OTT Optical Tweezes Toolbox

OTT Documentation

Isaac Lenton

Apr 19, 2022

TABLE OF CONTENTS

CHAPTER

ONE

INTRODUCTION

Welcome to the documentation for the Optical Tweezers Toolbox (OTT). This section provides a brief overview of the toolbox. Subsequent sections will guide you through installing the toolbox, and running the basic examples. The *Reference* section is automatically generated from the toolbox source code and provides a list of all the major toolbox functions and usage. The final section provide additional reference material for toolbox concepts.

Contents

1.1 About

The Optical Tweezers Toolbox is a collection of methods for modelling optically trapped particles. The toolbox includes methods for modelling tightly focussed laser beams, optical scattering, fluidic drag, and dynamics of optically trapped particles. The toolbox can be used to calculate the optical forces and torques acting on spherical and non-spherical particles in a laser beam. In addition, the toolbox includes methods for calculating fluidic drag and Brownian forces, as well as examples for basic dynamics simulations.

This is the second major release of the optical tweezers toolbox. Since version 1, the toolbox has been substantially re-written in order to focus on describing *beams* and *particles* instead of specific numerical methods. This version of the toolbox implements a more modular design using modules and objects. The new modular design is intended to make it easier to integrate other optical force calculation methods such as geometric optics and the discrete dipole approximation in a upcoming release. The other major change from version 1 is the inclusion of methods for calculating non-optical forces and simulating the dynamics of particles.

Previous releases can be downloaded from the OTT GitHub page.

1.2 Dependencies

The toolbox should be compatible with **Matlab 2018a or newer**. Previous versions of the toolbox were compatible with GNU Octave, however the current version does not appear to be compatible in our most recent test.

1.3 Licence

Except where otherwise noted, this toolbox is made available under the Creative Commons Attribution-NonCommercial 4.0 License. For full details see the file `LICENSE.md`. For use outside the conditions of the license, please contact us. The toolbox includes some third-party components, information about these components can be found in the documentation and corresponding file in the `thirdparty` directory.

If you use the toolbox in academic work, please cite it as follows

Todo: Add how to cite, and in README too

or using the following bibtex entry

Todo: Add how to cite, and in README too

1.4 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).

1.5 Contact us

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

CHAPTER

TWO

GETTING STARTED

This section will guide you through installing the toolbox and running your first toolbox commands either through the graphical user interface (GUI) or on the command line.

Contents

2.1 Installation

There are several methods for installing the toolbox. If you are only interested in using the latest stable release of the toolbox, the easiest method is via the Matlab Addons Explorer. Alternatively, you can download a specific release from the OTT GitHub page. The following sections will guide you through installing the toolbox and verifying that it is on the Matlab path.

2.1.1 Installing via Matlab Addons Explorer

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 log-in or create a Mathworks account to complete this step.

2.1.2 Using a .zip or cloning the repository

The latest version of OTT can be downloaded from the OTT GitHub page. Either select the "Code" button near the top of the screen and select your preferred download method, or navigate to the release page and select the .zip file for the desired release. Alternatively, you can clone the git repository directly. 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 downloaded a .zip file, you will need to extract the files to somewhere on your computer before proceeding.

Next, you need to tell Matlab where to find OTT. 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 file). The above line can be added to the start of each of your files or for a more permanent solution you can add it to the Matlab startup script.

2.1.3 Verifying the installation

To verify that the toolbox can be found by Matlab, type

```
help ott
```

into the Matlab command window, which should display the contents of `+ott/Contents.m` if everything installed correctly. If you have multiple versions of the toolbox installed and you wish to check which version is currently active, you can type

```
what ott
```

2.2 Using the graphical user interface

The toolbox includes a graphical user interface for many of the most common tasks. The user interface applications are located in the `ott.ui` sub-package. The easiest way to launch the graphical user interfaces is via the Launcher app. To start the Launcher, simply run the following on the Matlab command line (after the toolbox has been installed):

```
ott.ui.Launcher
```

Todo: Update the how to cite text in the Launcher (and finish launcher)

2.3 Using the toolbox functions

The toolbox contains a collection of functions and classes for performing a range of optical tweezers related simulation tasks. Different tasks, such as simulating drag or describing geometric shapes, are split into different sub-packages. Details about these sub-packages can be found in the *Reference* section or a short list will be printed to the screen when you run `help ott`. Within each package are either functions, classes, or other sub-packages which further group functions/classes based on their purpose. For example, the `beam` sub-package contains several classes for generating beams. In order to create a new beam instance you can either use the class constructor or a suitable static method (if provided). For example, to create a new Gaussian beam you would call either the `Gaussian` or `FromNa` method of the Gaussian class, for example

```
beam1 = ott.beam.Gaussian()
beam2 = ott.beam.Gaussian.FromNa(0.9)
```

In both cases you need to prefix the class name with the package name. If you intend to use a range of methods from one package, it is possible to import that package using

```
import ott.beam.*
```

Except for static functions (such as the `FromNa` method above), most class functions cannot be called without an instance of the class. For example, all `ott.beam.Beam` object implement a function called `efield` which calculates the electric field around the coordinate origin. In order to use this function you need to first construct a valid beam object (for example, using the `Gaussian` or `Gaussian.FromNa` methods above). The following example shows how to create a new Gaussian beam and calculate the field near the origin with the `efield` method.

```
beam = ott.beam.Gaussian()
E = beam.efield([0;0;1e-6])    % Calculate field near origin
```

The `examples` directory includes multiple examples demonstrating various features of the toolbox. To get started writing your own code, we recommend that you start by working through the examples and reading the *Examples* section of this manual. To get help on a specific method or class, you can either type `help <name of method>` or lookup the method/class in the *Reference* section of this manual. For further information on using Matlab packages and classes, refer to the Mathworks OOP documentation.

CHAPTER

THREE

EXAMPLES

This section describes key toolbox features and examples. For information on installing and getting started with the toolbox, refer to the *Getting Started* section. Further examples and live scripts can be found in the `examples` directory.

3.1 *ottBeams* Example : Creating and visualising beams

This example demonstrates how to create and use different kinds of optical tweezers toolbox (OTT) beams. The code for this example can be found in the examples directory, if OTT has been added to the path, run:

```
open examples/ottBeams.m
```

A live script is also available for this example:

```
open examples/liveScripts/beams.mlx
```

This example assumes the toolbox has been added to the Matlab path, for information, see adding-ott-to-matlabs-path.

Concents

3.1.1 Creating a beam

OTT provides two main methods for creating beams: using the classes in the *ott.beam* package and using the classes in the *ott.bsc* package. The *ott.beam* package is intended to provide a easy to use, high-level interface for interacting with beams. Internally, the *ott.beam* classes use the functions declared in *ott.bsc*, and in some cases, using the *ott.bsc* classes directly can give better run-times. However, for most use cases, the *ott.beam* classes should be adequate. In this example, we will focus on the *ott.beam* package.

The *ott.beam* package provides several classes representing common beams, for example, to create a Gaussian beam, call:

```
beam = ott.beam.Gaussian();
```

This create the default Gaussian beam (see the `ott.beam.Gaussian` documentation for the exact specification). We can change the beam properties such as the medium speed and power using the beam's properties or methods, for example:

```
beam.power = 0.1;           % Set power [Watts]
beam.speed = 3e8/1.33;      % Set speed [meters/second]
```

Some properties, such as wavelength, need to be set using beam methods. Beams are not handle classes, as such, it is important to store the resulting beam object. The following sets the wavelength by keeping the speed fixed:

```
beam = beam.setWavelength(532e-9, 'fixedSpeed'); % Set wavelength [meters]
```

Alternatively, we can create the beam with the desired properties at the start by passing a list of named arguments to the constructor:

```
beam = ott.beam.Gaussian('power', 0.1, ...
    'index_medium', 1.33, 'omega', 2*pi*3e8/532e-9);
```

Many of the *ott.beam* classes provide alternative construction methods for convenience, for example, the Gaussian and Laguerre–Gaussian beams provide a `FromNa` method that accepts a numerical aperture value as the first argument.

To view our beam, we can use one of the beam visualisation methods, for example, the following creates a XY slice through the beam focus:

```
beam.visNearfield();
```

This creates a new plot in the current figure window (or creates a new figure if required). Alternatively, we can request the resulting image data and plot the results ourselves (this is usually done in conjunction with specifying the image range):

```
xrange = linspace(-1, 1, 80)*1e-6;   % Range in meters
yrange = xrange;
im = beam.visNearfield('range', {xrange, yrange});
figure();
contour(xrange, yrange, im);
xlabel('X Position [m]');
ylabel('Y Position [m]');
```

3.1.2 Translations and rotations

Beams have a `position` and `rotation` property. These properties are applied to the beam whenever the beam is used (for example, when a visualisation method is called or when `getData` is called).

The position property is a 3x1 numeric vector. To shift the beam by 1 wavelength in the x direction, we can directly set the position property:

```
beam.position = [1;0;0]*beam.wavelength;
```

Alternatively, we can use the `ott.utils.TranslateHelper.translateXyz()` method. The translation method applies the translation on top of any existing displacement and returns a new copy of the beam, for example, to translate our previous beam along the Y-direction, we could use:

```
tbeam = beam.translateXyz([0;1;0]*beam.wavelength);
```

Rotations are stored as 3x3 rotation matrices. As with the `position` property, we can also directly set the `rotation` property, however it is often easier to use the `rotate*` methods from `ott.utils.RotateHelper`. The following rotates the beam pi/2 radians about the Y axis. When the beam is used, the rotation is applied before the translation:

```
rbeam = beam.rotateY(pi/2);
```

3.1.3 Arrays of beams

The toolbox supports three kinds of arrays: Coherent arrays, Incoherent arrays, and generic arrays. Incoherent/Coherent arrays represent beams which can be represented by a finite set of sub-beams. Generic arrays are simply collections of multiple beams.

To create a generic array, simply use Matlab's array syntax, for example:

```
beams = [ott.beam.Gaussian(), ...
    ott.beam.LaguerreGaussian('lmode', 10)];
```

Most operations can be applied to generic arrays. The result is the same as applying the operation to each element of the array. For example, to translate the array of beams:

```
tbeams = beams.translateXyz([1;0;0]*beam.wavelength);
```

Or to set the position of each beam with deal:

```
[tbeams.position] = deal([1;0;0]*beam.wavelength);
```

Or directly with element access:

```
tbeams(1).position = [1;2;3]*beam.wavelength;
```

Coherent and Incoherent arrays can be created using `ott.beam.Array`, for example:

```
cbeams = ott.beam.Array(beams, 'arrayType', 'coherent');
```

3.1.4 Calculating the change in momentum between two beams

The *ott.beam.Beam* class provides methods for calculating the change in momentum between two beams. Although it is more common to calculate the force acting on a particle (see the `ottForce.m` example), the following shows how to calculate the change in momentum between two beams:

```
beam1 = ott.beam.Gaussian();
beam2 = beam1.rotateY(pi/2);
force = beam1.force(beam2)
```

3.1.5 Creating a custom beam

Although the toolbox has several different beams commonly used in optical trapping (for a complete list, see the *beam* package reference section), it is sometimes necessary to create a custom beam. The most common scenario is when modelling an SLM or the experimentally measured field at the back aperture of the focussing objective. For this task we can use the *PmParaxial* class (for more control over the fields we could also use the *ott.bsc* classes). The following example shows how we could model a phase-only SLM illuminated by a Gaussian-like beam:

```
% Generate coordinates for pattern
x = linspace(-1, 1, 20);
y = linspace(-1, 1, 20);
[X, Y] = ndgrid(x, y);

% Calculate incident field
E0 = exp(-(X.^2 + Y.^2)./4);

% Calculate SLM-like pattern
kx = 2;
phi = 2*pi*kx*x;

% Calculate field at back aperture
E = E0 .* exp(1i*phi);

% Calculate beam
beam = ott.beam.PmParaxial.InterpProfile(X, Y, E);
```

3.1.6 Improving runtime

Toolbox beams are represented using a vector spherical wave function expansion. Depending on how many terms are included in the expansion, visualisation and translation functions can take quite a while. One method to improve the runtime is to reduce the number of terms in the expansion. By defaut, finite beams (such as Gaussians) are calcualted with ~10,000 terms and then truncated such that the resulting beam power doesnt drop bellow 2% of the original beam power. This threshold can be changed using, for example, to change it to 10% use:

```
ott.beam.BscFinite.getSetShrinkNmaxRelTol(0.01);
```

Next time we create a beam, it will use this new tolerance:

```
beam = ott.beam.Gaussian();
disp(beam.data.Nmax);
tic
```

(continues on next page)

(continued from previous page)

```
beam.visNearfield();
toc

% Change back to 2%
ott.beam.BscFinite.getSetShrinkNmaxRelTol(0.02);
```

For repeated field calculation at the same locations, there is a lot of data that can be re-used in the field calculation functions. Both the visNearfield and Gaussian beam generation functions require calculating fields. To speed up these methods, we can store the field data from a previous run. The following creates a plot with 2 different beams:

```
figure();
range = [1,1]*2e-6;

% Generage first beam with no prior data.  We use the recalculate
% method explicitly so we can get the returned data for repeated
% calculations.  For visualisation, we also get the returned data,
% but we also need to specify the plot axes explicitly to show the plot.
subplot(1, 2, 1);
tic
beam = ott.beam.LaguerreGaussian('lmode', 9, 'calculate', false);
[beam, dataBm] = beam.recalculate([]);
[~, ~, dataVs] = beam.visNearfield('range', range, 'plot_axes', gca());
toc

% Generate second beam with prior data for both beam and visNearfield.
subplot(1, 2, 2);
tic
beam = ott.beam.LaguerreGaussian('lmode', 7, 'calculate', false);
beam = beam.recalculate([], 'data', dataBm);
beam.visNearfield('range', range, 'data', dataVs);
toc
```

Different beams/methods support different optional parameters. It is not always faster to pass the data structure as an input. See the documentation for notes on improving speed and the supported arguments these methods support.

3.1.7 Further reading

For the full range of beams currently inculded in the toolbox, refer to the beams-package part of the reference section. The example code used to generate the overview figure in the reference section can be found in the `examples/packageOverview/` directory. More advanced beam functionality can be implemented by directly using the beam shape coefficient classes (the *`ott.bsc`* package). For a example which uses both *`ott.beam.Beam`* and *`ott.bsc.Bsc`*, see `examples/ottLandscape.m`.

3.2 *ottParticles* Example : Creating and using particles

This examples demonstrates how to create and use different kinds of optical tweezers toolbox (OTT) particles. The code for this example can be found in the examples directory, if OTT has been added to the path, run:

```
open examples/ottParticles.m
```

A live script is also available for this example:

```
open examples/liveScripts/particles.mlx
```

This example assumes the toolbox has been added to the Matlab path, for information, see adding-ott-to-matlabs-path.

Concents

3.2.1 Creating a particle with a simple geometry

OTT particles encapsulate the geometry, optical scattering and fluid drag information for a particle in an optical tweezers simulation. There are several ways to create particles, the simplest involve using the *FromShape* methods of the particle classes. These methods use the corresponding *FromShape* methods of the drag and T-matrix classes, however care should be taken when using large or complex particles, as these methods may not always give good approximations for the input geometry. The following create a shape and encapsulates it in a particle:

```
% Generate a geometric shape (units of meters)
shape = ott.shape.Sphere(1e-6);

% Create a particle
% For the T-matrix method, we need to specify relative index and wavelength
particle = ott.particle.Fixed.FromShape(shape, ...
     'index_relative', 1.2, 'wavelength0', 1064e-9);
```

We can visualise the particle using the `surf` method, this simply calls the shape's corresponding `surf` method:

```
particle.surf();
```

3.2.2 Creating a particle with custom properties

Particle properties don't have to be related: this is useful when there is no available method for modelling the drag or scattering of a particle, or when we want to visualise the shape using a simpler geometry.

To create a particle with custom properties, we can use the *Fixed* sub-class and provide our own T-matrix, drag and geometry. This class simply stores the provided properties.

For instance, we can create a particle instance with the geometry of a cylinder, T-matrix for a spheroid, and drag for a cylinder:

```
% Create geometry
wavelength = 1e-6;
shape = ott.shape.Cylinder(wavelength, wavelength);

% Create drag using FromShape
drag = ott.drag.Stokes.FromShape(shape, 'viscosity', 0.001);

% Create T-matrix for spheroid.
% There are several T-matrix methods included in the toolbox, Smarties
% works well for spheroidal particle.  This example creates a spheroid
% with similar dimensions to our Cylinder.  T-matrix methods use distance
% units with the dimensions of wavelength (particle/beams use meters).
index_relative = 1.2;
tmatrix = ott.tmatrix.Smarties(...
    shape.radius ./ wavelength, shape.height ./ wavelength, index_relative);
```

And then we can store these properties in a particle instance:

```
particle = ott.particle.Fixed(shape, 'drag', drag, 'tmatrix', tmatrix);
```

3.2.3 Translations and rotations

Similar to beams, particles have `rotation` and `position` properties which can be used to control the position/rotation of the particle.

Both rotation and position can be directly set, for example:

```
particle.position = [1;0;0]*wavelength;
```

And properties can be adjusted using the translate/rotate methods. As with beams, the rotate/translate methods return a copy of the object:

```
new_particle = particle.rotateY(pi/2);
```

We can see the effect of these operations by generating a surface plot with the `surf` method:

```
new_particle.surf();
```

For additional example, see the rotation/position part of the *ottBeams Example : Creating and visualising beams* (`ottBeam.m`) example.

3.2.4 Calculate optical scattering

Particles can scatter beams (when there is an appropriate T-matrix definition).

For this example, lets use a Gaussian beam:

```
beam = ott.beam.Gaussian();
```

And instead of only calculating the external fields, we can also calculate the internal fields by passing `'internal', true` to the particle constructor:

```
shape = ott.shape.Sphere(1e-6);
particle = ott.particle.Particle.FromShape(shape, ...
    'internal', true, 'index_relative', 1.2);
```

Not all T-matrix calculation methods support calculating internal fields. The T-matrix method that is used depends on the geometry (for this case, a sphere, the internal method should give fairly accurate results).

To calculate the scattering, we can use the `scatter` method:

```
sbeam = beam.scatter(particle);
```

The scattered beam stores an instance of the particle and the incident beam, allowing us to easily visualise the internal and external fields, for example, the following outputs the fields shown in [TODO]:

```
sbeam.visNearfield('axis', 'y', [1,1]*2e-6);
```

3.2.5 Calculate forces

Force can be calculated either directly using the `force` method of the scattered beam or using the `force` method of the incident beam. With the scattered beam:

```
force = sbeam.force();
```

And, with the incident beam:

```
force = beam.force(particle);
```

With both methods, the resulting force has units of Newtons. The incident beam method has the advantage that we can also specify a 3xN array of positions or rotations to apply to the particle, these replace any existing particle translations/rotations:

```
positions = randn(3, 5)*1e-6;
forces = beam.force(particle, 'position', positions);
```

Additional force calculation examples are provided in the *ottForce Example : Calculate and visualise force* (`ottForce.m`) example.

3.2.6 Further reading

The *Advanced examples* section shows how the particle class can be used for various tasks including dynamics simulations.

T-matrices cannot be calculated for all shapes, but if the particle is homogeneous and small enough to simulate, it should be possible to compare the scattering by the T-matrix to the scattering directly with DDA, giving an estimate for accuracy.

