
odtbrain Documentation

Release 0.4.6

Paul Müller

Oct 19, 2022

CONTENTS

1	Introduction	3
1.1	Installation	3
1.2	Theoretical background	3
1.3	Fields of Application	3
1.4	How to cite	4
2	Code reference	5
2.1	Data conversion methods	5
2.1.1	Sinogram preparation	5
2.1.2	Translation of object function to refractive index	6
2.2	2D inversion	7
2.2.1	Method summary	8
2.2.2	Backpropagation	8
2.2.3	Fourier mapping	9
2.2.4	Direct sum	10
2.3	3D inversion	11
2.3.1	Method summary	12
2.3.2	Backpropagation	12
2.3.3	Backpropagation with tilted axis of rotation	14
2.4	3D Apple core correction	16
3	Examples	19
3.1	2D examples	19
3.1.1	Mie off-center cylinder	19
3.1.2	Mie cylinder with unevenly spaced angles	22
3.1.3	Mie cylinder with incomplete angular coverage	25
3.1.4	FDTD cell phantom	29
3.2	3D examples	31
3.2.1	Missing apple core correction	31
3.2.2	HL60 cell	34
3.2.3	FDTD cell phantom	36
3.2.4	FDTD cell phantom with tilted axis of rotation	38
3.2.5	FDTD cell phantom with tilted and rolled axis of rotation	41
3.2.6	Mie sphere	44
4	Changelog	47
4.1	version 0.4.6	47
4.2	version 0.4.5	47
4.3	version 0.4.4	47
4.4	version 0.4.3	47

4.5	version 0.4.2	48
4.6	version 0.4.1	48
4.7	version 0.4.0	48
4.8	version 0.3.0	48
4.9	version 0.2.6	48
4.10	version 0.2.5	49
4.11	version 0.2.4	49
4.12	version 0.2.3	49
4.13	version 0.2.2	49
4.14	version 0.2.1	49
4.15	version 0.2.0	49
4.16	version 0.1.8	50
4.17	version 0.1.7	50
4.18	version 0.1.6	50
4.19	version 0.1.5	50
4.20	version 0.1.4	50
4.21	version 0.1.3	51
4.22	version 0.1.2	51
4.23	version 0.1.1	51
5	Bibliography	53
6	Indices and tables	55
	Bibliography	57
	Python Module Index	59
	Index	61

ODTbrain provides image reconstruction algorithms for Optical Diffraction Tomography with a Born and Rytov Approximation-based Inversion to compute the refractive index (n) in 2D and in 3D. This is the documentaion of ODTbrain version 0.4.6.

INTRODUCTION

This package provides reconstruction algorithms for diffraction tomography in two and three dimensions.

1.1 Installation

To install via the [Python Package Index \(PyPI\)](#), run:

```
pip install odtbrain
```

On some systems, the [FFTW3 library](#) might have to be installed manually before installing ODTbrain. All other dependencies are installed automatically. If the above command does not work, please refer to the installation instructions at the [GitHub repository](#) or [create an issue](#)

1.2 Theoretical background

A detailed summary of the underlying theory is available in [\[MSG15b\]](#).

The Fourier diffraction theorem states, that the Fourier transform $\hat{U}_{B,\phi_0}(\mathbf{k}_D)$ of the scattered field $u_B(\mathbf{r}_D)$, measured at a certain angle ϕ_0 , is distributed along a circular arc (2D) or along a semi-spherical surface (3D) in Fourier space, synthesizing the Fourier transform $\hat{F}(\mathbf{k})$ of the object function $f(\mathbf{r})$ [\[KS01\]](#), [\[Wol69\]](#).

$$\hat{F}(k_m(\mathbf{s} - \mathbf{s}_0)) = -\sqrt{\frac{2}{\pi}} \frac{ik_m}{a_0} M \hat{U}_{B,\phi_0}(\mathbf{k}_D) \exp(-ik_m M l_D)$$

In this notation, k_m is the wave number, \mathbf{s}_0 is the norm vector pointing at ϕ_0 , $M = \sqrt{1 - s_x^2}$ (2D) and $M = \sqrt{1 - s_x^2 - s_y^2}$ (3D) enforces the spherical constraint, and l_D is the distance from the center of the object function $f(\mathbf{r})$ to the detector plane \mathbf{r}_D .

