TRANSFORMATION_MATRIX(degree, basis_in, basis_out) or

[modeweights, col_modes, row_modes] = ...
PARAXIAL_TRANSFORMATION_MATRIX(degree, basis_in, basis_out, normal_mode)

inputs:

degree : paraxial degree of modes e.g. gaussian is 0. basis_in : 0 vortex LG, 1 vortex HG, [2,xi] vortex IG. basis_out : 0 LG, 1 HG, [2,xi] IG. normal_mode : 0 is default vortex->non-vortex. (because of toolbox modes)

1 makes the conversion non-vortex->non-vortex.

outputs:

modeweights
[weights of the conversion basis_in->basis_out. Format: each] row is the corresponding mode of the output

basis, each column for the input basis, such that conj. tranpose(basis_1 –> basis_2) = basis_2 –> basis_1. holds for (vortex->non-vortex)' == non-vortex->vortex.

col_modes
[outputs the LG, HG or IG indices corresponding to each] COLUMN of the matrix. [p,l], [m,n], [o,m,p].

row_modes
[outputs the LG, HG or IG indices corresponding to each ROW] of the matrix. [p,l], [m,n], [o,m,p].

ott.utils.**paraxial_beam_waist**(*paraxial_order*)

ott.utils.**lgmode**(*p*, *l*, *r*, *phi*, *z*, *theta*)

LGMODE calculates LG mode amplitude at z = 0

A = LGMODE(p,l,r,phi) calculates the LG mode amplitude for mode [p,l] at locations given in polar coordinates [r, phi]. r is in units of the beam width; r and phi can be matrices of equal size.

A = LGMODE(p,l,r,phi,z) computes the modes with z as well.

A = LGMODE(p,l,r,phi,z,theta) scales the beam waist according to the beam convergence angle theta (in degrees): w0=1/tan(theta).

ott.utils.**legendrerow**(*n*, *theta*)

LEGENDREROW gives the spherical coordinate recursion in m

pnm = LEGENDREROW(n, theta) gives the spherical recursion for a given n, theta.

This provides approximately no benefit over the MATLAB implimentation. It *may* provide a benefit in Octave. Inspiration from [Holmes and Featherstone, 2002] and [Jekeli et al., 2007].

ott.utils.**laguerre**(*p*, *l*, *X*)

LAGUERRE associated Laguerre function

L = LAGUERRE(p,l,X) evaluate associated Laguerre function. p and l must be integer scalars greater than zero

ott.utils.**incecoefficients**(*p*, *xi*)

ott.utils.**hgmode**(*m*, *n*, *x*, *y*, *z*, *theta*)

ott.utils.**hermite**(*n*, *X*)

ott.utils.**emField**(*krtp*, *type*, *nm*, *ab*, *varargin*)

EMFIELD calculates field from the vector spherical wave functions

[E, H, data] = emField(krtp, type, nm, ab, ...) calculates the E and H field for unit-less spherical coordinates krtp. krtp is a Nx3 matrix of [radial, polar, azimuthal] coordinates. type must be 'incoming', 'regular', or 'outgoing'.

Optional named parameters:

'saveData' bool saves data that can be used for repeated
calculations of the fields at these locations (default: nargout==3).

'data' data data to use from previous calculation 'calcE' bool calculate E field (default: true) 'calcH' bool calculate H field (default: true)

If internal fields are calculated only the theta and phi components of E are continuous at the boundary. Conversely, only the kr component of D is continuous at the boundary.

ott.utils.**combined_index**(*in1*, *in2*)

COMBINED_INDEX translates between (n,m) and combined index Mode indices and combined index are related by: ci = n * (n+1) + m.

[n,m] = COMBINED_INDEX(ci) calculates (n,m) from the combined index.

ci = COMBINED_INDEX(n,m) calculates the combined index from mode indices.

length = COMBINED_INDEX(Nmax, Nmax) calculates length of the beam vectors.

Nmax = COMBINED_INDEX(length) calculates Nmax from length of beam vectors.

4.5.7 Polarizability calculation

ott.utils.polarizability.**FCD**(*spacing*, *index*, *varargin*)

Filtered coupled dipole polarizability

Usage

alpha = FCD(spacing, index) Calculates a Nx1 element vector containing the isotropic polarizabilities for N dipoles.