DDA should work with most shapes, but this hasn't been thoroughly tested, if you find something interesting, let us know. For inspiration with creating different particle shapes, take a look at the `ott.shape` reference section and the shapes used in the `examples/packageOverview` examples.

3.3 *ottForce* Example : Calculate and visualise force

This example shows how to calculate and visualise force on a sphere. This example includes two visualisations: a 1-D force profile in the x/z directions, and a 2-D force profile in the x-z plane.

The example assumes the user is already familiar with the beam/particle classes. For examples of the beam/particle functionality, see the *ottBeams Example : Creating and visualising beams* and *ottParticles Example : Creating and using particles* sections of the documentation.

The code for this example can be found in the examples directory, if OTT has been added to the path, run:

```
open examples/ottBeams.m
```

A live script is also available for this example:

```
open examples/liveScripts/beams.mlx
```

This example assumes the toolbox has been added to the Matlab path, for information, see adding-ott-to-matlabs-path.

Concents

3.3.1 Setup the particle

The first step is to setup the particle. For this simulation we use a spherical particle with a 1 micron radius. We describe the geometry and let the `FromShape` method choose an appropriate T-matrix method (for spheres, this will be … *ott.tmatrix.Mie*):

```
% Create geometry for shape
radius = 1.0e-6;      % Sphere radius [m]
shape = ott.shape.Sphere(radius);

% Setup particle
particle = ott.particle.Particle.FromShape(shape, 'index_relative', 1.2);
```

3.3.2 Setup the beam

The next step is to calculate the incident beam. The following creates a Gaussian beam tightly focussed by a microscope objective. The back aperture of the microscope objective creates a hard edge, we model this by setting the `truncation_angle` parameter of the Gaussian beam class:

```
% Calculate truncation angle for modelling a microscope objective
NA = 1.2;
index_medium = 1.33;
truncation_angle = asin(NA/index_medium);

% All beam parameters have SI units (radians for angles)
beam = ott.beam.Gaussian(...
    'waist', 1.0e-6, 'power', 0.01, 'truncation_angle', truncation_angle, ...
    'index_medium', index_medium, 'wavelength0', 1064e-9);
```

3.3.3 Generate beam visualisation

It is always good to check the beam looks how we would expect. If the beam doesn't look correct then there may be something wrong with the choice of parameters or some parameters may be outside the range of values for which *ott.beam.Gaussian* gives reasonable results (in which case, consider using the underlying *ott.bsc* classes directly). We can visualise the above beam using:

```
figure();
subplot(1, 2, 1);
beam.visFarfield('dir', 'neg');
title('Far-field');
subplot(1, 2, 2);
beam.visNearfield();
title('Nearfield');
```

3.3.4 Calculate forces

For spherical particles in beams with only a single focus, we are often interested in the 1-dimensional position-force profiles. These can be calculated by simply using the beam's `force` method and specifying a `3xN` array of locations. For the axial force:

```
% Calculate force along axis
z = linspace(-1, 1, 100)*6*beam.wavelength;
fz = beam.force(particle, 'position', [0;0;1].*z);
```

For the radial force, we are interested in the force near the equilibrium (both in the axial and radial directions) The following code uses the axial force profile and then uses the *`ott.tools.FindTraps1d`* class to estimate where the axial equilibrium is. The radial force profile is then calculated at points passing through the axial equilibrium. If the particle/beam does not have an axial equilibrium then the coordinate origin is used:

```
% Find traps along axis
traps = ott.tools.FindTraps1d.FromArray(z, fz(3, :));

% Get the axial equilibrium (might have no trap, in which case use z=0)
if isempty(traps)
  z0 = 0.0;
else
  z0 = traps(1).position;
end

% Calculate radial force
r = linspace(-1, 1, 100)*6*beam.wavelength;
fr = beam.force(particle, 'position', [1;0;0].*r + [0;0;z0]);
```

The particle/beam functions use SI units and the resulting force/torque have units of [Newtons] and [Newton meters]. The following generates plots of the force profiles:

```
subplot(1, 2, 1);
plot(z, fz);
xlabel('z position [m]');
ylabel('force [N]');
legend({'X', 'Y', 'Z'});

subplot(1, 2, 2);
plot(r, fr);
xlabel('x position [m]');
ylabel('force [N]');
legend({'X', 'Y', 'Z'});
```

If the beam has an orbital component (for example, if there is orbital or spin angular momentum) then there may be a force component around the beam axis.

3.3.5 Generate 2-D visualisation of force

When the trap is harmonic, the axial/radial force profiles are sufficient to characterise the properties of the trap. However, this is rarely the case for complex beams or particles far from equilibrium.

An alternative method for visualising the optical force is to plot a colour map or vector field of the optical force as a function of position. The following plots such a visualisation (this may take some time depending on how many positions are required for the visualisation):

```
r = linspace(-1, 1, 40)*6*beam.wavelength;
z = linspace(-1, 1, 40)*6*beam.wavelength;

[R, Z] = meshgrid(r, z);
F = beam.force(particle, 'position', {R, 0*R, Z});

% Calculate magnitude of force
Fmag = vecnorm(F, 2, 3);

% Generate colormap/quiver plot showing force
imagesc(R, Z, Fmag);
hold on;
quiver(R, Z, Fmag(:, :, 1), Fmag(:, :, 3));
hold off;
xlabel('x position [m]');
ylabel('y position [m]');
cb = colorbar();
yaxis(cb, 'Force [N]');
```

3.3.6 Further reading

The main part of an optical tweezers simulation is force calculation, however another important component is being able to simulate how the particle moves, this requires calculating the drag and potentially thermal motion, and simulating the particle for a while. The toolbox has preliminary support for dynamics simulations using a fixed time step (see the `ottDynamics.m` example and the `ott.tools.Dynamics` class).

3.4 Advanced examples

This section provides a very brief overview of the more advanced examples currently included in the toolbox. These examples are all contained in the `examples/` directory.

Previous versions of the toolbox included other examples, most functionality is still present in this version of the toolbox but examples have not all been included.

Contents

- *DDA Vaterite*
- *Neural Networks for fast simulation*
- *Further reading*

3.4.1 Landscape

Calculates trapping landscapes. These show how optical trap properties, such as trap stiffness or maximum optical force, vary as a function of particle properties (such as radius and refractive index). The `ottLandscape.m` example calculates how maximum trap depth varies with different size and refractive index spherical particles, example output is shown in [TODO].

These types of calculations often involve looping over parameters. In some cases it is more optimal to pre-compute translations or rotations, such as (in the example) translations along the beam axis for different sized particles. This version of the toolbox also adds support for re-using VSWF data (for faster T-matrix calculation and field calculation), this is a feature which may be useful for certain types of trapping landscapes. See `ott.utils.VswfData` and its usage in the various beam and T-matrix classes.

3.4.2 Dynamics

The `ottDynamics.m` example shows how the `ott.tools.Dynamics` class can be used to simulate particle dynamics. This requires a beam, T-matrix and drag tensor (which can be provided as a *`ott.particle.Particle`* instance); and solves the Langevin equation using a fixed time step method.

The example creates a spherical particle and generates a trace of the position and a 2-D histogram of the XY position, example output is shown in [TODO].

3.4.3 Wall Effects

This `ottWallEffects.m` example builds on the `ottDynamics.m` example by using a Sphere-Wall drag tensor. Currently there is not optical interaction between the particle/beam/wall, this may be added in a future release. The example generates visualisations of [TODO], example output is shown in [TODO].

3.4.4 Non-spherical particles

This version of the toolbox is focussed on T-matrix methods for scattering calculations. Future versions of the toolbox aim to include DDA, shape surface approximation, and geometric optics. For now, the main methods for modelling almost-arbitrary shaped particles are point-matching, extended boundary conditions method (EBCM), and discrete dipole approximation.

The `ottNonSpherical.m` example shows how EBCM, DDA and PM can be used to model a cylindrical particle. For certain sizes, all three methods agree well, other sizes/aspect ratios, the methods start to disagree, as shown in [TODO]. Calculating T-matrices using different methods is a good method for validating the generated T-matrix/predicted force.

3.4.5 DDA Vaterite

The `ottDdaVaterite.m` example shows some of the more advanced features of the DDA implementation: the implementation can be used to model a inhomogeneous birefringent particle. The example calculates T-matrices for vaterites with a sheaf-of-wheat structure and calculates the torque about the z axis. Example output is shown in [TODO].

The DDA implementation also supports calculating only specific columns of the T-matrix. This can be useful when the T-matrix is illuminated by only a beam with particular orders; or when calculating T-matrices in parallel.

3.4.6 Neural Networks for fast simulation

The `ottNeuralNetwork.m` example shows how a neural network can be trained to rapidly predict optical force data. This requires first generating a large training data set which is used to train the network. Once trained, the network can be used for rapid force calculations (for inputs within the bounds of the training data) or easily shared with other researchers.

For this particular problem, interpolation could also be used, however the stored data sets are often much larger than the resulting neural network. Additionally, for problems with additional inputs/outputs (such as predicting force and torque from orientation and position), direct interpolation becomes a much more computationally expensive task.

3.4.7 Further reading

These examples cover most of the advanced functionality in the toolbox. For information about how different methods work, the reference section provides some insight. Otherwise, take a look at the *Further Reading* section or the previous version of the toolbox for additional examples. Experimental (viz., incomplete) new features can also be found in version 2 of the toolbox (available on GitHub).

3.5 Calculating forces with the GUI

Todo: A lot of this needs to be re-written and figures updated

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 in the force example script (`examples/ottForce.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, simulate scattering, and calculate the optical force.

Contents

3.5.1 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. 3.1 should display.

`images/examples/guiExample/generateLgBeam.png`

Fig. 3.1: 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. 3.2. Additionally, a variable should be created in the Matlab workspace for our new beam (we will use this variable later).

`images/examples/guiExample/beamOutput.png`

Fig. 3.2: The Generate LG Beam GUI after clicking the Generate button should now display the beam transverse and axial field distributions.

3.5.2 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. 3.3 shows the GUI after clicking generate.

images/examples/guiExample/tmatrix.png

Fig. 3.3: 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.

3.5.3 Calculating forces profiles

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. 3.4 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.

images/examples/guiExample/force.png

Fig. 3.4: 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.

3.5.4 Calculating force slice

The toolbox also provides a GUI for calculating a force field slice for particle positions in a plane.

Todo: This isn't done yet

CHAPTER

FOUR

REFERENCE

This section contains information about the different functions, packages and classes contained in the toolbox (specifically, the *+ott* directory). Most of this content is automatically generated from the source code documentation (which can be accessed using *help <component>*) with the exception of some additional examples and the summaries included at the start of each of the following sections.

4.1 *beam* Package

The *beam* package provides classes representing optical tweezers toolbox beams. The base class for beams is `Beam`. Sub-classes provide beam specialisations for specific beam types (such as `Gaussian`, `PlaneWave`, `Webber`), point matching for modelling the fields at the back aperture of a objective (`PmParaxial`), arrays of beams (`Coherent`, `Incoherent`), and `Scattered` beams. A summary is presented in Fig. 4.1.

Unlike `ott.bsc.Bsc` instances, `Beam` instances use SI units (i.e., force in Newtons, distance in meters).

Internally, the `Beam` classes use `ott.bsc.Bsc` to represent the fields. Most class properties are defined in separate classes (declared in the `properties` sub-package. In a future release of OTT, the `Beam` interface may change to support other types of field representations.

Contents

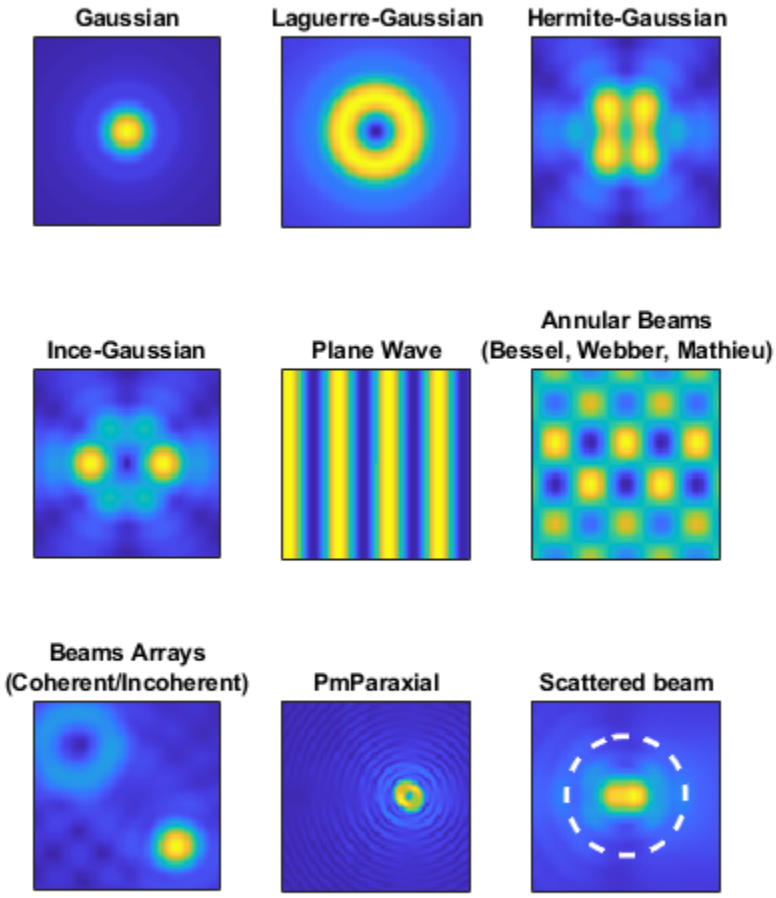

Fig. 4.1: Graphical display of the different kinds of beams currently included in the optical tweezers toolbox.

4.1.1 Base Classes

Beam

class `ott.beam.`**`Beam`**

Provides a high-level view of a optical tweezers toolbox beam. Inherits from `ott.utils.RotationPositionProp` and *matlab.mixin.Hetrogeneous*.

This class is the base class for OTT beams. `Beam` and its sub-classes provide a description of beams but do not implement any of the field or scattering calculation methods. These classes separate properties describing the beam (such as position, power and waist) from properties specific to the numerical calculation (VSWF data, Nmax, apparent power). Internally, `Beam` uses `ott.bsc.Bsc` for field calculations. Depending on the implementation, the `ott.bsc.Bsc` data is either stored or calculated when required.

The other major difference from `ott.bsc.Bsc` is the units. `Beam` uses SI units for all quantities, making integration with dynamics simulations easer.

In OTTv2, this interface will likely be extended to other types of beam/scattering methods (such as paraxial beams or other approximations).

Properties

- position – (3x1 numeric) Location of the beam
- rotation – (3x3 numeric) Orientation of the beam
- wavelength – Wavelength in medium [m]
- wavenumber – Wavenumber in medium [1/m]
- speed – Speed of light in the medium [m/s]
- speed0 – Speed of light in vacuum [m/s]
- scale – Scaling parameter for beam fields (default: 1)

Abstract properties

- omega – Optical angular frequency of light [1/s]
- index_medium – Refractive index of the medium

Methods

- setWavelength – Set wavelength property
- setWavenumber – Set wavenumber property

- scatter – Calculate how a beam is scattered

Field calculation methods

- ehfield – Calculate electric and magnetic fields around the origin
- ehfieldRtp – Calculate electric and magnetic fields around the origin
- ehfarfield – Calculate electric and magnetic fields in the far-field
- eparaxial – Calculate electric fields in the paraxial far-field
- hparaxial – Calculate magnetic fields in the paraxial far-field
- ehparaxial – Calculate electric and magnetic paraxial far-fields

Force and torque related methods

- forcetorque – Calculate the force and the torque between beams
- intensityMoment – Calculate moment of beam intensity in far-field
- force – Calculate the change in momentum between two beams
- torque – Calculate change in angular momentum between beams
- spin – Calculate change in spin momentum between beams

Field visualisation methods

- visNearfield – Generate a visualisation around the origin
- visFarfield – Generate a visualisation at the far-field
- visFarfieldSlice – Visualise the field on a angular slice
- visFarfieldSphere – Visualise the filed on a sphere

Mathematical operations

- times,mtimes – Scalar multiplication of beam fields
- rdivide,mrdivide – Scalar division of beam fields
- uminus – Flip beam field intensity
- plus, minus – Combine beams coherently
- or – Combine beams incoherently

Abstract methods

- efield – Calculate electric field around the origin
- hfield – Calculate magnetic field around the origin
- efieldRtp – Calculate electric field around the origin (sph. coords.)
- hfieldRtp – Calculate magnetic field around the origin (sph. coords.)
- efarfield – Calculate electric fields in the far-field
- hfarfield – Calculate magnetic fields in the far-field
- scatterInternal – Called to calculate the scattered beam

Empty

class ott.beam.**Empty**(*varargin*)

A beam with no fields. Inherits from *Beam*.

This class represents empty space with no fields. This is useful for default parameters to functions or for unassigned elements of beam arrays (part of Hetrogeneous interface).

Properties

- position - (3x1 numeric) Position of the empty space.
- rotation - (3x3 numeric) Orientation of the empty space.
- index_medium – Refractive index of the medium
- wavelength – Wavelength in medium [m]
- wavenumber – Wavenumber in medium [1/m]
- omega – Optical angular frequency of light [1/s]
- speed – Speed of light in the medium [m/s]
- speed0 – Speed of light in vacuum [m/s]

Methods

- efield, hfield, ehfield, … – Returns zeros
- scatter – Returns a scattered beam with empty parts

BscBeam

class ott.beam.**BscBeam**(*varargin*)

Beam class encapsulating a BSC instance. Inherits from `ott.beam.ArrayType`.

This class stores an internal *`ott.bsc.Bsc`* instance which it uses for calculating fields.

Methods in this class assume the beam is a regular beam (i.e., not the outgoing fields from a scattered beam).

Properties

- data – Internal BSC instance describing beam
- apparentPower – Apparent power of the BSC data

Methods

- recalculate – Can be overloaded by sub-classes to update data
- efield, hfield, … – Calculate fields in SI units, uses the `ott.bsc.Bsc` methods internally.
- force, torque, spin – Calculate force/torque/spin in SI units

Supported casts

- ott.bsc.Bsc – Get the BSC data after applying transformations

Additional properties/methods inherited from *Beam*.

BscFinite

class ott.beam.**BscFinite**(*varargin*)

A beam represented by a finite VSWF expansion. Inherits from *BscBeam*.

This class describes beams which can be represented using a finite VSWF expansion. The class stores a `Bsc` instance internally. BSC coefficients at any other location can be found by applying a translation to the beam data.

Far-fields are calculated without applying a translation to the BSC data, instead the fields are calculated at the origin and phase shifted.

As with the *BscBeam* class, this class assumes a regular beam.

Properties

- power – Power applied to the beam in `ott.bsc.bsc`.

Supported casts

- ott.bsc.Bsc – Get the BSC data after applying transformations

Additional properties/methods inherited from *BscBeam*.

BscInfinite

class ott.beam.**BscInfinite**(*varargin*)

Describes a beam with infinite spatial extent. Inherits from *BscBeam*.

This class is useful for describing plane waves, annular beams, and other beams with an infinite spatial extent. Values are only valid within the Nmax region, requesting values outside this region requires the beam to be re-calculated (or, for plane waves/annular beams, translated).