1.3 Fields of Application

The algorithms presented here are based on the (scalar) Helmholtz equation. Furthermore, the Born and Rytov approximations to the scattered wave $u(\mathbf{r})$ are used to linearize the problem for a straight-forward inversion.

The package is intended for optical diffraction tomography to determine the refractive index of biological cells. Because the Helmholtz equation is only an approximation to the Maxwell equations, describing the propagation of light, FDTD (Finite Difference Time Domain) simulations were performed to test the reconstruction algorithms within this package. The algorithms present in this package should also be valid for the following cases, but have not been tested appropriately:

- tomographic measurements of absorbing materials (complex refractive index $n(\mathbf{r})$)
- ultrasonic diffraction tomography, which is correctly described by the Helmholtz equation

1.4 How to cite

If you use ODTbrain in a scientific publication, please cite Müller et al., *BMC Bioinformatics* (2015) [[MSG15a](#)].

CODE REFERENCE

2.1 Data conversion methods

<code>odt_to_ri(f, res, nm)</code>	Convert the ODT object function to refractive index
<code>opt_to_ri(f, res, nm)</code>	Convert the OPT object function to refractive index
<code>sinogram_as_radon(uSin[, align])</code>	Compute the phase from a complex wave field sinogram
<code>sinogram_as_rytov(uSin[, u0, align])</code>	Convert the complex wave field sinogram to the Rytov phase

2.1.1 Sinogram preparation

Tomographic data sets consist of detector images for different rotational positions ϕ_0 of the object. Sinogram preparation means that the measured field $u(\mathbf{r})$ is transformed to either the Rytov approximation (diffraction tomography) or the Radon phase (classical tomography).

`odtbrain.sinogram_as_radon(uSin, align=True)`

Compute the phase from a complex wave field sinogram

This step is essential when using the ray approximation before computation of the refractive index with the inverse Radon transform.

Parameters

- **uSin** (*2d or 3d complex ndarray*) – The background-corrected sinogram of the complex scattered wave $u(\mathbf{r})/u_0(\mathbf{r})$. The first axis iterates through the angles ϕ_0 .
- **align** (*bool*) – Tries to correct for a phase offset in the phase sinogram.

Returns *phase* – The unwrapped phase array corresponding to *uSin*.

Return type 2d or 3d real ndarray

See also:

`skimage.restoration.unwrap_phase` phase unwrapping

`radontea.backproject_3d` e.g. reconstruction via backprojection

`odtbrain.sinogram_as_rytov(uSin, u0=1, align=True)`

Convert the complex wave field sinogram to the Rytov phase

This method applies the Rytov approximation to the recorded complex wave sinogram. To achieve this, the following filter is applied:

$$u_B(\mathbf{r}) = u_0(\mathbf{r}) \ln \left(\frac{u_R(\mathbf{r})}{u_0(\mathbf{r})} + 1 \right)$$

This filter step effectively replaces the Born approximation $u_B(\mathbf{r})$ with the Rytov approximation $u_R(\mathbf{r})$, assuming that the scattered field is equal to $u(\mathbf{r}) \approx u_R(\mathbf{r}) + u_0(\mathbf{r})$.

Parameters

- **uSin** (*2d or 3d complex ndarray*) – The sinogram of the complex wave $u_R(\mathbf{r}) + u_0(\mathbf{r})$. The first axis iterates through the angles ϕ_0 .
- **u0** (*ndarray of dimension as uSin or less, or int.*) – The incident plane wave $u_0(\mathbf{r})$ at the detector. If *u0* is “1”, it is assumed that the data is already background-corrected ($uSin = \frac{u_R(\mathbf{r})}{u_0(\mathbf{r})} + 1$). Note that if the reconstruction distance l_D of the original experiment is non-zero and *u0* is set to 1, then the reconstruction will be wrong; the field is not focused to the center of the reconstruction volume.
- **align** (*bool*) – Tries to correct for a phase offset in the phase sinogram.