Parameters

- spacing (numeric scalar) – lattice spacing parameter
- index (Nx1 numeric) – Relative refractive indices for N dipoles.

Optional named arguments

- k0 (numeric) – Wavenumber to scale spacing by. Default: `2*pi`.

ott.utils.polarizability.**LDR**(*spacing*, *index*, *varargin*)

Lattice dispersion relation polarizablity

Polarizability calculation based on

> Draine & Goodman, Beyond Clausius-Mossoti: wave propagation on a polarizable point lattice and the discrete dipole approximation, The Astrophysical Journal, 405:685-697, 1993 March 10

Usage

alpha = LDR(spacing, index, ...) Calculates a Nx1 element vector containing the isotropic polarisabilities for N dipoles.

alpha = LDR(spacing, index, kvec, E0, ...) As above but specifies the polarisability information for use with plane wave illumination.

Parameters

- spacing (numeric scalar) – lattice spacing parameter
- index (Nx1 numeric) – Relative refractive indices for N dipoles.
- kvec (1x3 numeric) – Wave vector [kx, ky, kz]
- E0 (1x3 numeric) – E-field polarisation [Ex, Ey, Ez]

Optional named arguments

- k0 (numeric) – Wavenumber to scale spacing by. Default: `2*pi`.

ott.utils.polarizability.**CM**(*spacing*, *index*)

Clausius-Mossoti Polarizability

Usage

alpha = CM(spacing, index) Calculates a Nx1 element vector containing the isotropic polarisabilities for N dipoles.

Parameters

- spacing (numeric scalar) – lattice spacing parameter
- index (Nx1 numeric) – Relative refractive indices for N dipoles.

4.6 Other functions

These functions operate on beams, particles or data generated by other methods.

There is also a `ott.warning` and `ott.change_warning` functions, for further information about these functions, see the documentation in their respective files.

Contents

4.6.1 forcetorque

`ott.`**`forcetorque`**(*ibeam*, *sbeam*, *varargin*)

FORCETORQUE calculate force/torque/spin in a 3D orthogonal space If the beam shape coefficients are in the original coordinates, this outputs the force, torque and spin in 3D carteisan coordinates.

Units are beam power. Force results should be multipled by n/c and torque results multiplied by 1/omega, assuiming the beam coefficients already have the correct units for power.

[fxyz,txyz,sxyz] = FORCETORQUE(ibeam, sbeam) calculates the force, torque and spin using the incident beam, ibeam, and the scattered beam, sbeam.

Output is stored in [3, 1] column vectors. If torque or spin are omitted, only force or force/torque are calculated.

FORCETORQUE(ibeam, T, 'position', position) first applies a translation to the beam. position can be a 3xN array, resulting in multiple force/torque calculations for each position.

FORCETORQUE(ibeam, T, 'rotation', rotation) effectively applies a rotation to the particle by first applying the rotation to the beam, scattering the beam by the T-matrix and applying the inverse rotation to the scattered beam. rotation can be a 3x3N array, resulting in multiple calculations.

If both position and rotation are present, the translation is applied first, followed by the rotation. If both position and rotation are arrays, they must have the same number of locations (N) or a single location (N=1).

ibeam can contain multiple beams. If multiple beams are present, the outputs are [3, nlocations, nbeams] arrays unless the coherent argument is set to true, in which case the beams are added after translation.

[fx,fy,fz,tx,ty,tz,sx,sy,sz] = FORCETORQUE(...) unpacks the force/torque/spin into separate output arguments.

This uses mathematical result of Farsund et al., 1996, in the form of Chricton and Marsden, 2000, and our standard T-matrix notation S.T. E_{inc}=sum_{nm}(aM+bN);

This file is part of the optical tweezers toolbox. See LICENSE.md for information about using/distributing this file.

4.6.2 axial_equilibrium

ott.**axial_equilibrium**(*tmatrix*, *beam*, *z*)

AXIAL_EQUILIBRIUM find equilibrium position and stiffness along beam axis

[z,kz] = AXIAL_EQUILIBRIUM(T,beam) attempts to locate the equilibrium position for the T-matrix T in beam starting with an initial guess at z = 0.

[z,kz] = axial_equilibrium(..., initial_guess) specifies an initial guess.