This class overloads the field calculation functions and requests a larger Nmax whenever points outside the valid range are requested. Far-field functions raise a warning that fields may not look as expected.

Properties

- Nmax – Nmax of the stored data

For methods/properties, see *BscBeam*.

4.1.2 Scattered

class ott.beam.**Scattered**(*varargin*)

Describes the beam scattered from a particle. Inherits from `ott.beam.Beam`.

When the internal field and the particle are supplied, the visualisation and field calculation method calculate either the external or internal fields depending on the requested points location. The visualisation methods also provide an option for showing the particle outline.

Properties

- incident – Beam data incident on the particle
- scattered – Beam data scattered by the particle
- outgoing – Outgoing modes radiation from the particle
- incoming – Incoming modes scattered by the particle

- internal – Internal beam data (optional)
- particle – Particle responsible for scattering (optional)

Field calculation methods

- efield – Calculate electric field around the origin
- hfield – Calculate magnetic field around the origin
- ehfield – Calculate electric and magnetic fields around the origin
- efieldRtp – Calculate electric field around the origin (sph. coords.)
- hfieldRtp – Calculate magnetic field around the origin (sph. coords.)
- ehfieldRtp – Calculate electric and magnetic fields around the origin
- efarfield – Calculate electric fields in the far-field
- hfarfield – Calculate magnetic fields in the far-field
- ehfarfield – Calculate electric and magnetic fields in the far-field
- eparaxial – Calculate electric fields in the paraxial far-field
- hparaxial – Calculate magnetic fields in the paraxial far-field
- ehparaxial – Calculate electric and magnetic paraxial far-fields

Force and torque related methods

- force – Calculate the change in momentum between two beams
- torque – Calculate change in angular momentum between beams
- spin – Calculate change in spin momentum between beams
- forcetorque – Calculate the force and the torque between beams

Field visualisation methods

- visNearfield – Generate a visualisation around the origin
- visFarfield – Generate a visualisation at the far-field
- visFarfieldSlice – Visualise the field on a angular slice
- visFarfieldSphere – Visualise the filed on a sphere

4.1.3 Beam types

Gaussian

class `ott.beam.`**`Gaussian`**(*varargin*)

Construct a VSWF representation of a tightly focussed Gaussian beam. Inherits from `ott.beam.BscFinite` and `ott.beam.properties.Gaussian`.

Properties

- waist – Beam waist radius [m]
- index_medium – Refractive index of the medium
- omega – Optical angular frequency of light [1/s]
- position – Position of the beam [m]

- rotation – Rotation of the beam [3x3 rotation matrix]
- power – Beam power [W]
- polbasis – (enum) Polarisation basis ('polar' or 'cartesian')
- polfield – (2 numeric) Field in theta/phi or x/y directions
- mapping – Paraxial to far-field beam mapping
- data – Internal BSC instance describing beam

Methods

- getData – Get data for specific Nmax
- recalculate – Recalculate the beam data

Static methods

- FromNa – Construct a beam specifying NA instead of waist

LaguerreGaussian

`class ott.beam.LaguerreGaussian(varargin)`

Construct a VSWF representation of tightly focussed Laguerre-Gaussian beam. Inherits from `ott.beam.BscFinite` and `ott.beam.properties.LaguerreGaussian`.

Properties

- waist – Beam waist radius [m]
- index_medium – Refractive index of the medium
- omega – Optical angular frequency of light [1/s]
- position – Position of the beam [m]
- rotation – Rotation of the beam [3x3 rotation matrix]
- power – Beam power [W]
- lmode – Azimuthal mode number
- pmode – Radial mode number
- polbasis – (enum) Polarisation basis ('polar' or 'cartesian')
- polfield – (2 numeric) Field in theta/phi or x/y directions
- mapping – Paraxial to far-field beam mapping
- data – Internal BSC instance describing beam

Methods

- getData – Get data for specific Nmax
- recalculate – Recalculate the beam data

Static methods

- FromNa – Construct a beam specifying NA instead of waist

HermiteGaussian

class `ott.beam.`**`HermiteGaussian`**(*varargin*)

Construct a VSWF representation of tightly focussed Hermite-Gaussian beam. Inherits from `ott.beam.BscFinite` and `ott.beam.properties.HermiteGaussian`.

Properties

- waist – Beam waist radius [m]
- index_medium – Refractive index of the medium
- omega – Optical angular frequency of light [1/s]
- position – Position of the beam [m]
- rotation – Rotation of the beam [3x3 rotation matrix]
- power – Beam power [W]
- mmode – Hermite mode number
- nmode – Hermite mode number
- polbasis – (enum) Polarisation basis ('polar' or 'cartesian')
- polfield – (2 numeric) Field in theta/phi or x/y directions
- mapping – Paraxial to far-field beam mapping
- data – Internal BSC instance describing beam

Methods

- getData – Get data for specific Nmax
- recalculate – Recalculate the beam data

Static methods

- FromNa – Construct a beam specifying NA instead of waist

InceGaussian

class `ott.beam.`**`InceGaussian`**(*varargin*)

Construct a VSWF representation of tightly focussed Laguerre-Gaussian beam. Inherits from `ott.beam.BscFinite` and `ott.beam.properties.InceGaussian`.

Properties

- waist – Beam waist radius [m]
- index_medium – Refractive index of the medium
- omega – Optical angular frequency of light [1/s]
- position – Position of the beam [m]
- rotation – Rotation of the beam [3x3 rotation matrix]
- power – Beam power [W]
- lmode – Azimuthal mode number
- porder – Paraxial mode order.
- ellipticity – Ellipticity of coordinates

- parity – Parity of beam ('even' or 'odd')
- polbasis – (enum) Polarisation basis ('polar' or 'cartesian')
- polfield – (2 numeric) Field in theta/phi or x/y directions
- mapping – Paraxial to far-field beam mapping
- data – Internal BSC instance describing beam

Methods

- getData – Get data for specific Nmax
- recalculate – Recalculate the beam data

Static methods

- FromNa – Construct a beam specifying NA instead of waist

PlaneWave

class `ott.beam.`**`PlaneWave`**`(`*varargin*`)`

VSWF representation of a Plane Wave beam. Inherits from `ott.beam.BscInfinite`.

Plane waves support smart translations, where the beam components are phase shifted rather than re-calculated.

Properties

- polarisation – (2 numeric) Polarisation in the x/y directions
- Nmax – Nmax of the stored data
- data – Internal BSC instance describing beam

Mathieu

class `ott.beam.`**`Mathieu`**`(`*varargin*`)`

Construct a VSWF representation of a Mathieu beam. Inherits from `ott.beam.BscInfinite` and `ott.beam.properties.Mathieu` and `ott.beam.mixin.BesselBscCast`.

Properties

- theta – Annular angle [radians]
- morder – Mathieu beam mode number
- ellipticity – Ellipticity of Mathieu beam
- parity – Parity of beam ('even' or 'odd')

Webber

class `ott.beam.`**`Webber`**`(`*varargin*`)`

Construct a VSWF representation of a Bessel beam. Inherits from `ott.beam.BscInfinite` and `ott.beam.properties.Webber`.

Properties

- theta – Annular angle [radians]
- alpha – Parameter describe Webber beam

- parity – Parity of beam (either 'even' or 'odd')
- data – Internal BSC instance describing beam

Bessel

class `ott.beam.`**`Bessel`**`(`*varargin*`)`

Construct a VSWF representation of a Bessel beam. Inherits from `ott.beam.BscInfinite` and `ott.beam.properties.Bessel`.

See base classes for list of properties/methods.

Annular

class `ott.beam.`**`Annular`**`(`*varargin*`)`

Construct a VSWF representation of a finite Annular beam Inherits from `ott.beam.BscWBessel` and `ott.beam.properties.Polarisation` and `ott.beam.properties.Profile` and `ott.beam.properties.Lmode`.

Represents beams internally as an array of Bessel-like beams. Applies the beam weights when the data is requested.

Properties

- polbasis – Polarisation basis ('polar' or 'cartesian')
- polfield – Polarisation field (theta/phi or x/y)
- theta – Low and upper angles of the annular
- profile – Anular profile (function_handle)
- lmode – Orbital angular momentum number

Static methods

- InterpProfile – Generate a beam profile using interpolation
- BeamProfile – Generate a beam profile from another beam

PmParaxial

class `ott.beam.`**`PmParaxial`**`(`*varargin*`)`

Construct a beam using paraxial far-field point matching. Inherits from *`BscBeam`* and `properties.Mapping`, `properties.Polarisation` and `properties.Profile`.

Properties

- mapping – Paraxial to far-field mapping
- polbasis – (enum) Polarisation basis ('polar' or 'cartesian')
- polfield – (2 numeric) Field in theta/phi or x/y directions
- truncation_angle – Maximum angle for beam. Default: []
- data – Internal BSC data

Methods

- recalculate – Update the internal data for new Nmax

Static methods

- InterpProfile – Generate a beam profile using interpolation
- BeamProfile – Construct beam profile from another beam

Supported casts

- ott.bsc.Bsc – Construct bsc instance

4.1.4 Array types

Coherent

`ott.beam.Coherent`

Incoherent

`ott.beam.Incoherent`

4.2 *particle* Package

The *particle* package provides classes which represent OTT particles. `Particle` instances combine the optical scattering method and drag calculation method into a single representation. There are currently two specialisations of `Particle`, summarised in Fig. 4.2: `Fixed` stores the geometry, T-matrix and drag for a particle; `Variable` recalculates the T-matrix and drag whenever the refractive index of particle geometry changes.

Unlike `ott.tmatrix.Tmatrix` instances, `Particle` instances use SI units (i.e., the particle geometry and position uses units of meters).

In a future version this interface may change to support other scattering methods, and/or more optimal T-matrix calculation methods (such as only building required columns of the T-matrix depending on the incident beam).

Contents

4.2.1 Particle

class `ott.particle.Particle`

Base class for particles in optical tweezers simulations. Inherits from `ott.utils.RotationPositionProp` and *matlab.mixin.Hetrogeneous*.

This class combined the optical scattering methods, drag calculation methods and other properties required to simulate the dynamics of a particle in an optical tweezers simulation. In future version of OTT, this class may change to support multiple scattering methods.

This is an abstract class. For instances of this class see `Variable` and :class:Fixed`.

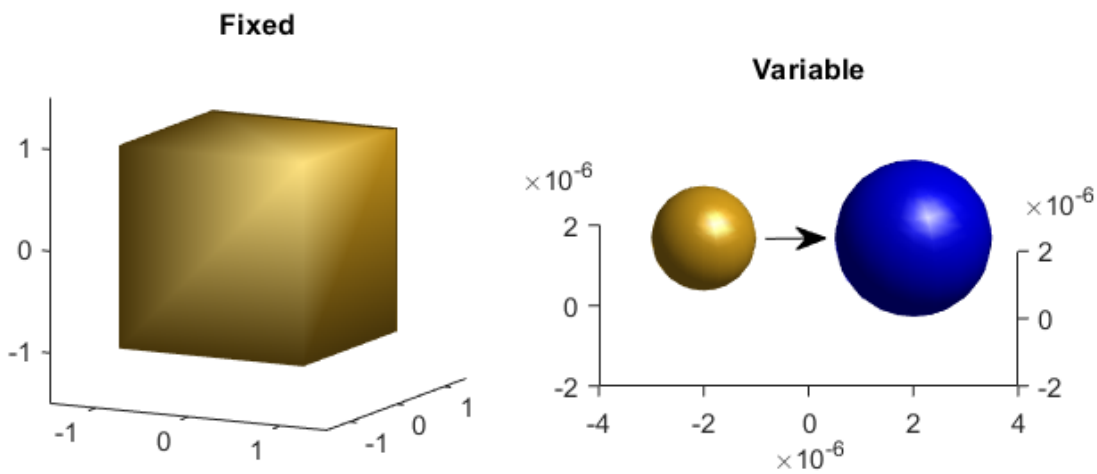

Fig. 4.2: Graphical display of the two different particle representations: fixed, which represents a constant particle; and variable, which represents a particle whose properties dynamicaly change.

Abstract properties

- shape – Geometric shape representing the particle
- tmatrix – Describes the scattering (particle-beam interaction)
- drag – Describes the drag (particle-fluid interaction)
- tinternal – T-matrix for internal scattered field (optional)
- mass – Particle mass [kg] (optional)
- moment – Particle moment of inertia [kg m^2] (optional)

Properties

- position – Particle position [m]
- rotation – Particle orientation (3x3 rotation matrix)

Methods

- surf – Uses the shape surf method to visualise the particle
- setMassFromDensity – Calcualte mass from homogeneous density

surf(*particle*, *varargin*)

Generate visualisation of the shape using the shape.surf method.

Applies the particle rotation and translation to the shape before visualisation.

Usage [...] = particle.surf(...)

For full details and usage, see `ott.shape.Shape.surf()`.

4.2.2 Fixed

class `ott.particle.`**`Fixed`**(*varargin*)

A particle with stored drag/tmatrix properties. Inherits from `ott.particle.Particle`.

Properties

- drag – Description of drag properties
- tmatrix – Description of optical scattering properties
- shape – Description of the geometry [m]
- tinternal – Internal T-matrix (optional)

Static methods

- FromShape – Construct a particle from a shape description

`Fixed`(*varargin*)

Construct a new particle instance with fixed properties

Usage particle = Fixed(shape, drag, tmatrix, ...)

Optional named arguments

- shape (ott.shape.Shape) – Geometry description. Units: m. Default: `[]`.
- drag (ott.drag.Stokes) – Drag tensor description. Default: `[]`.
- tmatrix (ott.tmatrix.Tmatrix) – Scattering description. Default: `[]`.
- tinternal (ott.tmatrix.Tmatrix) – Internal T-matrix. Default: `[]`.
- mass (numeric) – Particle mass. Default: `[]`.

static `FromShape`(*shape*, *varargin*)

Construct a particle from a shape description.

Usage particle = Fixed.FromShape(shape, ...)

Parameters

- shape (ott.shape.Shape) – Particle geometry. [m]

Named parameters

- index_relative (numeric) – Relative refractive index of particle. Default: `[]`. Must be supplied or computable from *index_particle* and *index_medium*.
- index_particle (numeric) – Refractive index in particle. Default: `[]`.
- index_medium (numeric) – Refractive index in medium. Default: `1.0` unless both *index_relative* and *index_particle* are supplied. In which case *index_medium* is ignored.
- wavelength0 (numeric) – Vacuum wavelength. [m] This parameter is required to calculate the T-matrix.
- viscosity (numeric) – Viscosity of medium. [Ns/m2] Default: `8.9e-4` (approximate viscosity of water).
- internal (logical) – If the internal T-matrix should also be computed. Default: `false`.
- mass (numeric) – Particle mass. Default: `[]`.

4.2.3 Variable

class `ott.particle.`**`Variable`**(*varargin*)

A particle whose drag/tmatrix are automatically recomputed. Inherits from `ott.particle.Particle`.

Changing the shape or refractive index of this particle causes the T-matrix and drag data to be re-calculated. This is useful for modelling particles with time-varying properties.

Properties

- shape – Shape describing the object (changes tmatrix/drag)
- index_relative – Particle refractive index (changes tmatrix)
- tmatrix_method – Method for T-matrix calculation
- tinternal_method – Internal T-matrix calculation methdo
- drag_method – Method for drag calculation

Dependent properties

- drag – Description of drag properties
- tmatrix – Description of optical scattering properties
- tinternal – Internal T-matrix (optional)

Methods

- setProperties – Set shape and index_relative simultaneously

Static methods

- FromShape – Create instance using FromShape of Tmatirx/Stokes
- Sphere – Construct instance using Sphere approximation
- StarShaped – Construct instance for star shaped particles

static **`FromShape`**(*varargin*)

Construct a new Variable particle using the FromShape method.

This function sets up the `tmatrix_method`, `tinternal_method` and `drag_method` functions to use the `FromShape` methods from `ott.tmatrix.Tmatrix` and `ott.drag.Stokes`.

Usage particle = ott.particle.Variable.FromShape(...)

Optional named arguments

- wavelength_medium (numeric) – Medium wavelength. [m] Used for converting shape units [m] to wavelength units. Default: `1064e-9` (a common IR trapping wavelength).
- viscosity (numeric) – Viscosity of medium. [Ns/m2] Default: `8.9e-4` (approximate viscosity of water).
- internal (logical) – If the internal T-matrix calculation method should also be set. Default: `false`.

Unmatched parameters passed to class constructor.

static **`Sphere`**(*varargin*)

Construct a variable particle for a spherical geometry

Uses the `FromShape` methods from `ott.tmatrix.Mie` and `ott.drag.StokesSphere`.

Usage particle = ott.particle.Variable.Sphere(...)

Optional named arguments

- wavelength_medium (numeric) – Medium wavelength. [m] Used for converting shape units [m] to wavelength units. Default: `1064e-9` (a common IR trapping wavelength).
- viscosity (numeric) – Viscosity of medium. [Ns/m2] Default: `8.9e-4` (approximate viscosity of water).
- internal (logical) – If the internal T-matrix calculation method should also be set. Default: `false`.

Unmatched parameters passed to class constructor.

static StarShaped(*varargin*)

Construct a variable particle for a star shaped geometry.

Uses the `FromShape` methods from `ott.tmatrix.Tmatrix` and `ott.drag.StokesStarShaped`.

Usage particle = ott.particle.Variable.StarShaped(...)

Optional named arguments

- wavelength_medium (numeric) – Medium wavelength. [m] Used for converting shape units [m] to wavelength units. Default: `1064e-9` (a common IR trapping wavelength).
- viscosity (numeric) – Viscosity of medium. [Ns/m2] Default: `8.9e-4` (approximate viscosity of water).
- internal (logical) – If the internal T-matrix calculation method should also be set. Default: `false`.

Unmatched parameters passed to class constructor.

Variable(*varargin*)

Construct a new Variable particle instance.

Usage particle = Variable(...)

Optional named arguments

- tmatrix_method ([] | function_handle) – T-matrix calculation method. Signature: `@(shape, ri, old_tmatrix)`. Default: `[]`.
- drag_method ([] | function_handle) – Drag calculation method. Signature: `@(shape, old_drag)`. Default: `[]`.
- tinternal_method ([] | function_handle) – internal T-matrix method. Signature: `@(shape, ri, old_tmatrix)`. Default: `[]`.
- initial_shape ([] | ott.shape.Shape) – Initial particle geometry. Default: `[]`.
- initial_index_relative (numeric) – Initial particle relative refractive index. Default: `[]`.

4.3 *bsc* Package

The *bsc* package provides classes for representing beams in a vector spherical wave function (VSWF) basis. The base class for all VSWF beams is *`Bsc`*, which provides methods for visualising beams, calculating forces, and manipulating the VSWF data. Similar to T-matrices, the data is stored internally as a vector, allowing the type to be changed to any valid Matlab matrix type (such as `gpuArray` or `sparse`).

Simple multipole beams can be created by directly specifying their beam shape coefficients. For more complex beams, the *`Pointmatch`*, *`PlaneWave`* and *`Annular`* classes provide provide methods to calculate BSC data. Additionally, the *`PlaneWave`* and *`Annular`* beams provide alternative translation methods that are more optimal for these types of beams. These methods are summarised in Fig. 4.3.