Returns **uB** – The Rytov-filtered complex sinogram $u_B(\mathbf{r})$.

Return type 2d or 3d real ndarray

See also:

`skimage.restoration.unwrap_phase` phase unwrapping

2.1.2 Translation of object function to refractive index

To obtain the refractive index map $n(\mathbf{r})$ from an object function $f(\mathbf{r})$ returned by e.g. `backpropagate_3d()`, an additional conversion step is necessary. For diffraction based models, `odt_to_ri()` must be used whereas for Radon-based models `opt_to_ri()` must be used.

`odtbrain.odt_to_ri(f, res, nm)`

Convert the ODT object function to refractive index

In ODT (Optical Diffraction Tomography), the object function is defined by the Helmholtz equation

$$f(\mathbf{r}) = k_m^2 \left[\left(\frac{n(\mathbf{r})}{n_m} \right)^2 - 1 \right]$$

with $k_m = \frac{2\pi n_m}{\lambda}$. By inverting this equation, we obtain the refractive index $n(\mathbf{r})$.

$$n(\mathbf{r}) = n_m \sqrt{\frac{f(\mathbf{r})}{k_m^2} + 1}$$

Parameters

- **f** (*n-dimensional ndarray*) – The reconstructed object function $f(\mathbf{r})$.
- **res** (*float*) – The size of the vacuum wave length λ in pixels.
- **nm** (*float*) – The refractive index of the medium n_m that surrounds the object in $f(\mathbf{r})$.

Returns **ri** – The complex refractive index $n(\mathbf{r})$.

Return type n-dimensional ndarray

Notes

Because this function computes the root of a complex number, there are several solutions to the refractive index. Always the positive (real) root of the refractive index is used.

`odtbrain.opt_to_ri(f, res, nm)`

Convert the OPT object function to refractive index

In OPT (Optical Projection Tomography), the object function is computed from the raw phase data. This method converts phase data to refractive index data.

$$n(\mathbf{r}) = n_m + \frac{f(\mathbf{r}) \cdot \lambda}{2\pi}$$

Parameters

- **f** (*n-dimensional ndarray*) – The reconstructed object function $f(\mathbf{r})$.
- **res** (*float*) – The size of the vacuum wave length λ in pixels.
- **nm** (*float*) – The refractive index of the medium n_m that surrounds the object in $f(\mathbf{r})$.

Returns **ri** – The complex refractive index $n(\mathbf{r})$.

Return type *n-dimensional ndarray*

Notes

This function is not meant to be used with diffraction tomography data. For ODT, use `odt_to_ri()` instead.

2.2 2D inversion

The first Born approximation for a 2D scattering problem with a plane wave $u_0(\mathbf{r}) = a_0 \exp(-ik_m \mathbf{s}_0 \mathbf{r})$ reads:

$$u_B(\mathbf{r}) = \iint d^2 r' G(\mathbf{r} - \mathbf{r}') f(\mathbf{r}') u_0(\mathbf{r}')$$

The Green's function in 2D is the zero-order Hankel function of the first kind:

$$G(\mathbf{r} - \mathbf{r}') = \frac{i}{4} H_0^{(1)}(k_m |\mathbf{r} - \mathbf{r}'|)$$

Solving for $f(\mathbf{r})$ yields the Fourier diffraction theorem in 2D

$$\hat{F}(k_m(\mathbf{s} - \mathbf{s}_0)) = -\sqrt{\frac{2}{\pi}} \frac{ik_m}{a_0} M \hat{U}_{B,\phi_0}(k_{Dx}) \exp(-ik_m M l_D)$$

where $\hat{F}(k_x, k_z)$ is the Fourier transformed object function and $\hat{U}_{B,\phi_0}(k_{Dx})$ is the Fourier transformed complex wave that travels along \mathbf{s}_0 (in the direction of ϕ_0) measured at the detector \mathbf{r}_D .