This file is part of the optical tweezers toolbox. See LICENSE.md for information about using/distributing this file.

4.6.3 find_equilibrium

ott.**find_equilibrium**(*z*, *fz*)

FIND_EQUILIBRIUM estimates equilibrium positions from position-force data

zeq = find_equilibrium(z, fz) finds the axial equilibrium given two vectors z and fz with the position and force values respectively.

Based on the code in example_gaussian from the original toolbox.

See also ott.axial_equilibrium

TODO: Generalize the code to find multiple equilibriums. TODO: z need not be a vector of scalars, we could have a array of

> position vectors representing some path we want to find the equilibrium on. We could do a similar thing for fz.

This file is part of the optical tweezers toolbox. See LICENSE.md for information about using/distributing this file.

4.6.4 trap_stiffness

ott.**trap_stiffness**(*beam*, *T*, *varargin*)

TRAP_STIFFNESS calculate the force and torque trap stiffness

[kf, kt, K] = TRAP_STIFFNESS(beam, T) calculate the trap stiffness for the force, kf, and torque, kt for a particle, T, in a beam. K is the full 6x2N stiffness matrix, each column corresponds to a different direction. K can be multiplied by the drag tensor to calculate the fluid+optical trap stiffness.

TRAP_STIFFNESS(..., 'position', x0) and TRAP_STIFFNESS(, 'rotation', R) specify the particle position and rotation.

TRAP_STIFFNESS(..., 'direction', d) specifies the directions to calculate the trap stiffness in. Specify 'beam' for the XYZ direction for the current beam orientation. For specific directions, specify a 3xN matrix of vectors to calculate the force along and torque around. Default is 'beam'.

TRAP_STIFFNESS(..., 'method', m) specifies the method to use for calculating the trap stiffness. Supported methods [calculations direction/other]:

> 'cntr' use central differences [4/0] 'fwd' use forward differences [2/1]

TRAP_STIFFNESS(..., 'step', [dx dt]) specifies the position and torque calculation step sizes (in beam units and radians). Default: 1e-3/beam.k_medium and 1e-3.

This file is part of the optical tweezers toolbox. See LICENSE.md for information about using/distributing this file.

4.6.5 find_traps

ott.**find_traps**(*position*, *force*, *varargin*)

FIND_TRAPS attempt to find and characterise traps from position-force data

traps = find_traps(position, force, ...) attempts to find and characterise possible traps for the given position and force data. Position and force should be vectors with position and force along one axis.

The returned traps is an array of structures with information about the trap equilibrium position, trap depth and trap stiffness and trap range.

Optional named arguments:

keep_unstable bool keep unstable equilibriums (default: false) depth_threshold_e num percentage of max depth for trap acceptance

Use [] for no threshold. (default: 1e-2).

force_zero_tol_e num percentage zero tolerance for pre-filtering
forces. Use [] for no threshold. (default: 1e-3).

group_stable bool group stable traps separated by smaller
unstable traps together (useful for finding trap depth) (default: false)

plot bool plot the traps, useful for diagnostics.
(default: false)

See also ott.find_equilibrium ott.axial_equilibrium and ott.trap_stiffness.

This file is part of the optical tweezers toolbox. See LICENSE.md for information about using/distributing this file.

CHAPTER

FIVE

CONCEPTUAL NOTES

This section provides more detailed information about various concepts used in the toolbox. It is a response to questions we have received about the toolbox, its accuracy and various implementation decisions.

Contents

5.1 Spherical wave representation

The toolbox represents fields using a vector spherical wave function (VSWF) basis. This basis is infinite and, with infinite basis functions, it can be used to represent any field. However, in practice we are often forced to choose a finite number of basis functions to approximate our field. The accuracy of our approximation depends on how similar our field is to the basis functions. For example, in the VSWF basis we can exactly represent the quadrapole field (see Fig. 5.1 a) with only a single basis function (the dipole mode). Conversely, a plane wave would need an infinite number of basis functions to be represented exactly.