Unlike the previous version of the toolbox, the *Bsc* class does not track the position/rotation of the beam. This is to avoid suggesting that the beam actually has a well defined position/rotation. Other notable differences include the use of Hetrogeneous arrays, support for arbitrary beam data data-types, and moving user friendly features such as nice units to `ott.beam.Beam`.

Bsc rotations have units of radians and displacements have units of medium wavelength. For a more user friendly interface with SI units, see `ott.beam.Beam`.

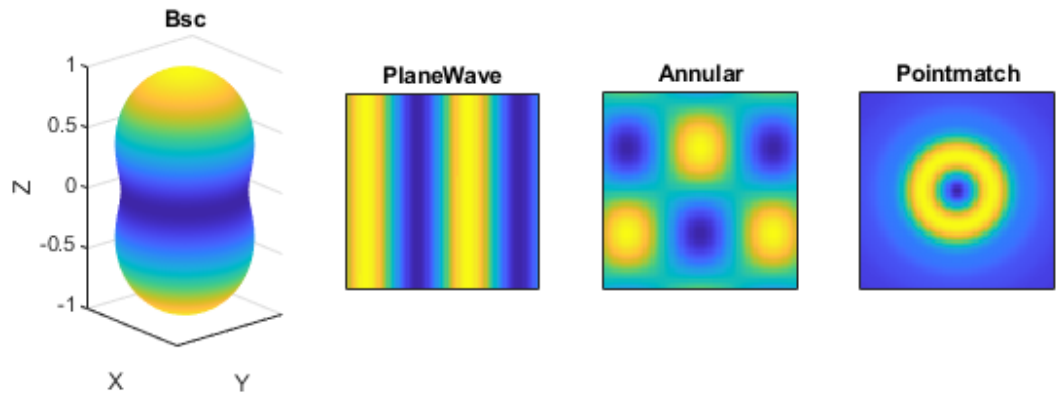

Fig. 4.3: Graphical display showing the different BSC classes currently included in the T-matrix. The Bsc class is the base class and can be used directly for simple multipole mode descriptions. The other classes provide specialisations and methods for approximating more complex beams.

Contents

4.3.1 Bsc

class `ott.bsc.Bsc`(*oa*, *ob*)

Class representing vector spherical wave function beam shape coefficients. Inherits from `ott.utils.RotateHelper` and `ott.utils.TranslateHelper`.

Unlike the previous version of the toolbox, the *Bsc* class does not track the position/rotation of the beam. This is to avoid suggesting that the beam actually has a well defined position/rotation. Other notable differences include the use of Hetrogeneous arrays, support for arbitrary beam data data-types, and moving user friendly features such as nice units to `ott.beam.Beam`.

Bsc rotations have units of radians and displacements have units of medium wavelength.

Properties

- a – Beam shape coefficients *a* vector
- b – Beam shape coefficients *b* vector
- Nmax – (Dependent) Truncation number for VSWF coefficients

- power – (Dependent) Power of the beam shape coefficients

Static methods

- FromDenseBeamVectors – Construct beam from dense beam vectors.
- BasisSet – Generate basis set of VSWF beams
- PmNearfield – Construct using near-field point matching
- PmFarfield – Construct using far-field point matching

Methods

- Bsc – Class constructor
- nbeams – Get the total number of beams in array
- issparse – Check if the beam data is sparse
- full – Make the beam data full
- sparse – Make the beam data sparse
- makeSparse – Make the beam data sparse (with additional options)
- setNmax – Resize beam data to desired Nmax
- shrinkNmax – Reduce Nmax while preserving beam power
- gpuArray – Make beam a gpuArray
- gather – Apply gather to beam data
- rotate* – (Inherited) Functions for rotating the beam
- translate* – (Inherited) Functions for translating the beam
- translateZ – Translation along the theta=0 (z) axis
- getCoefficients – Get a/b vectors with additional options
- setCoefficients – Set a/b vectors with additional options

Mathematical operations

- sum – Combine array of beams using summation
- times – Scalar multiplication of beam vectors
- mtimes – Scalar and matrix multiplication of beam vectors
- safeTimes – Matrix multiplication with support for shrinking
- rdivide – Scalar division of beam vectors
- mrdivide – Scalar division of beam vectors
- uminus – Negation of beam vectors
- minus – Subtraction of beam vectors
- plus – Addition of beam vectors
- real – Extract real part of BSC data
- imag – Extract imag part of BSC data
- abs – Calculate absolute value of BSC data

Field calculation methods

- efieldRtp – Calculate electric field around the origin
- hfieldRtp – Calculate magnetic field around the origin
- efarfield – Calculate electric fields in the far-field

Force and torque related methods

- force – Calculate the change in momentum between two beams
- torque – Calculate change in angular momentum between beams
- spin – Calculate change in spin momentum between beams

Casts

- ott.bsc.Bsc – Downcast BSC superclass to base class
- ott.tmatrix.Tmatrix – Create T-matrix from beam array

4.3.2 Pointmatch

`ott.bsc.`**`Pointmatch`**

4.3.3 PlaneWave

class `ott.bsc.`**`PlaneWave`**(*varargin*)

Bsc specialisation for plane waves

Plane waves can be translated in any direction by simply applying a phase shift to the beam. This class provides overloads for the beam translation functions implementing this optimisation.

Properties

- direction – Propagation direction of plane wave

Static methods

- FromDirection – Construct a beam for the specified direction

4.3.4 Annular

class `ott.bsc.`**`Annular`**(*varargin*)

Bsc specialisation for Bessel-like beams

Bessel beams and other annular beams can be translated axially with only a phase shift applied to the beam shape coefficients. This class overloads the axial translation function to implement this.

Properties

- theta – Angle describing annular

Static methods

- FromBessel – Construct Annular beam from Bessel specification.

4.4 *tmatrix* Package

The *tmatrix* package provides methods for calculating how particles scatter light via the T-matrix method. The main class in this package is `Tmatrix` which provides methods for manipulating the T-matrix data. The data is stored internally as a matrix, allowing the type to be changed to any valid Matlab matrix type (such as `gpuArray` or `sparse`).

Other classes in this package provide methods for calculating T-matrices using different approximations, these methods are summarised in Fig. 4.4. Most of these classes have a `FromShape` method which can be used to calculate the T-matrix from a geometric shape description. Once the T-matrix is calculated, it can be multiplied by a `ott.bsc.Bsc` object to calculate the scattered or internal fields.

The package also contains two sub-packages: `smatries` contains components of SMARTIES which are used in `Smarties`; and the `dda` sub-package contains components of a discrete dipole approximation implementation that will eventually move into OTTv2 (currently used only by `Dda`).

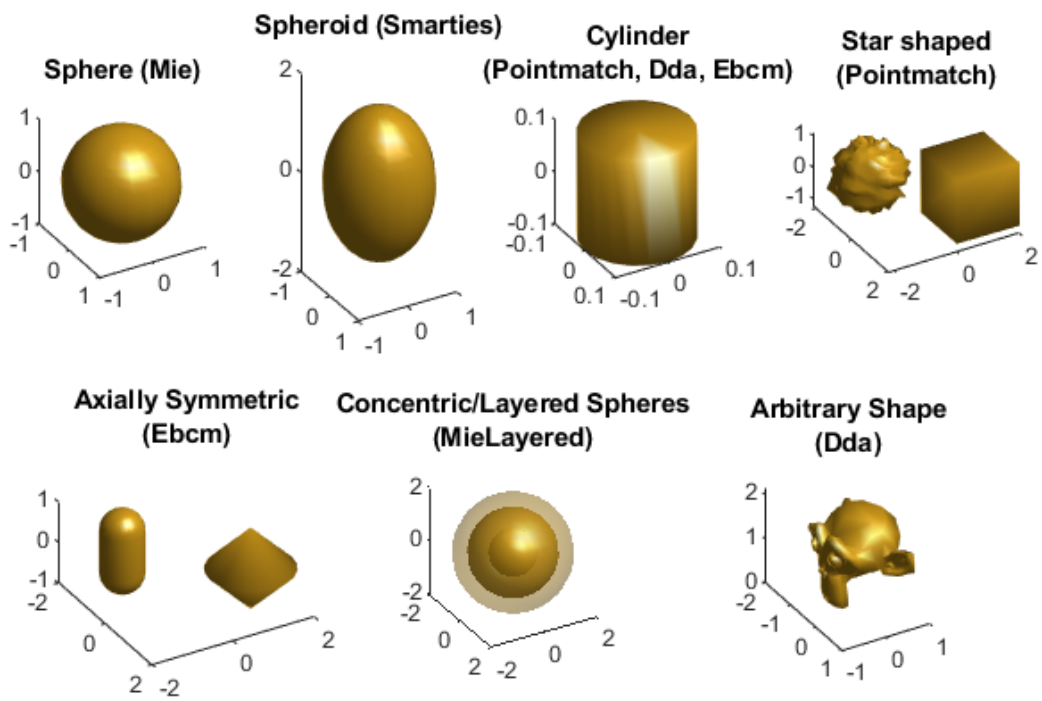

Fig. 4.4: Graphical display of different particle shapes and the corresponding T-matrix methods that might be suitable for modelling them. The accuracy of the resulting T-matrix often depends on the size and aspect ratio of the particle as well as the choosen method.

Contents

4.4.1 Tmatrix

class `ott.tmatrix.`**`Tmatrix`**`(`*varargin*`)`

Class representing 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 objects, 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`.

Properties

- data – The T-matrix this class encapsulates
- type – Type of T-matrix (total, scattered or internal)
- Nmax – Size of the T-matrix data (number of multipoles)
- total – Total-field instance of the T-matrix
- scattered – Scattered-field instance of the T-matrix

Methods

- Tmatrix – Class constructor
- issparse – Returns true if the internal data is sparse
- full – Convert internal data to full
- sparse – Convert internal data to sparse
- makeSparse – Make the data sparse (with additional options)
- setNmax – Resize data to desired Nmax
- shrinkNmax – Reduce Nmax while preserving column/row power
- gpuArray – Make T-matrix a gpuArray
- gather – Apply gather to T-matrix data
- setType – Set the T-matrix type property (doesn't change data)
- columnCheck – Calculate and check T-matrix column power
- mergeCols – Merge columns of tmatrices

Mathematical and matrix operations

- times – Scalar multiplication of data

- mtimes – Scalar and matrix multiplication of data
- rdivide – Scalar division of data
- mrdivide – Scalar division of data
- uminus – Negation of data
- minus – Subtraction of data
- plus – Addition of data
- real – Extract real part of T-matrix
- imag – Extract imaginary part of T-matrix
- diag – Extract the diagonal of the T-matrix

Static methods

- FromShape – Take a guess at a suitable T-matrix method
- SmartCylinder – Smart method selection for cylindrical particles

Casts

- ott.bsc.Bsc – Convert each T-matrix column to beam vector
- ott.tmatrix.Tmatrix – Downcast T-matrix superclass to base class

See also *`Tmatrix()`* and `ott.tmatrix.Mie`.

`static FromShape`(*shape*, *varargin*)

Take a guess at a suitable T-matrix method.

The T-matrix can only be calculated easily and accurately for a few very specific cases; as such, this function defaults to DDA for most particles which may result in very slow calculations (or failures due to memory limitations). The resulting T-matrices may not be accurate and it is recommended to inspect the fields and compare results with another method.

This method does the following

- spheres – Uses *`Mie`*.
- spheroids – Uses *`Smarties`*
- cylinders – Uses `SmartCylinder()`. This method may also work well for other semi-elongated rotationally symmetry shapes.
- rotationally symmetric – Uses *`Ebcm`*.
- star shaped – Uses *`Pointmatch`*.
- otherwise – Uses *`Dda`*.

For many types of particles it would be better to use another method (such as geometric optics for very large particles). This method may change in future releases when other methods are added or when limits of existing methods are further explored.

Usage tmatrix = ott.tmatrix.Tmatrix.FromShape(shape, index_relative)

Parameters

- shape (ott.shape.Shape) – Shape to generate T-matrix for. Shape dimensions should be in units of wavelength.
- index_relative (numeric) – Relative refractive index.
- internal (logical) – If the T-matrix should be an internal T-matrix. Default: `false`.

Tmatrix(*varargin*)

Construct a new T-matrix object.

Usage tmatrix = Tmatrix(...) New empty T-matrix. Leaves the data uninitialised.

tmatrix = Tmatrix(data, ...) Initializes the data with the matrix *data*.

Parameters

- data (NxM numeric | cell) – The T-matrix data. Typically a sparse or full matrix. Data must be empty or valid T-matrix size. If cell, describes T-matrix array and elements must be a cell array of NxM matrices.

Optional named arguments

- type (enum) – Type of T-matrix. Must be 'internal', 'scattered' or 'total'. Default: `'scattered'`.

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

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

columnCheck(*tmatrix*, *varargin*)

Check the power in each column

For a non-absorbing total-field T-matrix, the power in each column should add up to unity (power should be conserved).

Usage tmatrix.columnCheck(...) Raises a warning if the power drops bellow a threshold.

column_power = tmatrix.columnCheck(...) Returns the power in each column.

Optional named arguments

- threshold (numeric) – Threshold for power loss warnings/errors. Default: `1.0e-3`.
- action (enum) – Action to take if power lost/gained. Can be 'warn', 'error' or 'none'. If no outputs present, the default behaviour is 'warn'. Otherwise 'none'.

diag(*tmatrix*, *k*)

Extract the T-matrix diagonal

Usage d = diag(tmatrix) Returns the diagonal in vector format

d = diag(tmatrix, K) Extracts the K-th diagonal of the T-matrix.

[dA, dB] = diag(...) Returns the diagonal of the upper and lower parts separately.

full(*tmatrix*)

Convert the data to a full matrix

Usage tmatrix = full(tmatrix)

gather(*tmatrix*)

Apply *gather* to data.

If the data is a gpuArray, returns a copy of the data in the local workspace with data transferred from the GPU.

Usage tmatrix = gather(tmatrix)

gpuArray(*tmatrix*)

Copies the tmatrix data to the GPU

Usage tmatrix = gpuArray(tmatrix)

imag(*tmatrix*)

Extract imaginary part of T-matrix

Usage tmatrix = imag(tmatrix);

issparse(*tmatrix*)

Returns true if the data is sparse

Usage b = issparse(tmatrix)

makeSparse(*tmatrix*, *varargin*)

Make the T-matrix data sparse by removing near-zero power elements

Treats each column as a beam shape vector and applies `ott.bsc.Bsc.makeSparse()` to each column.

Usage tmatrix = tmatrix.makeSparse(...)

Optional named arguments

- AbsTol (numeric) – Absolute tolerance for removing elements. Default: `[]`.
- RelTol (numeric) – Relative tolerance for removing elements. Power is relative to power in each column. Default: `1.0e-15`.

If both AbsTol and RelTol are specified, only elements satisfying both conditions are kept.

mergeCols(*tmatrix*, *tmatrix2*, *ci*)

Merge columns of two T-matrices

Usage tmatrix = tmatrix.mergeCols(tmatrix2, ci) Keeps all cols of the first T-matrix except thouse replaced by tmatrix2 (specified by ci).

Parameters

- tmatrix1, tmatrix2 – T-matrices to merge
- ci – (N numeric) Combined indices of T-matrix columns to replace. For each `ci` entry, keeps 2 columns from `tmatrix2` (i.e., both TE and TM modes).

minus(*a*, *b*)

Minus operation on tmatrix

Usage tmatrix = tmatrix1 - tmatrix2

tmatrix = other - tmatrix tmatrix = tmatrix - other

Note: Uses the uminus operation of the second argument.

mtimes(*a*, *b*)

Matrix and scalar multiplication of T-matrix data

Usage beam = tmatrix * beam Calculate how a beam is scattered by the T-matrix, increase beam or T-matrix Nmax if required.

tmatrix = tmatrix1 * tmatrix2 Multiply T-matrix data, increasing Nmax if required.

vector = vector * tmatrix vector = tmatrix * vector

tmatrix = scalar * tmatrix tmatrix = tmatrix * scalar

plus(*a*, *b*)

Apply plus operation on T-matrix data

Usage tmatrix = tmatrix1 + tmatrix2;

tmatrix = other + tmatrix; tmatrix = tmatrix + other;

rdivide(*tmatrix*, *o*)

Scalar division of T-matrix data

Usage tmatrix = tmatrix ./ scalar

real(*tmatrix*)

Extract real part of T-matrix

Usage tmatrix = real(tmatrix);

setNmax(*tmatrix*, *nmax*, *varargin*)

Resize the T-matrix, with additional options

Usage tmarix = tmatrix.setNmax(nmax, ...) or tmatrix.Nmax = nmax Set the Nmax, a optional warning is issued if truncation occurs.

Parameters

- Nmax (1 | 2 numeric) – Nmax for both dimensions or vector with Nmax for *[rows, cols]*.

Optional named arguments

- AbsTol (numeric) – Absolute tolerance for removing elements. Default: `[]`.
- RelTol (numeric) – Relative tolerance for removing rows. Power is relative to power in each column. Default: `1.0e-15`.
- ColTol (numeric) – Absolute tolerance for removing columns. Default: `1.0e-15`.
- powerloss (enum) – Action to take when column power is lost. Can be one of 'ignore', 'warn' or 'error'. Default: `'warn'`.

setType(*tmatrix*, *val*)

Set the T-matrix type paramter (without raising a warning)

Usage tmatrix = tmatrix.setType(val);

shrinkNmax(*tmatrix*, *varargin*)

Shrink the size of the T-matrix while preserving power

Converts to a scattered or internal T-matrix and then removes columns with no significant power and rows by passing each column to `ott.bsc.Bsc.shrinkNmax()`.

Usage tmatrix = tmatrix.shrinkNmax(...)

Optional named arguments

- AbsTol (numeric) – Absolute tolerance for removing elements. Default: `[]`.
- RelTol (numeric) – Relative tolerance for removing rows. Power is relative to power in each column. Default: `1.0e-15`.
- ColTol (numeric) – Absolute tolerance for removing columns. Default: `1.0e-15`.

sparse(*tmatrix*)

Convert the data to a sparse matrix

This function doesn't change the data. For a method that removes near-zeros elements, see `makeSparse()`.

Usage tmatrix = sparse(tmatrix)

times(*a*, *b*)

Element-wise multiplication of T-matrix data

Usage tmatrix = tmatrix1 .* tmatrix2;

tmatrix = other .* tmatrix; tmatrix = tmatrix .* other;

4.4.2 Dda

class ott.tmatrix.**Dda**(*dda*, *varargin*)

Construct a T-matrix using the discrete dipole approximation. Inherits from `ott.tmatrix.Tmatrix`.

This method calculates how each VSWF mode is scattered and then uses point-matching to calculate each column of the T-matrix. The field calculations are done using the discrete dipole approximation, but the same approach could be used for T-matrix calculation via any other field calculation method.

The current DDA implementation requires a lot of memory. 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). The aim of the next OTT release is to include geometric optics and finite difference time domain as alternative methods for force calculation with these larger particles.

Properties

- dda – DDA instance used for field calculation
- pmrtp – (3xN numeric) Locations for point matching

Static methods

- FromShape – Construct from a geometric shape
- DefaultPmrtp – Build default `pmrtp` locatiosn for point matching
- DefaultProgressCallback – Default progress callback for method

See also *Dda()*, `ott.tmatrix.dda`.