The following identities are used:

$$\begin{aligned} k_m(\mathbf{s} - \mathbf{s}_0) &= k_{Dx} \mathbf{t}_\perp + k_m(M - 1) \mathbf{s}_0 \\ \mathbf{s}_0 &= (p_0, M_0) = (-\sin \phi_0, \cos \phi_0) \\ \mathbf{t}_\perp &= (-M_0, p_0) = (\cos \phi_0, \sin \phi_0) \end{aligned}$$

2.2.1 Method summary

<code>backpropagate_2d(uSin, angles, res, nm[, ...])</code>	2D backpropagation with the Fourier diffraction theorem
<code>fourier_map_2d(uSin, angles, res, nm[, lD, ...])</code>	2D Fourier mapping with the Fourier diffraction theorem
<code>integrate_2d(uSin, angles, res, nm[, lD, ...])</code>	(slow) 2D reconstruction with the Fourier diffraction theorem

2.2.2 Backpropagation

`odtbrain.backpropagate_2d(uSin, angles, res, nm, lD=0, coords=None, weight_angles=True, onlyreal=False, padding=True, padval=0, count=None, max_count=None, verbose=0)`

2D backpropagation with the Fourier diffraction theorem

Two-dimensional diffraction tomography reconstruction algorithm for scattering of a plane wave $u_0(\mathbf{r}) = u_0(x, z)$ by a dielectric object with refractive index $n(x, z)$.

This method implements the 2D backpropagation algorithm [MSG15b].

$$f(\mathbf{r}) = -\frac{ik_m}{2\pi} \sum_{j=1}^N \Delta\phi_0 D_{-\phi_j} \left\{ \text{FFT}_{1D}^{-1} \left\{ |k_{Dx}| \frac{\text{FFT}_{1D} \{u_{B,\phi_j}(x_D)\}}{u_0(l_D)} \exp[ik_m(M-1) \cdot (z_{\phi_j} - l_D)] \right\} \right\}$$

with the forward FFT_{1D} and inverse FFT_{1D}^{-1} 1D fast Fourier transform, the rotational operator $D_{-\phi_j}$, the angular distance between the projections $\Delta\phi_0$, the ramp filter in Fourier space $|k_{Dx}|$, and the propagation distance $(z_{\phi_j} - l_D)$.

Parameters

- **uSin** (`(A, N) ndarray`) – Two-dimensional sinogram of line recordings $u_{B,\phi_j}(x_D)$ divided by the incident plane wave $u_0(l_D)$ measured at the detector.
- **angles** (`(A,) ndarray`) – Angular positions ϕ_j of *uSin* in radians.
- **res** (`float`) – Vacuum wavelength of the light λ in pixels.
- **nm** (`float`) – Refractive index of the surrounding medium n_m .
- **lD** (`float`) – Distance from center of rotation to detector plane l_D in pixels.
- **coords** (`None [(2, M) ndarray]`) – Computes only the output image at these coordinates. This keyword is reserved for future versions and is not implemented yet.
- **weight_angles** (`bool`) – If *True*, weights each backpropagated projection with a factor proportional to the angular distance between the neighboring projections.

$$\Delta\phi_0 \mapsto \Delta\phi_j = \frac{\phi_{j+1} - \phi_{j-1}}{2}$$

New in version 0.1.1.

- **onlyreal** (`bool`) – If *True*, only the real part of the reconstructed image will be returned. This saves computation time.
- **padding** (`bool`) – Pad the input data to the second next power of 2 before Fourier transforming. This reduces artifacts and speeds up the process for input image sizes that are not powers of 2.
- **padval** (`float`) – The value used for padding. This is important for the Rytov approximation, where an approximate zero in the phase might translate to $2i$ due to the unwrapping algorithm. In that case, this value should be a multiple of $2i$. If *padval* is *None*, then the edge values are used for padding (see documentation of `numpy.pad()`).

- **count** (multiprocessing.Value or *None*) – Can be used to monitor the progress of the algorithm. Initially, the value of *max_count.value* is incremented by the total number of steps. At each step, the value of *count.value* is incremented.
- **max_count** (multiprocessing.Value or *None*) – Can be used to monitor the progress of the algorithm. Initially, the value of *max_count.value* is incremented by the total number of steps. At each step, the value of *count.value* is incremented.
- **verbose** (*int*) – Increment to increase verbosity.