For optical scattering calculations, we often don't need to represent our field exactly everywhere: in most cases it is sufficient to represent the field exactly only around the scattering object. In a VSWF basis, we are able to accurately represent the fields in spherical region located at the centre of our coordinate system. The size of the region is determined by the number of VSWF spherical modes, i.e. N_{max}. By choosing an appropriate N_{max} we are able to represent plane waves and other non-localised waves in a finite region surrounding our particle, as shown in Fig. 5.1 b. This can create some difficulty if our particle cannot be circumscribed by such a sphere, as is the case for infinite slabs. For modelling scattering by infinite slabs, it would be better to use a plane wave basis.

For certain type of beams, such as focussed Gaussian beams, most of the information describing the beam passes through an aperture with a finite radius, as shown in Fig. 5.1 c. These beams can be represented accurately as long as N_{max} is large enough to surround this aperture. This is the condition used for automatic N_{max} selection in `ott.BscPmGauss`.

The accuracy of translated beam depends on the N_{max} in the translated region and the accuracy of the original beam around the new origin. For plane waves and other beams with infinite extent, this means that translations outside the original N_{max} region may not be accurate. For the case of plane waves, this can be circumvented by implementing translations as phase shifts. For Gaussian beams, as long as the original beam has a large enough N_{max} to accurately describe the beam, the beam can be translated to almost any location (within the accuracy of the translation method). This is illustrated in figure Fig. 5.2.

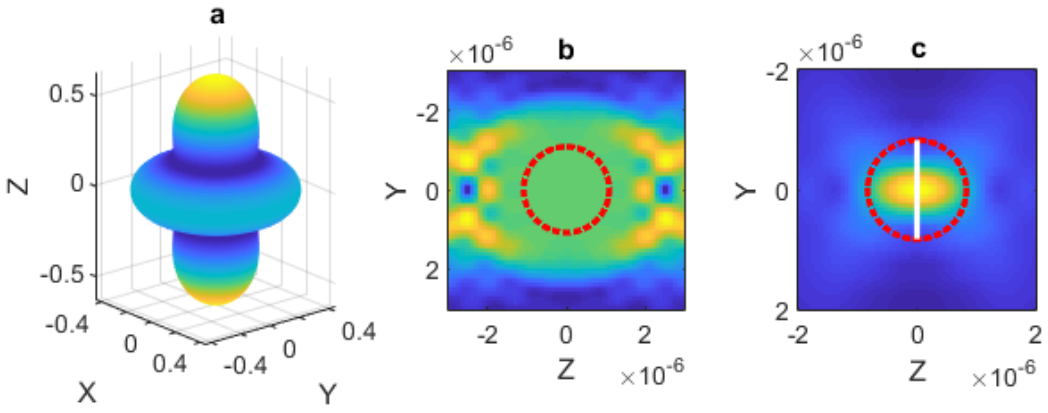


Fig. 5.1: Three different VSWF beams. (a) a quadrapole mode, visualised in the far-field. (b) a plane wave, only valid within the N_{max} region shown by the red circle. (c) a Gaussian beam where most of the beam information passes through the aperture shown by the white line, the equivilant N_{max} region is shown by the red circle.

Translations are not typically reversible. If the beam is translated away from the origin, the translated N_{max} will need to be larger than the original N_{max} in order to contain the same information. If the N_{max} of the translated beam is not large enough, the translation will be irreversible.

The above discussion considered only incident beams. For scattered beams, the scattering is described exactly by the multipole expansion for the region inside the particle's N_{max} and the accuracy depends on how accurately the T-matrix models the particle. As soon as a scattered field is translated, the N_{max} at the new coordinate origin describes the region where the fields are accurately approximated.

Note: This section is based on the user manual for OTT 1.2. The new text includes a discussion about non-square transations, i.e. different original and translated N_{max}.

5.2 Scattering and the Rayleigh Hypothesis

In order to represent non-spherical particles with a T-matrix we assume the particle scatters like an inhomogeneous sphere. The T-matrix for the light scattered by this particle is typically only valid outside the particle's circumscribing sphere. This idea is illustrated in figure Fig. 5.3 a. For isolated particles, this doesn't normally cause a problem. Care should be taken when simulating more than one particle when the circumscribing spheres overlap, see figure Fig. 5.3 b; or when using the fields within the circumscribing sphere.