Dda(*dda*, *varargin*)

Construct a T-matrix using a DDA simulation for field calculation

Usage tmatrix = Dda(dda, . . .)

[tmatrix, incData, pmData] = Dda(dda, . . .) Also returns the VSWF data structures used for calculating the incident field and the point matching field.

Parameters

- dda – (ott.tmatrix.dda.Dda instance) The DDA instance used for field calculations.

Optional named arguments

- Nmax – (numeric) Size of the VSWF expansion used for the T-matrix point matching (determines T-matrix rows). Default: `ott.utils.ka2nmax(2*pi*shape.maxRadius)` (may need different values to give convergence for some shapes).
- ci – (N numeric) Number of modes to calculate scattering for. This determines number of T-matrix columns. Default: `1:ott.utils.combined_index(Nmax, Nmax)` (all modes).
- pmrtp – (3xN numeric) Coordinatse for point matching. Radial coordinate must all be finite or all be Inf. Default uses `ott.tmatrix.Dda.DefaultPmrtp`.

- incData (ott.utils.VswfData) – Data structure for repeated incident field calculations. Default: `ott.utils.VswfData()`.
- pmData (ott.utils.VswfData) – Data structure for repeated point matching field calculations. Default: `ott.utils.VswfData()`.

Unmatched parameters passed to `calculate_columns()`.

static DefaultPmrtp(*Nmax*, *varargin*)

Build default grid of points for point matching

Usage pmrtp = DefaultPmrtp(Nmax, ...)

Parameters

- Nmax – (numeric) Row Nmax for generated T-matrix.

Optional named parameters

- radius – (numeric) Radius for point matching locations. Must be positive scalar or Inf for far-field point matching. Default: `Inf`.
- angulargrid – ({theta, phi}) Angular grid of points for calculation of radii. Default is equally spaced angles with the number of points determined by Nmax.
- xySymmetry – (logical) If the generated grid should be for mirror symmetry DDA. Default: `false`.
- zRotSymmetry – (numeric) If the generated grid should be for rotationally symmetric DDA. Default: `1`.

static DefaultProgressCallback(*data*)

Default progress callback for Dda

Prints the progress to the terminal.

Usage DefaultProgressCallback(data)

Parameters

- data (struct) – Structure with two fields: index and total.

static FromShape(*shape*, *varargin*)

Construct a T-matrix from a geometric shape

Usage tmatrix = ott.tmatrix.Dda.FromShape(shape, ...)

[tmatrix, incData, pmData] = ott.tmatrix.Dda.FromShape(...) Returns the field calculation data for repeated calculations.

Optional named arguments

- spacing – (numeric) – Dipole spacing in wavelength units. Default: `1/20`.
- polarizability – (function_handle | 3x3 numeric) Method to calculate polarizability or 3x3 tensor for homogeneous material. Default: `@(xyz, spacing, ri) polarizability.LDR(spacing, ri)`
- index_relative – (function_handle | numeric) Method to calculate relative refractive index or homogeneous value. Ignored if polarizability is a 3x3 tensor.
- low_memory – (logical) If we should use the low memory DDA implementation. Default: `false`.

Additional parameters passed to class constructor.

4.4.3 Ebcm

class `ott.tmatrix.Ebcm`(*varargin*)

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

Implements the extended boundary conditions methods for rotationally symmetric homogeneous particles.

Properties

- points – (2xN numeric) Surface points [r; theta]
- normals – (2xN numeric) Normals at points [nr; ntheta]
- areas – (1xN numeric) Conic section surface areas
- index_relative – (numeric) Relative refractive index of particle.
- xySymmetry – (logical) True if using xy-mirror optimisations
- invMethod – Inversion method used for T-matrix calculation

Static methods

- FromShape – Construct from geometric shape object.

Additional methods/properties inherited from *`Tmatrix`*.

This class is based on `tmatrix_ebcm_axisym.m` from OTTv1.

`Ebcm`(*varargin*)

Calculates T-matrix using extended boundary condition method

Usage tmatrix = Ebcm(points, normals, area, index_relative, ...) Calculate external T-matrix (unless *internal* is true)

[external, internal, data] = Ebcm(...) Calculate both the internal and external T-matrices, and return the VswfData structure used during calculations.

Parameters

- points (2xN numeric) – Coordinates describing surface. Spherical coordinates (omitting azimuthal angle: *[r; theta]*).
- normals (2xN numeric) – Normals at surface points.
- areas (N numeric) – Area elements at surface points.
- index_relative (numeric) – Particle relative refractive index.

Optional named parameters

- xySymmetry (logical) – If calculation should use xy-mirror symmetry optimisations. Default: `false`. Doesn't check points describe valid mirror symmetric shape.
- invMethod (enum|function handle) – Inversion method for T-matrix calculation. Currently supported methods are 'forwardslash', 'backslash', 'inv', 'pinv'. A custom inversion method can be specified using a function handle with the format `inv_method(RgQ, Q)` which returns the T-matrix data. Default: `forwardslash`. Only used for external T-matrix calculation, may change in future.
- internal (logical) – If true, the returned T-matrix is an internal T-matrix. Ignored for two outputs. Default: `false`.
- Nmax (numeric) – Size of the VSWF expansion used for the T-matrix calculation. In some cases it can be reduced after construction. Default: `ott.utis.ka2nmax(2*pi*shape.maxRadius)` (may need different values to give convergence for some shapes).

- verbose (logical) – If true, outputs the condition number of the Q and RgQ matrices. Default: `false`.
- data (ott.utils.VswfData) – Field data for repeated field calculation. Default is an empty VswfData structure.

static FromShape(*varargin*)

Construct a T-matrix using EBCM from a shape object.

If the shape object is not of type `ott.shape.AxisymLerp`, first casts the shape to this type using the default cast (if supported, otherwise raises an error).

Usage tmatrix = Ebcm.FromAxisymInterpShape(shape, index_relative, ...) Calculate external T-matrix (unless *internal* is true)

[external, internal] = Ebcm.FromAxisymInterpShape(...) Calculate both the internal and external T-matrices.

Parameters

- shape (ott.shape.Shape) – Description of shape geometry. Object must be a AxisymInterp or be castable to AxisymInterp.
- index_relative (numeric) – Particle relative refractive index.

See *Ebcm()* for additional named parameters.

4.4.4 Mie

class ott.tmatrix.**Mie**(*varargin*)

Construct T-matrix with Mie scattering coefficients. Inherits from `ott.tmatrix.Homogeneous`.

The Mie coefficients describe the scattering of a sphere. They can also be used to give a reasonable estimate of the force for non-spherical particles when no other suitable method is available.

This class supports both homogeneous dielectric (conductive and non-conductive) and magnetic isotropic materials. For other materials, consider *Dda* or *MieLayered* (for layered spheres).

Properties

- radius – Radius of sphere
- index_relative – Relative refractive index
- relative_permeability – Relative permeability

Static methods

- FromShape – Uses ShapeMaxRadius but raises a warning
- ShapeVolume – Construct with radius set from particle volume
- ShapeMaxRadius – Construct with radius set from particle max radius

See base class for additional methods/properties.

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

static FromShape(*shape*, *varargin*)

Construct a T-matrix from a shape object.

Uses *ShapeMaxRadius()* but raises a warning if the shape isn't a sphere.

Usage tmatrix = Mie.FromShape(shape, ...)

Parameters

- shape (ott.shape.Shape) – The shape input.

All other parameters passed to constructor.

`Mie`(*varargin*)

Construct a new Mie T-matrix for a sphere.

Usage tmatirx = Mie(radius, index_relative, ...) Calculate the external T-matrix (unless *internal = true*).

[external, internal] = Mie(radius, index_relative, ...) Calculate both the internal and external T-matrices.

Parameters

- radius (numeric) – Radius of the sphere (in wavelength units).
- index_relative (numeric) – The relative refractive index of the particle compared to the surrounding medium.

Optional named parameters

- relative_permeability (numeric) – Relative permeability of the particle compared to surrounding medium. Default: `1.0`.
- Nmax (numeric) – Size of the VSWF expansion used for the T-matrix calculation. Default: `ott.utis.ka2nmax(2*pi*radius)` (external) or `ott.utis.ka2nmax(2*pi*radius*index_relative)` (internal).
- internal (logical) – If true, the returned T-matrix is an internal T-matrix. Ignored for two outputs. Default: `false`.

`static ShapeMaxRadius`(*shape*, *varargin*)

Construct Mie T-matrix with radius from shape max radius

Usage tmatrix = Mie.ShapeMaxRadius(shape, ...)

Parameters

- shape (ott.shape.Shape) – The shape input.

All other parameters passed to constructor.

`static ShapeVolume`(*shape*, *varargin*)

Construct Mie T-matrix with radius from shape volume

Usage tmatrix = Mie.ShapeVolume(shape, ...)

Parameters

- shape (ott.shape.Shape) – The shape input.

All other parameters passed to constructor.

4.4.5 MieLayered

class `ott.tmatrix.MieLayered`(*varargin*)

Construct T-matrices for a layered sphere. Inherits from `ott.tmatrix.Tmatrix`.

This class implements the first part of

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

and can be used to model layered spherical particles.

Properties

- radii – Radii of sphere layers (inside to outside)
- relative_indices – Relative refractive indices (inside to outside)

Static methods

- FromShape – Construct layered sphere from array of shapes

See base class for additional methods/properties.

This class is based on tmatrix_mie_layered.m from ottv1.

static FromShape(*shape*, *relative_indices*, *varargin*)

Construct a T-matrix from a shape array

Uses the maxRadius property of each shape in the shape array for the sphere radii.

Shapes radii must be in ascending order. If the shapes aren't all centred, raises a warning.

Usage tmatrix = MieLayered.FromShape(shape, relative_indices, ...)

All other parameters are passed to MieLayered.

MieLayered(*varargin*)

Construct a new Mie layered T-matrix

Usage tmatirx = MieLayered(radii, relativeMedium, ...) Calculate the external T-matrix (unless *internal = true*).

[external, internal] = Mie(radii, relativeMedium, ...) Calculate both the internal and external T-matrices.

Parameters

- radii (numeric) – Radius of sphere layers (in relative units unless optional parameter *wavelength* is specified). Radii should be in ascending order.
- relativeMedium (ott.beam.medium.RelativeMedium) – The relative medium describing the particle material layers. Particle must be isotropic homogeneous magnetic or dielectric. The number of layers should match number of radii.

Optional named parameters

- wavelength (numeric) – Used to convert radius input to relative units, i.e. *radius_rel = radius ./ wavelength*. This parameter not used for setting the T-matrix material. Default: `1.0` (i.e., radius is already in relative units).
- Nmax (numeric) – Size of the VSWF expansion used for the T-matrix calculation. Can be reduced after construction. Default: `100` (should be numerically stable).
- internal (logical) – If true, the returned T-matrix is an internal T-matrix. Ignored for two outputs. Default: `false`.

4.4.6 Pointmatch

class `ott.tmatrix.`**`Pointmatch`**(*varargin*)

Constructs a T-matrix using the point matching method. Inherits from `ott.tmatrix.Homogeneous`.

The point matching method is described in

> T. A. Nieminen, H. Rubinsztein-Dunlop, N. R. Heckenberg JQSRT 79-80, 1019-1029 (2003), 10.1016/S0022-4073(02)00336-9

The method can be used to construct T-matrices for star shaped particles with aspect ratios close to unity. Supports homogeneous isotropic materials. This implementation includes symmetry optimisations for rotationally symmetric and mirror symmetric particles.

Properties

- index_relative – Relative refractive index of particle
- rtp – Locations used for point matching
- nrtp – Surface normals at rtp-locations (spherical coords.)
- zRotSymmetry – Z-axis rotational symmetry (1 = no symmetry)
- xySymmetry – XY mirror symmetry (logical)

Static methods

- DefaultProgressCallback – Default progress call-back method
- FromShape – Construct T-matrix from star shape

This class is based on tmatrix_pm.m from ottv1.

static **`DefaultProgressCallback`**(*data*)

Default progress callback for Pointmatch

Prints the progress to the terminal.

Usage DefaultProgressCallback(data)

Parameters

- data (struct) – Structure with three fields: stage (either 'setup' or 'inv'), iteration (numeric), and total (numeric).

static **`FromShape`**(*shape*, *varargin*)

Construct T-matrix using point matching from a star shaped object

Usage tmatrix = Pointmatch.FromShape(shape, index_relative, ...) Calculate external T-matrix.

[external, internal] = Pointmatch.FromShape(...) Calculate external and internal T-matrices.

Parameters

- shape (ott.shape.Shape) – A star-shaped object describing the geometry (must have a valid `starRadii` method).
- index_relative (numeric) – Relative refractive index.

Optional named parameters

- Nmax (numeric) – Size of the VSWF expansion used for the T-matrix calculation. In some cases it can be reduced after construction. Default: `ott.utils.ka2nmax(2*pi*shape.maxRadius)` (may need different values to give convergence for some shapes).

- angulargrid ({theta, phi}) – Angular grid of points for calculation of radii. Default is equally spaced angles with the number of points determined by Nmax.

Additional parameters are passed to the class constructor.

Pointmatch(*varargin*)

Calculates T-matrix using the point matching method.

Usage tmatrix = Pointmatch(rtp, nrtp, index_relative, ...) Calculate external T-matrix.

[external, internal] = Pointmatch(rtp, nrtp, index_relative, ...) Calculate external and internal T-matrices.

Parameters

- rtp (3xN numeric) – Locations of surface points to point-match.
- nrtp (3xN numeric) – Normals at surface locations. Spherical coordinates.
- index_relative (numeric) – Relative refractive index.

Optional named parameters

- Nmax (numeric) – Size of the VSWF expansion used for the T-matrix calculation. In some cases it can be reduced after construction. Default: `ott.utis.ka2nmax(2*pi*max(rtp(1, :)))` (may need different values to give convergence for some shapes).
- zRotSymmetry (numeric) – Degree of rotational symmetry about the z-axis. Default: `1` (no symmetry).
- xySymmetry (logical) – If the particle is mirror symmetry about the xy-plane. Default: `false`.
- internal (logical) – If true, the returned T-matrix is an internal T-matrix. Ignored for two outputs. Default: `false`.
- progress (function_handle) – Function to call for progress updates during method evaluation. Takes one argument, see *`DefaultProgressCallback()`* for more information. Default: `[]` (for Nmax < 20) and `@DefaultProgressCallback` (otherwise).

4.4.7 Smarties

class `ott.tmatrix.`**Smarties**(*varargin*)

Constructs a T-matrix using SMARTIES. Inherits from `ott.tmatrix.Homogeneous`.

SMARTIES is a method for calculating T-matrices for spheroids, full details can be found in

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

SMARTIES is distributed with a Creative Commons Attribution-NonCommercial 4.0 International License. Copyright 2015 Walter Somerville, Baptiste Auguié, and Eric Le Ru. This version of OTT includes a minimal version of SMARTIES for T-matrix calculation, if you use the SMARTIES code, please cite the above paper.

Properties

- ordinary – Ordinary radius
- extraordinary – Extra-ordinary radius
- index_relative – Relative refractive index of particle

Static methods

- FromShape – Construct a T-matrix from a shape description

See also Smarties, `ott.tmatrix.Mie` and `ott.tmatrix.smarties`.

static FromShape(*shape*, *varargin*)

Construct a T-matrix using SMARTIES/EBCM for spheroids.

Usage tmatrix = Smarties.FromShape(shape, index_relative, ...)

[texternal, tinternal] = Smarties.FromShape(...)

Parameters

- shape (ott.shape.Shape) – A spheroid with the extraordinary axis aligned to the z-axis.
- index_relative (numeric) – The relative refractive index of the particle compared to the surrounding medium.

All other parameters passed to class constructor.

Smarties(*varargin*)

Construct a T-matrix using SMARTIES/EBCM for spheroids.

Usage tmatrix = Smarties(ordinary, extraordinary, index_relative...)

[texternal, tinternal] = Smarties(...)

Parameters

- ordinary (numeric) – Ordinary radius.
- extraordinary (numeric) – Extraordinary radius.
- index_relative (numeric) – The relative refractive index of the particle compared to the surrounding medium.

Optional named parameters

- internal (logical) – If true, the returned T-matrix is an internal T-matrix. Ignored for two outputs. Default: `false`.
- Nmax (numeric) – Size of the VSWF expansion used for the T-matrix calculation. Default: `ott.utis.ka2nmax(2*pi*max(radius))` (external) or `ott.utis.ka2nmax(2*pi*max(radius)*index_relative)` (internal).
- npts (numeric) – Number of points for surface integral. Default: `Nmax*Nmax`.
- verbose (logical) – Enables additional output from SMARTIES. Default: `false`.

4.4.8 Sub-packages

These sub-packages are used by T-matrix calculation methods and only minimal documentation is provided. Their location may change in a future releases (for instance, when DDA is extended to include force calculation methods).

SMARTIES

Contains components of the SMARTIES package for easy installation with the optical tweezers toolbox. Users interested in SMARTIES may want to download the full version, for details see

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

DDA

class `ott.tmatrix.dda.`**`Dipole`**(*varargin*)

Describes an array of radiating dipoles.

This class stores the dipole locations and polarizations. The scatted fields can be calculated using:

```
Es = F p
```

where F describes the locations where fields should be calculated and p describes the polarization of each dipole. The class provides methods for calculating F and Es.

Methods

- setDipoles – Set the dipole data (location/polarization)
- efield – Calculate E near-fields
- hfield – Calculate H near-fields
- efarfield – Calculate E far-fields
- hfarfield – Calculate H far-fields
- efarfield_matrix – Calculate far-field matrix for field calculation
- hfarfield_matrix – Calculate far-field matrix for field calculation
- enearfield_matrix – Calculate near-field matrix for field calculation
- hnearfield_matrix – Calculate near-field matrix for field calculation
- mtimes – Apply field matrix and calculate fields

Properties

- location – Dipole locations
- polarization – Dipole polarization
- xySymmetry – True if using z-mirror symmetry
- zRotSymmetry – Order of z-rotational symmetry (0 - infinite)
- parity – Parity of incident beam
- rorder – Rotational order of incident beam
- ndipoles – Number of dipoles in the array
- nbeams – Number of beams in the array

`Dipole`(*varargin*)

Construct a new dipole array

Usage beam = Dipole(locations, polarization, ...) Parameters can also be passed as named arguments.

Parameters

- locations (3xN numeric) – Locations of dipoles
- polarization (3NxM) – Dipole polarizations sorted packaged in [x1;y1;z1; x2;y2;z2; ...] order.

Optional named parameters

- parity (enum) – Parity of incident beam (even or odd). Only used when using z_mirror. Default: `'even'`.
- rorder (numeric) – Rotational order of incident beam. Only used when using z_rotation. Default: `0`.
- xySymmetry (logical) – If the particle has z-mirror symmetry. Default: `false`.
- zRotSymmetry (numeric) – Order of the particle is z-rotational symmetric. Default: `1`. If `0`, uses fourth order rotational symmetry (might change in a future release).

efarfield(*beam*, *rtp*, *varargin*)

Calculate the E-field

Usage E = beam.efarfield(rtp)

Parameters

- rtp – (3xN | 2xN numeric) Spherical coordinates. Either [radius; theta; phi] or [theta; phi]. Radius is ignored.