Returns **f** – Reconstructed object function $f(\mathbf{r})$ as defined by the Helmholtz equation. $f(x, z) = k_m^2 \left(\left(\frac{n(x, z)}{n_m} \right)^2 - 1 \right)$

Return type ndarray of shape (N,N), complex if *onlyreal* is *False*

See also:

odt_to_ri conversion of the object function $f(\mathbf{r})$ to refractive index $n(\mathbf{r})$

radontea.backproject backprojection based on the Fourier slice theorem

Notes

Do not use the parameter *ID* in combination with the Rytov approximation - the propagation is not correctly described. Instead, numerically refocus the sinogram prior to converting it to Rytov data (using e.g. [odtbrain.sinogram_as_rytov\(\)](#)) with a numerical focusing algorithm (available in the Python package *nrefocus*).

2.2.3 Fourier mapping

odtbrain.fourier_map_2d(*uSin*, *angles*, *res*, *nm*, *ID*=0, *semi_coverage*=*False*, *coords*=*None*, *count*=*None*, *max_count*=*None*, *verbose*=0)

2D Fourier mapping with the Fourier diffraction theorem

Two-dimensional diffraction tomography reconstruction algorithm for scattering of a plane wave $u_0(\mathbf{r}) = u_0(x, z)$ by a dielectric object with refractive index $n(x, z)$.

This function implements the solution by interpolation in Fourier space.

Parameters

- **uSin** ((*A*, *N*) ndarray) – Two-dimensional sinogram of line recordings $u_{B, \phi_j}(x_D)$ divided by the incident plane wave $u_0(l_D)$ measured at the detector.
- **angles** ((*A*,) ndarray) – Angular positions ϕ_j of *uSin* in radians.
- **res** (*float*) – Vacuum wavelength of the light λ in pixels.
- **nm** (*float*) – Refractive index of the surrounding medium n_m .
- **ID** (*float*) – Distance from center of rotation to detector plane l_D in pixels.
- **semi_coverage** (*bool*) – If set to *True*, it is assumed that the sinogram does not necessarily cover the full angular range from 0 to 2, but an equidistant coverage over 2 can be achieved by inferring point (anti)symmetry of the (imaginary) real parts of the Fourier transform of *f*. Valid for any set of angles {X} that result in a 2 coverage with the union set {X}U{X+}.
- **coords** (*None* [(*2*, *M*) ndarray]) – Computes only the output image at these coordinates. This keyword is reserved for future versions and is not implemented yet.

- **count** (multiprocessing.Value or *None*) – Can be used to monitor the progress of the algorithm. Initially, the value of *max_count.value* is incremented by the total number of steps. At each step, the value of *count.value* is incremented.
- **max_count** (multiprocessing.Value or *None*) – Can be used to monitor the progress of the algorithm. Initially, the value of *max_count.value* is incremented by the total number of steps. At each step, the value of *count.value* is incremented.
- **verbose** (*int*) – Increment to increase verbosity.

Returns **f** – Reconstructed object function $f(\mathbf{r})$ as defined by the Helmholtz equation. $f(x, z) = k_m^2 \left(\left(\frac{n(x, z)}{n_m} \right)^2 - 1 \right)$

Return type ndarray of shape (N,N), complex if *onlyreal* is *False*

See also:

[*backpropagate_2d*](#) implementation by backpropagation

[*odt_to_ri*](#) conversion of the object function $f(\mathbf{r})$ to refractive index $n(\mathbf{r})$

Notes

Do not use the parameter *ID* in combination with the Rytov approximation - the propagation is not correctly described. Instead, numerically refocus the sinogram prior to converting it to Rytov data (using e.g. [*odtbrain.sinogram_as_rytov\(\)*](#)) with a numerical focusing algorithm (available in the Python package *nrefocus*).

The interpolation in Fourier space (which is done with [*scipy.interpolate.griddata\(\)*](#)) may be unstable and lead to artifacts if the data to interpolate contains sharp spikes. This issue is not handled at all by this method (in fact, a test has been removed in version 0.2.6 because *griddata* gave different results on Windows and Linux).