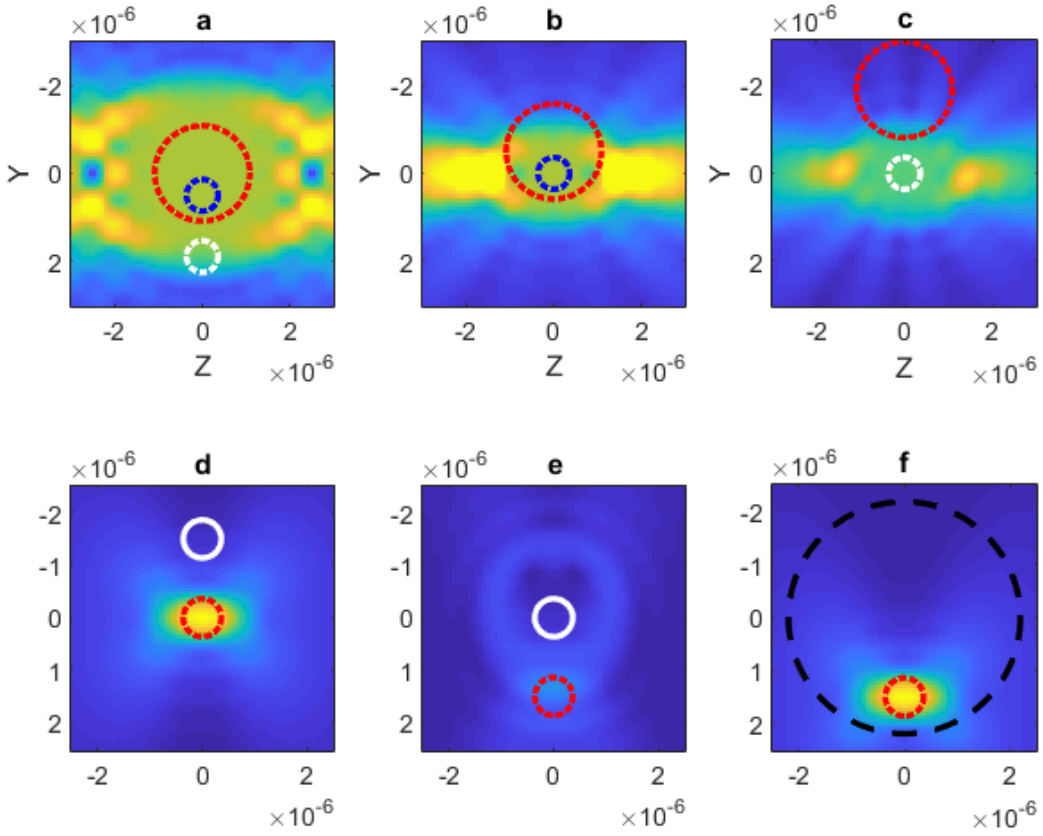


Fig. 5.2: Effect of regular translations on plane waves (a-c) and a focused Gaussian beam (d-f). (a) shows a plane wave whose Nmax region is marked by the red dashed line. The beam can be translated to the blue circle accurately, shown in (b), but cannot be translated to position outside the Nmax region such as the white circle shown in (c). In this case, the region in (c) is still fairly flat but the amplitude is not preserved. (d) show a Gaussian beam with $N_{max} = 6$ (red-line). The beam can be translated anywhere acurately, but the translation is only reversible if the new N_{max} includes the origin as illustrated by (e) irreversible and (f) reversible.

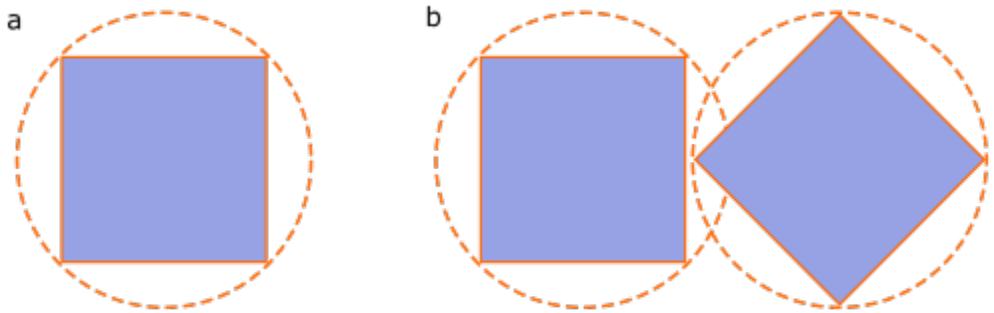


Fig. 5.3: (a) The T-matrix for a cube is calculated assuming a circumscribing sphere (illustrated by the outer circle). (b) Two particles whose circumscribing spheres overlap may cause numerical difficulties.