Unmatched parameters are passed to *efarfield_matrix()*.

efarfield_matrix(*beam*, *rtp*, *varargin*)

Evaluates the electric far-field matrix for a set of points

Can be applied to the beam to evaluate the scattered fields.

Es = F * beam

Usage F = beam.farfield_matrix(rtp, ...)

Parameters

- rtp (2xN|3xN numeric) – Far-field coordinates. Can either be [theta; phi] or [r; theta; phi] in which case *r* is ignored.

Optional named arguments

- low_memory (logical) – If true, evaluates the low-memory version of F. Default: `false`.

efield(*beam*, *xyz*, *varargin*)

Calculate the E-field

Evaluates:

```
Es = F * p
```

Usage E = beam.efield(xyz)

Parameters

- xyz – (3xN numeric) Cartesian coordinates.

Unmatched parameters are passed to `enearfield_matrix()`.

mtimes(*F*, *beam*)

Matrix multiplication for calculating scattered fields

Evaluates:

```
Es = F * p
```

Usage Es = F * beam Does not reshape the output.

Parameters

- F (numeric) – Field matrix calculated using `enearfield_matrix()`, *`efarfield_matrix()`*, `hnearfield_matrix()`, or `hfarfield_matrix()`.

setDipoles(*beam*, *locations*, *polarization*)

Set the dipole position and polarization data

Usage beam = beam.setDipoles(location, polarization)

Parameters

- location (3xN numeric) – Locations of dipoles
- polarization (3NxM) – Dipole polarizations sorted packaged in [x1;y1;z1; x2;y2;z2; . . .] order.

class ott.tmatrix.dda.**Dda**(*varargin*)

Minimal implementation of the discrete dipole approximation.

This class calculates the dipole polarizations from the interaction matrix A and the incident electric field E:

```
p = A \ E
```

The class implements methods for calculating the interaction matrix with optimisation for mirror, rotational symmetry and low memory.

Properties

- locations – Location of each dipole
- polarizability – Dipole polarizabilities
- xySymmetry – True if using z-mirror symmetry
- zRotSymmetry – Order of z-rotational symmetry (0 - infinite)
- ndipoles – Number of dipoles in the array

Methods

- solve – Solve for dipole polarizations
- interaction_matrix – Calculate the interaction matrix

Static methods

- FromShape – Construct instance from geometric shape

Dda(*varargin*)

Construct new DDA solver instance.

Usage dda = Dda(locations, polarizabilities, . . .)

Parameters

- voxels – (3xN numeric) Voxel locations in Cartesian coordinates.

- polarizabilities – (3x3N numeric) Array of 3x3 dipole polarizability tensors.

Optional named arguments

- xySymmetry (logical) – If the particle has z-mirror symmetry. Default: `false`. If true, locations with z<0 are removed.
- zRotSymmetry (numeric) – Order of the particle z-rotational symmetry. Default: `1`. If `0`, uses fourth order rotational symmetry (might change in a future release). If zRotSymmetry is 2, removes x<0 points. If zRotSymmetry is mod 4, removes x<0 | y<0 points. No other options supported for now.

static FromShape(*shape*, *varargin*)

Construct a DDA instance from a geometric shape.

Usage dda = ott.tmatrix.dda.Dda.FromShape(shape, ...)

Optional named arguments

- spacing – (numeric) – Dipole spacing in wavelength units. Default: `1/20`.
- polarizability – (function_handle | 1x1 | 3x3 numeric) Particle polarizability. Must be a function handle for inhomogeneous materials or either a scalar or 3x3 matrix for homogeneous materials. Default: `@(xyz, spacing, ri) polarizability.LDR(spacing, ri)`
- index_relative – (function_handle | numeric) Method to calculate relative refractive index or homogeneous value. Ignored if polarizability is not a function handle.

Unmatched parameters are passed to `FromPolarizability()`.

interaction_matrix(*dda*, *varargin*)

Calculate the interaction matrix assuming memory is limited

This method is rather time consuming and involves a lot of redundant calculations if called repeatedly with different rorder or parity parameters. For a faster but more memory intensive method see `DdaHighMemory`.

Usage A = dda.interaction_matrix(rorder, parity)

Optional parameters

- parity (enum) – Parity of incident beam (even or odd). Only used when using xySymmetry. Default: `'even'`.
- rorder (numeric) – Rotational order of incident beam. Only used when using zRotSymmetry. Default: `0`.

solve(*dda*, *Eincident*, *varargin*)

Solve for dipole polarizations

Usage dipoles = dda.solve(Eincident, ...)

Parameters

- Eincident – (3NxM numeric) Incident electric field at each dipole. Format: [x1;y1;z1;x2;y2;z2;...].

Optional named arguments

- solver – (function_handle) Solver to use. Good solvers to try include `gmres`, `bicgstab` and `\`. Default: `@(A, E) A \ E`.
- multiBeam – (logical) If true, passes the whole beam into the solver, otherwise iterates over each beam. Default: `true`.

- parity (enum) – Parity of incident beam (even or odd). Only used when using z_mirror. Default: `'even'`.
- rorder (numeric) – Rotational order of incident beam. Only used when using z_rotation. Default: `0`.

class `ott.tmatrix.dda.`**`DdaHighMem`**(*varargin*)

Slightly faster but more memory intensive version of DDA. Inherits from `ott.tmatrix.dda.Dda`.

Properties

- interaction – Stored interaction matrix

Methods

- update_interaction_matrix – Re-calculate the interaction matrix
- interaction_matrix – Get a usable interaction matrix

Static methods

- FromShape – Construct from a geometric shape

For other methods/properties, see *`Dda`*.

`DdaHighMem`(*varargin*)

Construct DDA instance and pre-compute data for interaction matrix.

Usage dda = DdaHigMem(dda) Convert an existing DDA instance into a pre-computed instance.

dda = DdaHighMem(locations, interaction, ...) Construct a new DDA instance. See base class for parameters.

static `FromShape`(*shape*, *varargin*)

Construct a DDA instance from a geometric shape.

Usage dda = ott.tmatrix.dda.Dda.FromShape(shape, ...)

Optional named arguments

- spacing – (numeric) – Dipole spacing in wavelength units. Default: `1/20`.
- polarizability – (function_handle | 3x3 numeric) Method to calculate polarizability or 3x3 tensor for homogeneous material. Default: `@(xyz, spacing, ri) polarizability.LDR(spacing, ri)`
- index_relative – (function_handle | numeric) Method to calculate relative refractive index or homogeneous value. Ignored if polarizability is a 3x3 tensor.

For further details and options, see *`Dda.FromShape()`*.

`update_interaction_matrix`(*dda*)

Update the interaction matrix data.

This method is called when the class is constructed. If you change the class properties, call this method again.

Usage dda = dda.update_interaction_matrix()

DDA polarizability methods

`ott.tmatrix.dda.polarizability.CM`(*spacing*, *index*)

Clausius-Mossoti Polarizability

Evaluates:

```
\alpha = \frac{3}{4\pi} s^3 \frac{n^2 - 1}{n^2 + 2}
```

Where n is the refractive index and s is the dipole spacing. This has units of L^3. To convert to SI units, multiply by $4\pi\epsilon_0$.

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

Parameters

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

`ott.tmatrix.dda.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.tmatrix.dda.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`.

4.5 *shape* Package

The *shape* package provides a collection of simple geometric shapes, methods for building arbitrary geometric shapes from functions/points, and methods to load geometry from files. The following sections describe the different shapes currently include in the toolbox. Additional shapes can be created using the shape builders or by sub-classing `Shape` to define a new shape class.

The basic shapes currently included in the toolbox focus on simple geometries and shapes of particles commonly trapped in optical tweezers. A summary of the different shapes and methods currently included is shown in Fig. 4.5. The package is split into four main sections: *Simple geometric shapes*, *Shape builders*, *File loaders* and *Collections*. The shape builders can be used to create arbitrary shapes from sets of vertices or parametric functions. In addition to the simple geometric shapes, some of the shape builder classes define static methods for building other commonly used shapes. More complex shapes can be created in external CAD programs (such as Blender) and loaded using the file loaders. The shape sets can be used to combine shapes or invert geometries.

The design of the shape classes is motivated by the scattering methods. Most scattering methods involve surface integrals, volume integrals or calculation of surface normals. These classes describe geometries with these quantities in mind. The complexity of various scattering simulations can often be reduced when the particle is star shaped, mirror symmetric or rotational symmetric. Consequently, all shapes have methods for querying these properties, although they may not be implemented for certain shapes (check documentation).

Contents

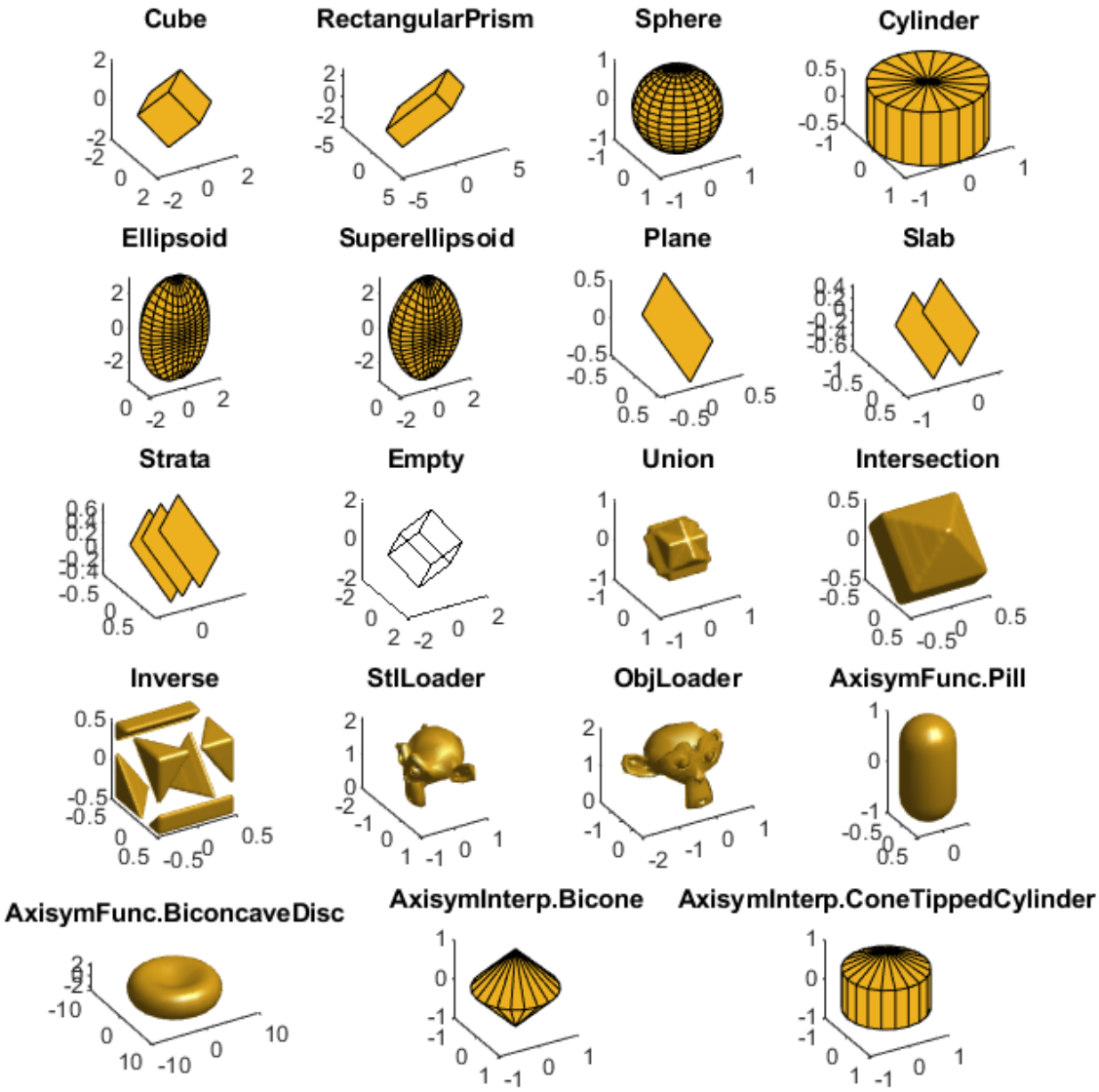

Fig. 4.5: Graphical display of the different shape creation methods currently included in the toolbox. Plot titles correspond to shapes classes and static methods of shapes classes. Most methods display the default geometry generated using *surf*. File loaders show different views of the suzane mesh; Empty is reprsented by a face-less cube; and collections show combinations of cubes.

4.5.1 Base class

The base class for all OTT shapes is the *Shape* class. This class defines the interface expected by most of the scattering methods. You will probably only need to use this class directly when implementing complex custom shapes. For simpler custom shapes, see the *Shape builders*. Additional classes describing additional conditions on shape geometry or helpers for defining commonly used methods can be found in the `ott.shapes.mixin` package.

Shape

class `ott.shape.`**`Shape`**(*varargin*)

Shape abstract class for optical tweezers toolbox shapes. Inherits from `ott.utils.RotationPositionProp` and *matlab.mixin.Hetrogeneous*.

Properties disregard the rotation/position properties and describe the object in the local coordinates.

Properties

- position – Location of shape `[x, y, z]`
- rotation – Orientation of the particle (3x3 rotation matrix)

Abstract properties

- maxRadius – Maximum particle radius
- volume – Particle volume
- boundingBox – Cartesian coordinate bounding box
- starShaped – True if the shape is star-shaped
- xySymmetry – True if shape is xy-mirror symmetric
- zRotSymmetry – z-axis rotational symmetry of particle

Methods

- surf – Generate surface visualisation
- scale – Scale the geometry (uses *scaleInternal*)
- voxels – Generate array of voxels or voxel visualisation
- insideRtp – Determine if Spherical point is inside shape
- insideXyz – Determine if Cartesian point is inside shape
- normalsRtp – Calculate normals at surface location
- normalsXyz – Calculate normals at surface location
- writeWavefrontObj – write shape to Wavefront OBJ file

- intersect – Calculate intersection between vectors and surface
- starRadii – Calculate radii of star shaped particle
- intersectAll – Calculate intersection between vectors and surface
- intersectBoundingBox – Calculate intersection with bounding box
- getBoundingBox – Get the bounding box with transformations applied
- getBoundingBoxShape – Get a shape representing the bounding box
- rotate* – Functions for rotating the entity
- translate* – Functions for translating the entity
- operator/ – Scale the object (uses scale)
- operator* – Scale the object (uses scale)
- operator| – Union operator: creates a new set
- operator& – Intersection operator: creates a new set
- operator~ – Inverse operator: creates a new `Inverse`.

Abstract methods

- scaleInternal – Scale the geometry (called by times/rdivide)
- surfInternal – Generate surface visualisation
- surfPoints – Calculate points for surface integration
- intersectInternal – Method called by intersect
- intersectAllInternal – Method called by intersectAll
- insideRtpInternal – Determine if Spherical point is inside shape
- insideXyzInternal – Determine if Cartesian point is inside shape
- normalsRtpInternal – Calculate normals at surface location
- normalsXyzInternal – Calculate normals at surface location

Supported casts

- TriangularMesh – Requires cast for PatchMesh

Shape(*varargin*)

Construct a new shape instance.

This class cannot be instanced directly, use one of the other shape descriptions to create a new shape.

Usage shape = shape@ott.shape.Shape(...)

Optional named arguments

- position (3 numeric) – Position of the shape. Default: `[0;0;0]`.
- rotation (3x3 numeric) – Orientation of the shape. Default: `eye(3)`.

and(*a*, *b*)

Create a intersection of two shapes

Usage shape = shape1 & shape2

getBoundingBox(*shape*, *varargin*)

Get bounding box after applying transformations

Usage bb = shape.getBoundingBox(...)

Optional named arguments

- origin (enum) – Coordinate origin. 'local' or 'global'.

insideRtp(*shape*, *varargin*)

Determine if point is inside the shape (Spherical coordinates)

Usage b = shape.insideRtp(rtp, ...)

Parameters

- rtp (3xN numeric) – Spherical coordinates.

Optional arguments

- origin (enum) – Coordinate system origin. Either 'global' or 'local' for world coordinates or shape coordinates.

insideXyz(*shape*, *varargin*)

Determine if point is inside the shape (Cartesian coordinates)

Usage b = shape.insideXyz(xyz, ...)

Parameters

- xyz (3xN numeric) – Cartesian coordinates.

Optional arguments

- origin (enum) – Coordinate system origin. Either 'global' or 'local' for world coordinates or shape coordinates.

intersect(*shape*, *x0*, *x1*, *varargin*)

Calculate intersection locations and normals.

Any rays that don't intersect shape result in nans.

Usage [locs, norms, dist] = shape.intersectAll(shape, vecs, ...)

Parameters

- vecs (3xM Vector) – vectors to intersect. Must be castable to a ott.utils.Vector object.

Returns

- locs (3xN numeric) – intersections with N locations
- norms (3xN numeric) – surface normals at N intersections

Optional named arguments

- origin (enum) – Coordinate origin. 'local' or 'global'.

Additional arguments passed to intersectInternal.

isosurface(*shape*, *varargin*)

Generate an isosurfrace for the shape from the voxel data

This method is more intensive than the surf method and often less accurate but provides a useful alternative when the shape doesn't directly support a surf method.

Usage FV = shape.isosurface(...)

Optional named parameters

- samples (3 numeric) – Number of samples per Cartesian axes.
- visualise (logical) – Show the visualisation. Default: `nargout == 0`
- origin (enum) – Coordinate system origin. Either 'global' or 'local' for world coordinates or shape coordinates. Default: `'global'`.
- axis (handle) – Axes hanlde to place plot in. Default: `[]`, uses gca() when available.

normalsRtp(*shape*, *varargin*)

Calculate normals at the specified surface locations

Usage nxyz = shape.normalsRtp(rtp, ...) Result is in Cartesian coordinates.

Parameters

- rtp (3xN numeric) – Spherical coordinates.

Optional arguments

- origin (enum) – Coordinate system origin. Either 'global' or 'local' for world coordinates or shape coordinates.

normalsXyz(*shape*, *varargin*)

Calculate normals at the specified surface locations

Usage nxyz = shape.normalsXyz(xyz, ...) Result is in Cartesian coordinates.

Parameters

- xyz (3xN numeric) – Cartesian coordinates.

Optional arguments

- origin (enum) – Coordinate system origin. Either 'global' or 'local' for world coordinates or shape coordinates.

not(*shape*)

Take the inverse of the shape

Usage shape = ~shape

or(*a*, *b*)

Create a union of two shapes

Usage shape = shape1 | shape2

surf(*shape*, *varargin*)

Generate a visualisation of the shape

Usage shape.surf(...) Display visualisation of shape(s).

S = shape.surf(...) Returns a array of handles to the generated patches or when no visualisation is enabled, creates a cell array of structures that can be passed to patch.

Optional named parameters

- axes (handle) – axis to place surface in (default: gca)
- surfOptions (cell) – options to be passed to patch (default: {})
- showNormals (logical) – Show surface normals (default: false)
- origin (enum) – Coordinate origin for drawing. Can be 'global' or 'local' Default: 'global'.

- visualise (logical) – Show the visualisation. Default: `true`
- normalScale (numeric) – Scale for normal vectors. Default: `0.1`.