2.2.4 Direct sum

`odtbrain.integrate_2d(uSin, angles, res, nm, ID=0, coords=None, count=None, max_count=None, verbose=0)`
(slow) 2D reconstruction with the Fourier diffraction theorem

Two-dimensional diffraction tomography reconstruction algorithm for scattering of a plane wave $u_0(\mathbf{r}) = u_0(x, z)$ by a dielectric object with refractive index $n(x, z)$.

This function implements the solution by summation in real space, which is extremely slow.

Parameters

- **uSin** ((*A*, *N*) ndarray) – Two-dimensional sinogram of line recordings $u_{B, \phi_j}(x_D)$ divided by the incident plane wave $u_0(l_D)$ measured at the detector.
- **angles** ((*A*,) ndarray) – Angular positions ϕ_j of *uSin* in radians.
- **res** (*float*) – Vacuum wavelength of the light λ in pixels.
- **nm** (*float*) – Refractive index of the surrounding medium n_m .
- **ID** (*float*) – Distance from center of rotation to detector plane l_D in pixels.
- **coords** (*None* or (*2*, *M*) ndarray) – Computes only the output image at these coordinates. This keyword is reserved for future versions and is not implemented yet.

- **count** (multiprocessing.Value or *None*) – Can be used to monitor the progress of the algorithm. Initially, the value of *max_count.value* is incremented by the total number of steps. At each step, the value of *count.value* is incremented.
- **max_count** (multiprocessing.Value or *None*) – Can be used to monitor the progress of the algorithm. Initially, the value of *max_count.value* is incremented by the total number of steps. At each step, the value of *count.value* is incremented.
- **verbose** (*int*) – Increment to increase verbosity.

Returns **f** – Reconstructed object function $f(\mathbf{r})$ as defined by the Helmholtz equation. $f(x, z) = k_m^2 \left(\left(\frac{n(x, z)}{n_m} \right)^2 - 1 \right)$

Return type ndarray of shape (N,N), complex if *onlyreal* is *False*

See also:

[*backpropagate_2d*](#) implementation by backprojection

[*fourier_map_2d*](#) implementation by Fourier interpolation

[*odt_to_ri*](#) conversion of the object function $f(\mathbf{r})$ to refractive index $n(\mathbf{r})$

Notes

This method is not meant for production use. The computation time is very long and the reconstruction quality is bad. This function is included in the package, because of its educational value, exemplifying the backpropagation algorithm.

Do not use the parameter *ID* in combination with the Rytov approximation - the propagation is not correctly described. Instead, numerically refocus the sinogram prior to converting it to Rytov data (using e.g. [*odtbrain.sinogram_as_rytov\(\)*](#)) with a numerical focusing algorithm (available in the Python package *nrefocus*).

2.3 3D inversion

The first Born approximation for a 3D scattering problem with a plane wave $u_0(\mathbf{r}) = a_0 \exp(-ik_m \mathbf{s}_0 \mathbf{r})$ reads:

$$u_B(\mathbf{r}) = \iiint d^3 r' G(\mathbf{r} - \mathbf{r}') f(\mathbf{r}') u_0(\mathbf{r}')$$

The Green's function in 3D can be written as:

$$G(\mathbf{r} - \mathbf{r}') = \frac{ik_m}{8\pi^2} \iint dp dq \frac{1}{M} \exp\{ik_m [p(x - x') + q(y - y') + M(z - z')]\}$$

with

$$M = \sqrt{1 - p^2 - q^2}$$

Solving for $f(\mathbf{r})$ yields the Fourier diffraction theorem in 3D

$$\hat{F}(k_m(\mathbf{s} - \mathbf{s}_0)) = -\sqrt{\frac{2}{\pi}} \frac{ik_m}{a_0} M \hat{U}_{B, \phi_0}(k_{Dx}, k_{Dy}) \exp(-ik_m M l_D)$$

where $\hat{F}(k_x, k_y, k_z)$ is the Fourier transformed object function and $\hat{U}_{B, \phi_0}(k_{Dx}, k_{Dy})$ is the Fourier transformed complex wave that travels along \mathbf{s}_0 (in the direction of ϕ_0) measured at the detector \mathbf{r}_D .