5.3 Point-matching and angle projections

Several of the beam generation functions in the toolbox support different angular mapping/scaling factors for the projection between the Paraxial far-field and the angular far-field. These factors come about due to the unwrapping of the lens hemisphere onto a plane. Two possible unwrapping techniques are shown in Fig. 5.4 along with the corresponding fields for a Gaussian beam using these two unwrapping methods. One technique (the `tantheta` option for *`ott.BscPmGauss`*) results in more power at higher angles. In the paraxial limit, both these methods produce similar results. A realistic lens is likely somewhere between these two models; at present not all OTT functions support arbitrary mapping functions.

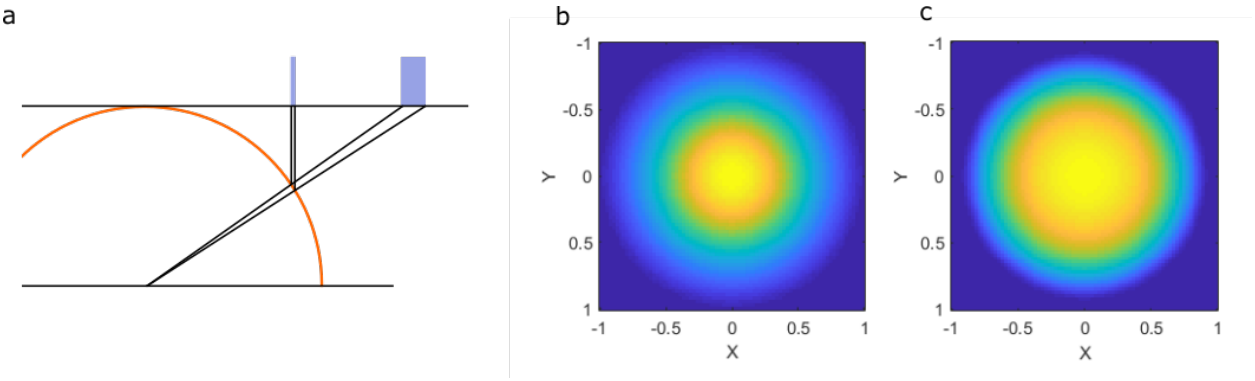


Fig. 5.4: The difference in angular scaling comes from the projection between the lens hemisphere and the lens back-aperture. (a) shows an illustration of the difference in power for the sample angle with two mappings. (b) shows the projected field of a Gaussian beam back aperture with the $sin(\theta)$ mapping and (c) a $tan(\theta)$ mapping.

5.4 Beam truncation angle (for `ott.BscPmGauss`)

Warning: This section will move in a future release.

`ott.BscPmGauss` can be used to simulate various Gaussian-like beams. By default, the class doesn't truncate the beams as a normals microscope objective would, this can be seen in the following example (shown in figure Fig. 5.5):

```
figure();
NA = 0.8;

subplot(1, 2, 1);
beam = ott.BscPmGauss('NA', NA, 'index_medium', 1.33);
beam.basis = 'incoming';
beam.visualiseFarfield('dir', 'neg');
title('Default');

subplot(1, 2, 2);
beam = ott.BscPmGauss('NA', NA, 'index_medium', 1.33, ...
      'truncation_angle', asin(NA./1.33));
beam.basis = 'incoming';
beam.visualiseFarfield('dir', 'neg');
title('Truncated');
```