Additional parameters passed to `surfInternal()`.

voxels(*shape*, *varargin*)

Generate an array of xyz coordinates for voxels inside the shape

Usage voxels(spacing) shows a visualisation of the shape with circles placed at locations on a Cartesian grid.

xyz = voxels(spacing) returns the voxel locations.

Optional named arguments

- 'plotoptions' Options to pass to the plot3 function
- 'visualise' Show the visualisation (default: nargout == 0)
- origin (enum) – Coordinate system origin. Either 'global' or 'local' for world coordinates or shape coordinates. Default: `'global'`.
- axes (handle) – Axes hanlde to place plot in. Default: `[]`, uses gca() when available.

writeWavefrontObj(*shape*, *filename*)

Write representation of shape to Wavefront OBJ file

Convert the shape to a TriangularMesh and write it to a file.

Usage shape.writeWavefrontObj(filename)

Parameters

- filename (char | string) – Filename for file to write to.

4.5.2 Simple geometric shapes

Cube

class ott.shape.**Cube**(*varargin*)

Simple geometric cube. Inherits from *Shape*.

Properties

- width – Width of the cube

Additional properties inherited from base.

Cube(*varargin*)

Construct a cube.

Usage shape = Cube(width, ...) Parameters can be passed as named arguments.

Additional parameters are passed to base.

RectangularPrism

class ott.shape.**RectangularPrism**(*varargin*)

Simple geometric rectangular prism. Inherits from *Shape*.

Properties

- widths – Widths of each side [x; y; z]

Supported casts

- ott.shape.Cube – Only if widths all match

Additional properties inherited from base.

RectangularPrism(*varargin*)

Construct a rectangular base prism

Usage shape = RectangularPrism(widths, ...) Parameters can be passed as named arguments.

Additional parameters are passed to base.

Sphere

class ott.shape.**Sphere**(*varargin*)

Spherical particle. Inherits from *Shape*.

Properties

- radius – Radius of the sphere

Additional properties inherited from base.

Sphere(*varargin*)

Construct a new sphere

Usage shape = Sphere(radius, ...)

Additional parameters passed to base class.

Cylinder

class ott.shape.**Cylinder**(*varargin*)

A simple cylindrical shape. Inherits from *Shape*, `mixin.AxisymStarShape` and `mixin.IsosurfSurfPoints`.

Properties

- radius – Radius of the cylinder
- height – Height of the cylinder

Cylinder(*varargin*)

Construct a new cylinder

Usage shape = Cylinder(radius, height, ...) Parameters can also be passed as named arguments.

Additional parameters passed to base *Shape*.

Ellipsoid

class ott.shape.**Ellipsoid**(*varargin*)

Ellipsoid shape

Properties:

- radii – Radii along Cartesian axes [X; Y; Z]

Supported casts

- ott.shape.Spehre – Only works when ellipsoid is a sphere
- ott.shape.AxisymInterp – Only works when zRotSymmetry == 0
- ott.shape.PatchMesh

Ellipsoid(*varargin*)

Construct a new ellipsoid

Usage shape = Ellipsoid(radii, ...) Parameters can be passed as named arguments.

Additional parameters passed to base.

Superellipsoid

class ott.shape.**Superellipsoid**(*varargin*)

Superellipsoid shape

In Cartesian coordinates, a superellipsoid is defined by

$$(|x/a|^{2/e} + |y/b|^{2/e})^{e/n} + |z/c|^{2/n} = 1$$

where a, b, c are the radii along Cartesian directions, and e, n are the east-west and north-south smoothness parameters. For more details see https://en.wikipedia.org/wiki/Superellipsoid

Properties

- radii – Radii along Cartesian axes [X; Y; Z]
- ew – East-West smoothness (ew = 1 for ellipsoid)
- ns – North-South smoothness (sw = 1 for ellipsoid)

Superellipsoid(*varargin*)

Construct a new superellipsoid

Usage shape = Superellipsoid(radii, ew, ns, ...)

Parameters

- radii (3 numeric) – Radii along Cartesian axes.
- ew (numeric) – East-west smoothness (xy-plane) Default: 0.8
- ns (numeric) – North-south smoothness (z-axis) Default: 1.2

Plane

class `ott.shape.Plane`(*varargin*)

Shape describing a plane with infinite extent Inherits from `ott.shape.Shape`.

Dependent properties

- normal – Vector representing surface normal
- offset – Offset of surface from coordinate origin

Supported casts

- TriangularMesh – (Inherited) Uses PatchMesh
- PatchMesh
- Strata
- Slab

`Plane`(*varargin*)

Construct a new infinite plane

Usage shape = Plane(normal, . . .)

Optional named arguments

- normal (3xN numeric) – Normals to planes, pointing outside. Default: `[]`. Overwrites any values set with *rotation*.
- offset (1xN numeric) – Offset of the plane from the position. Default: `[]`. Overwrites any values set with *position*.
- position (3xN numeric) – Position of the plane. Default: `[0;0;0]`.
- rotation (3x3N numeric) – Plane orientations. Default: `eye(3)`.

Slab

class `ott.shape.Slab`(*varargin*)

Shape describing a slab with infinite extent in two directions Inherits from `ott.shape.Shape`.

Properties

- normal – Vector representing surface normal
- depth – Depth of the slab

Supported casts

- TriangularMesh – (Inherited) Uses PatchMesh
- PatchMesh – Uses Strata
- Strata

`Slab`(*varargin*)

Construct a new infinite slab

Usage shape = Slab(depth, normal, . . .)

Optional named arguments

- depth (N numeric) – Depth of surface. Default: 0.5.

- normal (3xN numeric) – Normals to planes. Default: `[]`. Overwrites any values set with *rotation*.
- position (3xN numeric) – Position of the plane. Default: `[0;0;0]`.
- rotation (3x3N numeric) – Plane orientations. Default: `eye(3)`.

Strata

class `ott.shape.`**`Strata`**`(`*varargin*`)`

Shape describing a series of stratified interfaces. Inherits from `Plane`.

This shape describes a series of layered planes. When the number of layers is equal to 2, this object can be converted to a Slab. All points above the first layer are considered to be inside the shape.

Properties

- normal – Vector representing surface normal
- offset – Offset of surface from coordinate origin
- depths – Depth of each layer (must be positive)

`Strata``(`*varargin*`)`

Construct a new infinite slab

Usage shape = Slab(depths, normal, . . .)

Optional named parameters

- depths (numeric) – Depth of each layer. Default: `[0.2, 0.5]`.
- normal (3x1 numeric) – Surface normal. Default: `[]`. Overwrite any value set with `rotation`.
- offset (numeric) – Offset of the plane from the position. Default: `[]`. Overwrites any values set with `position`.
- position (3x1 numeric) – Position of the plane. Default: `[0;0;0]`.
- rotation (3x3 numeric) – Plane orientation. Default: `eye(3)`.

Empty

class `ott.shape.`**`Empty`**`(`*varargin*`)`

An empty shape (with no geometry)

The element still inherits from Shape and has position and rotation properties. The object can be visualised with surf, which draws a cube with transparent faces.

This is the default element in an empty Shape array.

Properties

- volume – Shape has no volume
- maxRadius – Shape max radius is zero

Methods

- surf – Draws a hollow cube
- surfPoints – Returns an empty array

Empty(*varargin*)

Construct a new empty shape

Usage shape = Empty(...)

Optional parameters

- position (3xN numeric) – Position of the shape. Default: `[0;0;0]`.
- rotation (3x3N numeric) – Orientation of the shape. Default: `eye(3)`.

4.5.3 Shape builders

TriangularMesh

class ott.shape.**TriangularMesh**(*verts*, *faces*, *varargin*)

Describes a mesh formed by triangular patches.

This class is similar to `PatchMesh` except the patches must be triangles (described by three vertices).

Properties

- verts – 3xN matrix of vertex locations
- faces – 3xN matrix of vertex indices describing faces
- norms – 3xN matrix of face normal vectors
- zRotSymmetry – (Can be set) rotational symmetry of shape
- xySymmetry – (Can be set) mirror symmetry of shape
- starShaped – (Can be set) if the particle is star shaped

Methods

- subdivide – Add an extra vertex to the centre of each face

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

TriangularMesh(*verts*, *faces*, *varargin*)

Construct a new triangular mesh representation

Usage shape = TriangularMesh(verts, faces)

Parameters

- verts (3xN numeric) – Vertex locations
- faces (3xN numeric) – Indices describing faces

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

Any faces formed by duplicate vertices are removed.

PatchMesh

class ott.shape.**PatchMesh**(*verts*, *faces*, *varargin*)

A surface resembling Matlab polygon patches.

This surface casts to `TriangularMesh` for most operations.

Properties

- verts – (3xN numeric) Array of vertices for forming faces
- faces – (mxN numeric) Vertex indices for polygons

PatchMesh(*verts*, *faces*, *varargin*)

Construct a new Patch mesh representation

Usage shape = PatchMesh(verts, faces, ...)

Parameters

- verts (3xN numeric) – Array of vertices for forming faces
- faces (mxN numeric) – Vertex indices for polygons

AxisymInterp

class ott.shape.**AxisymInterp**(*varargin*)

Rotationally symmetric shape described by discrete set of points.

Shape produced when converting this object to a patch uses linear interpolation between points and discrete rotational segments.

Properties

- points – Discrete points describing surface [rho; z]

Methods

- boundaryPoints – Calculate points for line integration
- surfPoints – Calculate points for surface integration

Supported casts

- PatchMesh

Static methods

- Bicone – Create a bicone
- ConeTippedCylinder – Create a cone-tipped cylinder

Volume is computed numerically, may change in future.

Additional properties/methods inherited from base.

AxisymInterp(*varargin*)

Construct a new rotationally symmetry shape from discrete points

Usage shape = AxisymInterp(points, ...)

Parameters

- points (2xN numeric) – Array of points in cylindrical coordinates describing shape geometry. [rho; z]

Additional parameters passed to base.

static Bicone(*varargin*)

Construct a bicone

A Bicone is xy-mirror symmetric and has three vertices, two on the +(ve)/-(ve) axes and one in the mirror symmetric plane.

Usage shape = AxisymInterp.Bicone(height, radius, ...)

Parameters

- height (numeric) – Total height of the shape (default: 2.0)
- radius (numeric) – Radius of the cone (default: 1.0)

Additional parameters passed to class constructor.

static ConeTippedCylinder(*varargin*)

Construct a cone-tipped cylinder

Usage shape = ConeTippedCylinder(height, radius, coneHeight, ...)

Parameters

- height (numeric) – total height of the shape (default: 2.0)
- radius (numeric) – radius of cylinder (default: 1.0)
- coneHeight (numeric) – Height of the cone segment (default: 0.5)

Additional parameters passed to class constructor.

AxisymFunc

class ott.shape.**AxisymFunc**(*varargin*)

Rotationally symmetry shape described by a function

Properties

- func – Function describing surface
- type – Type of function (radial | angular | axial | axialSym)
- range – Range of function parameter values (default: [-Inf, Inf])

Methods

- surf – Visualise the shape (via PatchMesh)

Static methods

- BiconcaveDisc – Create a biconcave disk shape
- Pill – Create a pill tipped shaped cylinder

Supported casts

- AxisymInterp
- PatchMesh – Via AxisymInterp

AxisymFunc(*varargin*)

Construct a new rotationally symmetric shape from a function

Usage shape = AxisymFunc(func, type, ...)

Parameters

- func (function_handle) – A function handle describing the shape surface. The function should take a single vectorised argument. The argument will depend on the *type*: radial: z, angular: theta, axial: r
- type (enum) – Type of function. Can either be 'angular', 'radial', 'axial' or 'axialSym'.

Optional named arguments

- range (2 numeric) – Range of function parameter values. Default: `[-Inf, Inf]` (radial), `[-pi, pi]` (angular), and `[0, Inf]` (axial/axialSym).

Additional parameters passed to base.

static BiconcaveDisc(*varargin*)

Construct a biconcave disc shape

This shape can be used to model cells such as unstressed Red Blood Cells. It implements the function:

```
z(r) = D \sqrt{1 - \frac{4r^2}{D^2}} \left(a_0 +
    \frac{a_1 r^2}{D^2} + \frac{a_2 r^4}{D^4} \right)
```

where D is the particle diameter and a are shape coefficients.

Usage shape = AxisymFunc.BiconcaveDisc(radius, coefficients, ...)

Parameters

- radius (numeric) – Radius of disc. Default: `7.82`.
- coefficients (3 numeric) – Coefficients describing shape. Default: `[0.0518, 2.0026, -4.491]`.

Additional parameters are passed to shape constructor.

static Pill(*varargin*)

Construct a pill-shaped particle

Constructs a cylindrical shaped rod with spherical end caps.

Usage shape = AxisymFunc.Pill(height, radius, capRadius, ...)

Parameters

- height (numeric) – Total height of pill including end caps. Default: `2.0`.
- radius (numeric) – Radius of rod segment. Default: `0.5`.
- capRadius (numeric) – Radius of end cap. Default: `radius`. If *capRadius* is greater than *radius*, adds a sharp edge at the intersection of the cap and the rod. If radius is less, adds a flat end-cap.

Additional parameters are passed to class constructor.

4.5.4 File loaders

For more complex shapes, it is often more convinent to work with a dedicated computer aided design (CAD) program such as Blender. The toolbox currently includes two types of commonly used CAD formats: STL and Wavefront OBJ.

StlLoader

class `ott.shape.StlLoader`(*filename*, *varargin*)

Load a shape from a STL file. Inherits from `TriangularMesh`.

Properties

- filename – Name of the file this object loaded

Additional methods and properties inherited from base class.

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.

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

`StlLoader`(*filename*, *varargin*)

Construct a new shape from a STL file

Usage shape = StlLoader(filename, ...) Loads the face and vertex information contained in the file.

Only supports binary STL files.

This function uses 3rd party code, see `tplicenses/stl_EricJohnson.txt` for licensing information.

ObjLoader

class `ott.shape.ObjLoader`(*filename*)

Load a shape from a Wavefront OBJ file. Inherits from `TriangularMesh`.

Properties

- filename – Filename for loaded OBJ file

The file format is described at https://en.wikipedia.org/wiki/Wavefront_.obj_file

`ObjLoader`(*filename*)

Construct a new shape from a Wavefront OBJ file

Loads the face and vertex information contained in the file. Faces are converted to triangles.

Usage shape = ObjLoader(filename)

4.5.5 Collections

These classes can be used for combining or modifying existing geometry. For example, to combine two shapes so that all points inside either shape are considered inside the combined shape, use a union:

```
shape = shape1 | shape2;
```

Or, to create a shape from the intersection of two shapes, use:

```
shape = shape1 & shape2;
```

To invert the inside/outside of a shape, use the inverse:

```
shape = ~shape;
```

Set

class ott.shape.**Set**(*shapes*, *varargin*)

Collection of shapes Inherits from *Shape*.

This is the base class for collections of shapes including *Union* and *Intersection*.

Properties

- shapes – Shapes forming the set
- volume – Calculated numerically
- starShaped – Variable, default false
- xySymmetry – Variable, default false
- zRotSymmetry – Variable, default 1

Set(*shapes*, *varargin*)

Construct a new shape set

This is the abstract constructor for shape sets. Use Union or Intersection for instances.

Usage shape = shape@ott.shape.Set(shapes, . . .)

Stores shapes and passes optional arguments to base.

Union

class ott.shape.**Union**(*varargin*)

Represents union between two shapes (| operator). Inherits from *Set*.

Properties

- shapes – Shapes forming the set
- maxRadius – Estimated from bounding box
- volume – Calculated numerically
- boundingBox – Surrounding all shapes

Methods

- operator| – (Overloaded) Allow daisy chains

Union(*varargin*)

Construct a union shape set.

Usage shape = Union(shapes, . . .)

All parameters are passed to base class.

Intersection

class ott.shape.**Intersection**(*varargin*)

Represents intersection between two shapes (& operator). Inherits from `Set`.

Properties

- shapes – Shapes forming the set
- maxRadius – Estimated from bounding box
- volume – Calculated numerically
- boundingBox – Surrounding intersection

Methods

- operator& – (Overloaded) Allow daisy chains

Intersection(*varargin*)

Construct a intersection shape set.

Usage shape = Intersection(shapes, . . .)

All parameters are passed to base class.

Inverse

class ott.shape.**Inverse**(*internal*, *varargin*)

Inverts the geometry of a shape. Inherits from `Shape`.

Properties

- internal – Internal shape that is inversed
- volume – If volume was finite, makes it infinite
- maxRadius – If maxRadius was finite, makes it infinite

Methods

- operator~ – Smart inverse

Inverse(*internal*, *varargin*)

Construct a new inverse shape

Usage shape = Inverse(internal, . . .)

Additional parameters passed to base.

4.6 *drag* Package

The *drag* package provides methods for calculating Stokes drag for spherical and non-spherical particles in unbounded free fluid and methods to calculate the wall effects for spheres near walls and eccentric spheres. These methods assume Stokes flow (creeping flow) (i.e., low Reynolds number).

All drag methods inherit from `Stokes`. The simplest way to construct the drag tensor for a set of shapes is to call the static method `Stokes.FromShape()` with the geometry, for example:

```
shape = ott.shapes.Sphere();
drag = ott.drag.Stokes.FromShape(shape);
```

`Stokes.FromShape()` can also be called with arrays of shapes. In this case, the method will attempt to generate an array of drag tensors for the specified configuration and raise a warning or error when the configuration is unlikely to behave as expected.

Classes in this package can be classified into two main categories: methods for *Free particles* and methods for *Wall effects*. A summary of these classes is shown graphically in Fig. 4.6.

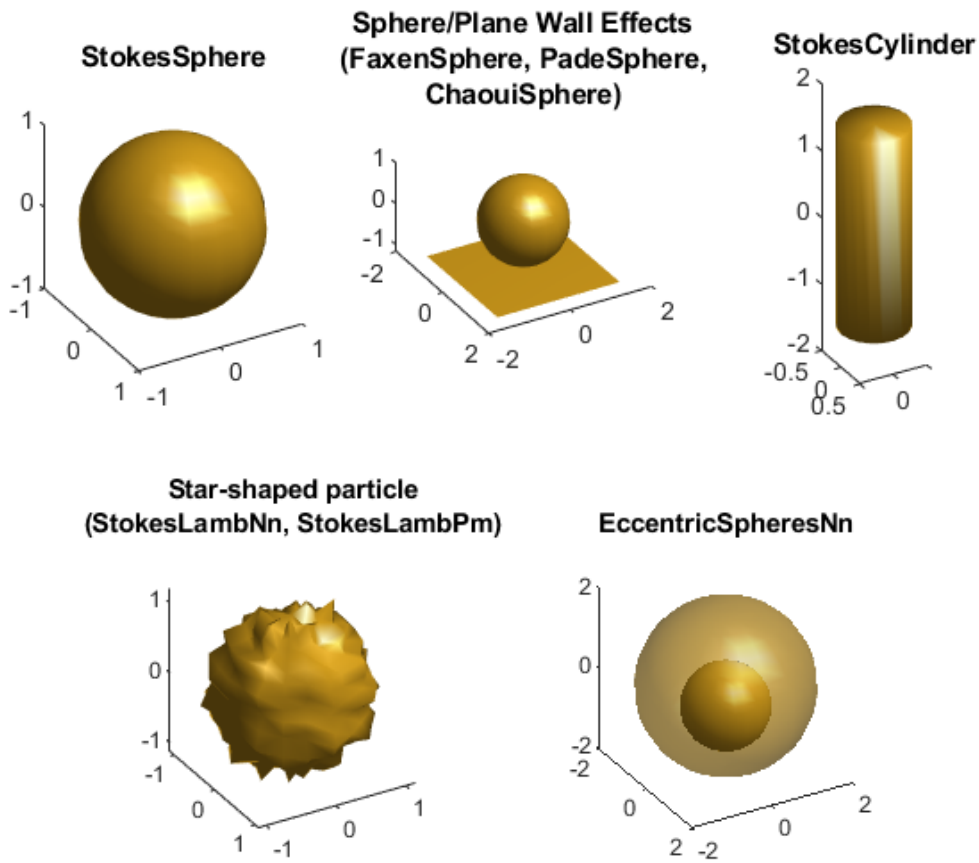

Fig. 4.6: Graphical display of different particle shapes/scenarious for which drag methods are provided. Other shapes/geometries can be approximated by choosing one of the above drag methods.