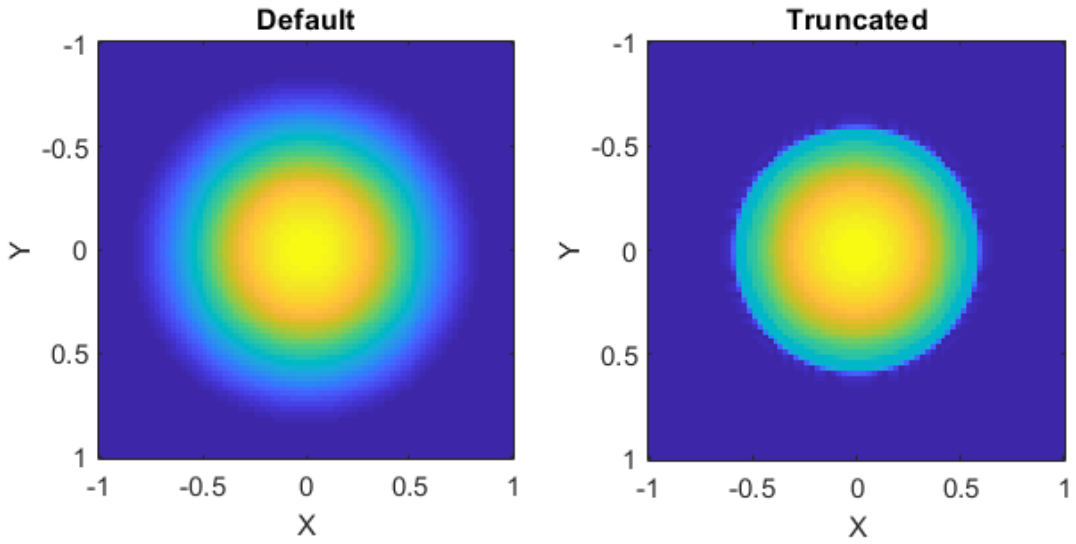


Fig. 5.5: Example output from the `ott.BscPmGauss` showing the far-field intensity patterns of two Gaussian beams with the same numerical aperture. (left) shows the default output, where the Gaussian falls off gradually to the edge of the hemisphere. (right) shows a beam truncated, effectively simulating a microscope back-aperture.

CHAPTER

SIX

FURTHER READING

This page lists paper describing parts of the toolbox.

Papers describing the toolbox

- T. A. Nieminen, V. L. Y. Loke, A. B. Stilgoe, G. Knoener, A. M. Branczyk, N. R. Heckenberg, H. Rubinsztein-Dunlop, "Optical tweezers computational toolbox", Journal of Optics A 9, S196-S203 (2007)
- T. A. Nieminen, V. L. Y. Loke, G. Knoener, A. M. Branczyk, "Toolbox for calculation of optical forces and torques", PIERS Online 3(3), 338-342 (2007)

More about computational modelling of optical tweezers:

- T. A. Nieminen, N. R. Heckenberg, H. Rubinsztein-Dunlop, "Computational modelling of optical tweezers", Proc. SPIE 5514, 514-523 (2004)

More about our beam multipole expansion algorithm:

- T. A. Nieminen, H. Rubinsztein-Dunlop, N. R. Heckenberg, "Multipole expansion of strongly focussed laser beams", Journal of Quantitative Spectroscopy and Radiative Transfer 79-80, 1005-1017 (2003)

More about our T-matrix algorithm:

- T. A. Nieminen, H. Rubinsztein-Dunlop, N. R. Heckenberg, "Calculation of the T-matrix: general considerations and application of the point-matching method", Journal of Quantitative Spectroscopy and Radiative Transfer 79-80, 1019-1029 (2003)

The multipole rotation matrix algorithm we used:

- C. H. Choi, J. Ivanic, M. S. Gordon, K. Ruedenberg, "Rapid and stable determination of rotation matrices between spherical harmonics by direct recursion" Journal of Chemical Physics 111, 8825-8831 (1999)

The multipole translation algorithm we used:

- G. Videen, "Light scattering from a sphere near a plane interface", pp 81-96 in: F. Moreno and F. Gonzalez (eds), Light Scattering from Microstructures, LNP 534, Springer-Verlag, Berlin, 2000

More on optical trapping landscapes:

- A. B. Stilgoe, T. A. Nieminen, G. Knoener, N. R. Heckenberg, H. Rubinsztein-Dunlop, "The effect of Mie resonances on trapping in optical tweezers", Optics Express, 15039-15051 (2008)

Multi-layer sphere algorithm:

- W. Yang, "Improved recursive algorithm for light scattering by a multilayered sphere", Applied Optics 42(9), (2003)

CHAPTER

SEVEN

DOCUMENTATION TERMS OF USE

MATLAB MODULE INDEX

O

INDEX

T

U

V

W

X