Contents

4.6.1 Generic classes

The base class for all drag tensors is the `Stokes` class. This class specifies the common functionality for drag tensors and provides the static method for creating drag tensors from shape arrays.

The `StokesData` class is an instance of the `Stokes` class which provides data storage of the drag and inverse drag tensors. Unlike other classes, which compute the drag tensors as needed, the `StokesData` class stores the drag tensors. All other classes can be cast to the `StokesData` class using:

```
dragData = ott.drag.StokesData(drag);
```

This operation performs all drag calculations and applies any rotations to the particle; this may provide a performance improvement if the same drag is needed for repeated calculations.

Stokes (base class)

class `ott.drag.`**`Stokes`**(*varargin*)

Base class for 6-vector force/torque drag tensors. Inherits from `ott.utils.RotateHelper`.

This class is the base class for drag tensors which can be described by a 3x3 translational, rotational, and cross-term matrices in Cartesian coordinates.

Properties

- forward – Rank 2 forward tensor
- inverse – Rank 2 inverse tensor
- rotation – Rotation matrix to apply to forward/inverse

Abstract properties

- forwardInternal – Forward tensor without rotation
- inverseInternal – Inverse tensor without rotation

Dependent properties

- gamma – Translational component of tensor

- delta – Rotational component of tensor
- crossterms – Cross-terms component of tensor (UD)
- igamma – Inverse translational component of tensor
- idelta – Inverse rotational component of tensor
- icrossterms – Inverse cross-terms component of tensor (UD)

Methods

- inv – Return the inverse or calculate the inverse
- mtimes – Multiply the tensor or calculate the forward and mul
- vecnorm – Compute vecnorm of tensor rows
- diag – Get the diagonal of the forward tensor
- rotate* – Rotate the tensor around the X,Y,Z axis

Static methods

- FromShape – Construct a drag tensor from a shape or array

Supported casts

- double – Get the forward drag tensor
- StokesData – Pre-computes forward/inverse tensors
- ott.shape.Shape – Gets a geometrical representation of the object

See also `StokesSphere` and `StokesLambNn`.

`static FromShape`(*shape*, *varargin*)

Attempt to select an appropriate drag calculation method.

Not all shapes are supported, most shapes default to a sphere whose radius matches the maxRadius of the shape.

If the input is a single shape, attempts to choose an appropriate method for the shape. May raise warnings if no good method is found.

If the input contains a plane and a finite extent particle, models the particle as a sphere near a wall. Raises a warning if the particle is not a sphere.

If the input contains a particle inside the circumscribing sphere of another particle, uses EccentricSpheresNn. Raises a warning if the particles are not spheres.

For all other arrays, assumes the particles are non-interacting and generates drag tensors for each particle. If the particles are within 5 radii of each other, raises a warning.

Usage drag = Stokes.FromShape(shape, eta, ...)

Parameters

- shape (ott.shape.Shape) – A shape or array of shapes.
- eta (numeric) – Viscosity (passed to class constructor).

Additional parameters passed to class constructors.

`Stokes`(*varargin*)

Construct a new drag tensor instance.

Usage drag = Stokes(...)

Optional named parameters

- rotation (3x3 numeric) – Initial rotation property. Default: `eye(3)`.

StokesData

class `ott.drag.`**`StokesData`**(*varargin*)

Stokes drag tensor definition with explicit forward and inverse data

Properties

- inverse – (6x6 numeric) inverse drag tensor
- forward – (6x6 numeric) forward drag tensor

Additional methods/properties inherited from `Stokes`.

`StokesData`(*varargin*)

Construct new Stokes drag tensor instance with explicit data

Usage drag = StokesDrag(forward, inverse, . . .)

Parameters

- forward (6x6 numeric) – Forward drag tensor
- inverse (6x6 numeric) – Inverse drag tensor

If either of *forward* or *inverse* are omitted, they are calculated from the corresponding tensor.

Parameters can also be passed as named arguments. Unmatched parameters are passed to `Stokes`.

4.6.2 Free particles

These classes calculate drag tensors for particles in unbounded free fluid. Spheres have an analytical solution and can be calculated using `StokesSphere`. Arbitrary shapes can be calculated in spherical coordinates using point-matching using Lamb Series via `StokesLambPm`. `StokesLambNn` is a pre-trained neural network for the Lamb series solution for arbitrary shaped particles which enables faster drag prediction with comparable accuracy to `StokesLambPm`. For Cylinders or elongated particles with high aspect ratios it may be better to use the `StokesCylinder` approximation instead.

StokesSphere

class `ott.drag.`**`StokesSphere`**(*varargin*)

Drag tensor for a sphere with Stokes Drag

Properties

- radius – Radius of sphere
- viscosity – Viscosity of medium
- forward – Calculated drag tensor
- inverse – Calculate from forward

Supported casts

- ott.shape.Shape – Constructs a sphere shape

See `Stokes` for other methods/parameters.

`StokesSphere`(*varargin*)

Calculate drag tensors for spherical particle in Stokes drag.

Usage: tensor = Sphere(radius, viscosity, ...)

Parameters

- radius – (numeric) Radius of particle (default: 1.0)
- viscosity – (numeric) Viscosity of medium (default: 1.0)

Parameters can also be passed as named arguments. Unmatched parameters are passed to `Stokes`.

StokesCylinder

`class ott.drag.StokesCylinder`(*varargin*)

Drag tensor for long/slender cylindrical particles.

Uses the results from

> Maria M. Tirado and José García de la Torre J. Chem. Phys. 71, 2581 (1979); https://doi.org/10.1063/1.438613

And from

> María M. Tirado and José García de la Torre J. Chem. Phys. 73, 1986 (1980); https://doi.org/10.1063/1.440288

which provide lookup tables for the drag corrections for slender cylinders (aspect ratio *height/diameter* above 2).

Properties

- radius – Cylinder radius
- height – Cylinder height
- viscosity – Medium viscosity
- forward – Calculated drag tensor
- inverse – Inverse drag tensor (calculated from *forward*)

Supported casts

- ott.shape.Shape – Construct a cylinder

Additional properties/methods inherited from `Stokes`.

`StokesCylinder`(*varargin*)

Calculate drag tensors for cylindrical particles

Usage: tensor = Sphere(radius, height, viscosity, ...)

Parameters

- radius – (numeric) Radius of cylinder (default: 1.0)
- height – (numeric) Height of cylinder (default: 4.0)
- viscosity – (numeric) Viscosity of medium (default: 1.0)

Parameters can also be passed as named arguments. Unmatched parameters are passed to `Stokes`.

StokesLambNn

class ott.drag.**StokesLambNn**(*varargin*)

Calculate stokes drag for star shaped particles using pre-trained NN. Inherits from `StokesStarShaped`.

Uses the neural network from Lachlan Gibson's PhD thesis (2016).

Properties

- inverse – Calculated from *forward*
- forward – Drag tensor calculated using point matching.
- viscosity – Viscosity of medium
- shape – A star shaped particle describing the geometry

Static methods

- SphereGrid – Generate grid of angular points for NN
- LoadNetwork – Load the neural network

Additional methods/properties inherited from `Stokes`.

StokesLambNn(*varargin*)

Construct star-shaped drag method using Neural network approach.

Usage drag = StokesLambNn(shape, . . .)

Parameters

- shape (ott.shape.Shape) – Star shaped particle.

Parameters can also be passed as named arguments. Additional parameters passed to base.

StokesLambPm

class ott.drag.**StokesLambPm**(*varargin*)

Calculate drag coefficients using Lamb series and point matching. Inherits from `StokesStarShaped`

Properties

- inverse – Calculated from *forward*
- forward – Drag tensor calculated using point matching.
- viscosity – Viscosity of medium
- shape – A star shaped particle describing the geometry

Additional methods/properties inherited from `Stokes`.

Based on and incorporating code by Lachlan Gibson (2016)

StokesLambPm(*varargin*)

Construct star-shaped drag method using point-matching method.

Usage drag = StokesLambPm(shape, . . .)

Parameters

- shape (ott.shape.Shape) – Star shaped particle.

Parameters can also be passed as named arguments. Additional parameters passed to base.

4.6.3 Wall effects

The toolbox includes several approximation methods for calculating drag for spheres near walls. The `EccentricSpheresNn` can be used to simulate a sphere inside another sphere or by making the outer sphere radius significantly larger than the inner sphere, a sphere near a wall. `FaxenSphere`, `ChaouiSphere` and `PadeSphere` provide drag tensors for spheres near walls with varying approximations. `FaxenSphere` may provide slightly faster calculation time compared to the other methods but only works well when the particle is a fair distance from the wall.

EccentricSpheresNn

class ott.drag.**EccentricSpheresNn**(*varargin*)

Calculate drag on an eccentric sphere using Gibson's NN approach. Inherits from `Stokes`.

Uses the NN from

> Lachlan J. Gibson, et al. Phys. Rev. E 99, 043304 https://doi.org/10.1103/PhysRevE.99.043304

Properties

- innerRadius – Radius of inner sphere
- outerRadius – Radius of outer sphere
- separation – Minimum separation between spheres
- viscosity – Viscosity of surrounding fluid (default: 1.0)
- forward – Computed drag tensor
- inverse – Computed from *forward*.

See `Stokes` for other methods/parameters.

FaxenSphere

class ott.drag.**FaxenSphere**(*varargin*)

Stokes drag with Faxen's corrections for movement near a plane. Inherits from `StokesSphereWall`.

Faxen's correction can provide reasonable estimates for the drag on a spherical particle moving near a planar surface. The approximation works well to within about 1 particle radius from the surface.

This implementation uses the Faxen's corrections described in

> J. Leach, et al. Phys. Rev. E 79, 026301 https://doi.org/10.1103/PhysRevE.79.026301

For the rotational-translational Faxen's coupling, we use Eq. 7-4.29 from

> John Happel and Howard Brenner, Low Reynolds number hydrodynamics, (1983) https://doi.org/10.1007/978-94-009-8352-6

This class assumes the surface is perpendicular to the z axis, positioned bellow the spherical particle.

Properties

- radius – Radius of the sphere
- viscosity – Viscosity of the medium
- separation – Distance between the sphere centre and plane
- forward – Computed drag tensor

- inverse – Computed from *forward*.

See `Stokes` for other methods/parameters.

FaxenSphere(*varargin*)

Construct a new Faxen's corrected sphere drag tensor.

Usage drag = FaxenSphere(radius, separation, viscosity, ...) Radius and separation should be specified in the same units.

Parameters

- radius – (numeric) Radius of sphere (default: 1)
- separation – Separation between sphere centre and surface
- viscosity – Viscosity of medium (default: 1.0)

Parameters can also be passed as named arguments. Unmatched parameters are passed to `Stokes`.

PadeSphere

class ott.drag.**PadeSphere**(*varargin*)

Creeping flow around a sphere in shear flow near to a wall. Inherits from `StokesSphereWall`.

This class implements the Pade series approximation for spherical particles moving near a planar surface. The approximation should work for spacing between the sphere surface and plane larger than 10^{-2} radius.

This class uses the coefficients included in

> M. Chaoui and F. Feuillebois, Creeping Flow Around a Sphere in a Shear Flow Close to a Wall. The Quarterly Journal of Mechanics and Applied Mathematics, Volume 56, Issue 3, August 2003, Pages 381–410, https://doi.org/10.1093/qjmam/56.3.381

This class assumes the surface is perpendicular to the z axis, positioned bellow the spherical particle.

Properties

- radius – Radius of the sphere
- viscosity – Viscosity of the medium
- separation – Distance between the sphere centre and plane
- forward – Computed drag tensor
- inverse – Computed from *forward*.

See `Stokes` for other methods/parameters.

PadeSphere(*varargin*)

Construct a new creeping flow sphere-wall drag tensor.

Usage: drag = PadeSphere(radius, separation, viscosity, ...) radius and separation should be specified in the same units.

Parameters:

- radius – Radius of sphere (default: 1.0)
- separation – Separation between sphere centre and surface
- viscosity – Viscosity of medium (default: 1.0)

Parameters can also be passed as named arguments. Unmatched parameters are passed to `Stokes`.

ChaouiSphere

class ott.drag.**ChaouiSphere**(*varargin*)

Creeping flow around a sphere close to a wall. Inherits from `StokesSphereWall`.

This class implements a polynomial fit to the exact solution for spherical particles moving in creeping flow near a planar surface. The approximation should work for spacing between the sphere surface and plane between 10^{-6} radius and 1 radius.

This class implements the polynomial approximation described in

> M. Chaoui and F. Feuillebois, Creeping Flow Around a Sphere in a Shear Flow Close to a Wall. The Quarterly Journal of Mechanics and Applied Mathematics, Volume 56, Issue 3, August 2003, Pages 381–410, https://doi.org/10.1093/qjmam/56.3.381

This class assumes the surface is perpendicular to the z axis, positioned bellow the spherical particle.

Properties

- radius – Radius of the sphere
- viscosity – Viscosity of the medium
- separation – Distance between the sphere centre and plane
- forward – Computed drag tensor
- inverse – Computed from *forward*.

See `Stokes` for other methods/parameters.

ChaouiSphere(*varargin*)

Construct a new creeping flow sphere-wall drag tensor.

Usage: drag = ChaouiSphere(radius, separation, viscosity, ...) radius and separation should be specified in the same units.

Parameters:

- radius – Radius of sphere
- separation – Separation between sphere centre and surface
- viscosity – Viscosity of medium (optional, default: 1.0)

Parameters can also be passed as named arguments. Unmatched parameters are passed to `Stokes`.

4.6.4 Abstract base classes

In addition to the `Stokes` abstract base class, the following classes can also be used for implementing custom drag classes.

StokesSphereWall

class ott.drag.**StokesSphereWall**(*varargin*)

Abstract base class for sphere-wall drag calculation methods. Inherits from *Stokes*.

Properties

- radius – Radius of sphere
- separation – Distance from sphere centre to wall
- viscosity – Viscosity of medium
- inverse – Calculated from *forward*

Abstract properties

- forward – Drag tensor calculated by method

Static methods

- FromShape – Construct a drag tensor from a shape array

Supported casts

- ott.shape.Shape – Cast to plane and sphere

static FromShape(*shape*, *varargin*)

Attempt to choose a sphere-wall drag method based on a shape array

Usage drag = StokesSphereWall.FromShape(shape, ...)

Parameters

- shape (ott.shape.Shape) – Array of shapes. Must be two elements, one of which must be a plane.

Additional parameters are passed to corresponding class constructor.

StokesSphereWall(*varargin*)

Construct base class for sphere-wall methods

Usage drag = drag@ott.drag.StokesSphereWall(radius, separation, viscosity, ...)

Parameters

- radius (numeric) – Particle radius (default: 1.0)
- separation (numeric) – Separation (default: Inf)
- viscosity (numeric) – Viscosity (default: 1.0)

Parameters can also be passed as named arguments. Unmatched parameters are passed to *Stokes*.

StokesStarShaped

class ott.drag.**StokesStarShaped**(*varargin*)

Abstract base class for star shaped particle methods in an unbounded medium. Inherits from *Stokes*.

Properties

- inverse – Calculated from forward
- viscosity – Viscosity of medium
- shape – A star shaped particle describing the geometry

Abstract properties

- forwardInternal – Drag tensor calculated by method

Static methods FromShape – Defers to StokesLambNn (may change in future)

Supported casts

- ott.shape.Shape – Retrieves the geometry (see *shape*)

`static FromShape`(*shape*, *varargin*)

Construct drag tensor for star shaped particle

Usage StokesStarShaped.FromShape(shape, ...)

Passes all arguments to `StokesLambNn`.

`StokesStarShaped`(*varargin*)

Construct base class for star-shaped particle methods

Usage drag = drag@ott.drag.StokesStarShaped(shape, viscosity, ...)

Parameters

- shape (ott.shape.Shape) – A shape object with a starShaped property with the *true* value.
- viscosity (numeric) – Viscosity of medium (default: 1.0)

Parameters can also be passed as named arguments. Unmatched parameters are passed to `Stokes`.

4.7 *dynamics* Package

The *dynamics* package implements fixed step size methods for modelling particle dynamics. This package is a work-in-progress and content may change in a future release.

4.7.1 Classes

Dynamics

Isolated

WallEffect

Todo: todo dynamics error

4.8 *tools* Package

The *tools* package contains a collection of tools for working with optical tweezers simulations including methods for finding trap locations or calculating/visualising velocity fields.

Todo: tools package

class `ott.tools.`**`FindTraps1d`**

Descriptions and method for finding traps in 1-dimension.

This class contains static methods for finding traps in one dimension. Instances of the class simply describe properties of traps, such as trap location, depth, stiffness.

Units of properties depend on the method that created the FindTraps1d instance (see documentation for static methods).

Properties

- position – Trap position
- stiffness – Trap stiffness at equilibrium (force/position units)
- depth – Optical trap depth (force units)
- range – Range of optical trap (position units)
- minforce – Minimum force in trap (force units)
- maxforce – Maximum force in trap (force units)
- minposition – Position of minimum force (position units)
- maxposition – Position of maximum force (position units)
- globalStiffness – Stiffness between min/max force locations

Methods

- groupStable – Groups stable traps based on trap depth
- plot – Generate a plot of the traps in the array

Static methods

- FromArray – Find traps from an array of force/position data
- Bisection – Uses bisection method to find trap position
- Newton – uses Newton's method to find trap position

4.9 *utils* Packages

The *utils* packages contains utility functions using throughout the toolbox.

Todo: utils packages docs

`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]

4.10 *ui* Package

The *ui* package contains the graphical user interfaces for the toolbox.

Todo: Finish ui package docs

class `ott.ui.Launcher`

Displays a list of graphical user interfaces included with OTT.

Usage ott.ui.Launcher() – Creates a new instance.

ott.ui.Launcher.LaunchOrPresent() – Creates a new instance of presents an existing instance.

4.11 Errors and Warnings

This page lists common errors and warnings generated by the toolbox and tips on how to resolve them.

Todo: Errors and warnings

CHAPTER

FIVE

CONCEPTUAL NOTES

Todo: Should all these sections stay here???

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.

`images/Conception/vswfField.png`

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.

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.

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.

`images/Conception/translations.png`

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.

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.

`images/Conception/squareSphere.png`

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.

`images/Conception/angularScaling.png`

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');
```

images/Conception/truncationAngle.png

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

G

H

I

L

M

N

O

P

R

S

T

U

V

W