The following identities are used:

$$\begin{aligned} k_m(\mathbf{s} - \mathbf{s}_0) &= k_{Dx} \mathbf{t}_\perp + k_m(M - 1) \mathbf{s}_0 \\ \mathbf{s} &= (p, q, M) \\ \mathbf{s}_0 &= (p_0, q_0, M_0) = (-\sin \phi_0, 0, \cos \phi_0) \\ \mathbf{t}_\perp &= \left(\cos \phi_0, \frac{k_{Dy}}{k_{Dx}}, \sin \phi_0 \right)^\top \end{aligned}$$

2.3.1 Method summary

<code>backpropagate_3d(uSin, angles, res, nm[, ...])</code>	3D backpropagation
<code>backpropagate_3d_tilted(uSin, angles, res, nm)</code>	3D backpropagation with a tilted axis of rotation

2.3.2 Backpropagation

`odtbrain.backpropagate_3d(uSin, angles, res, nm, lD=0, coords=None, weight_angles=True, onlyreal=False, padding=(True, True), padfac=1.75, padval='edge', intp_order=2, dtype=None, num_cores=2, save_memory=False, copy=True, count=None, max_count=None, verbose=0)`

3D backpropagation

Three-dimensional diffraction tomography reconstruction algorithm for scattering of a plane wave $u_0(\mathbf{r}) = u_0(x, y, z)$ by a dielectric object with refractive index $n(x, y, z)$.

This method implements the 3D backpropagation algorithm [MSG15b].

$$f(\mathbf{r}) = -\frac{ik_m}{2\pi} \sum_{j=1}^N \Delta\phi_0 D_{-\phi_j} \left\{ \text{FFT}_{2D}^{-1} \left\{ |k_{Dx}| \frac{\text{FFT}_{2D} \{u_{B,\phi_j}(x_D, y_D)\}}{u_0(l_D)} \exp[ik_m(M-1) \cdot (z_{\phi_j} - l_D)] \right\} \right\}$$

with the forward FFT_{2D} and inverse FFT_{2D}^{-1} 2D fast Fourier transform, the rotational operator $D_{-\phi_j}$, the angular distance between the projections $\Delta\phi_0$, the ramp filter in Fourier space $|k_{Dx}|$, and the propagation distance $(z_{\phi_j} - l_D)$.

Parameters

- **uSin** ((*A*, *Ny*, *Nx*) *ndarray*) – Three-dimensional sinogram of plane recordings $u_{B,\phi_j}(x_D, y_D)$ divided by the incident plane wave $u_0(l_D)$ measured at the detector.
- **angles** ((*A*,) *ndarray*) – Angular positions ϕ_j of *uSin* in radians.
- **res** (*float*) – Vacuum wavelength of the light λ in pixels.
- **nm** (*float*) – Refractive index of the surrounding medium n_m .
- **lD** (*float*) – Distance from center of rotation to detector plane l_D in pixels.
- **coords** (*None* [(3, *M*) *ndarray*]) – Only compute the output image at these coordinates. This keyword is reserved for future versions and is not implemented yet.
- **weight_angles** (*bool*) – If *True*, weights each backpropagated projection with a factor proportional to the angular distance between the neighboring projections.

$$\Delta\phi_0 \mapsto \Delta\phi_j = \frac{\phi_{j+1} - \phi_{j-1}}{2}$$

New in version 0.1.1.

- **onlyreal** (*bool*) – If *True*, only the real part of the reconstructed image will be returned. This saves computation time.
- **padding** (*tuple of bool*) – Pad the input data to the second next power of 2 before Fourier transforming. This reduces artifacts and speeds up the process for input image sizes that are not powers of 2. The default is padding in x and y: *padding*=(*True*, *True*). For padding only in x-direction (e.g. for cylindrical symmetries), set *padding* to (*True*, *False*). To turn off padding, set it to (*False*, *False*).
- **padfac** (*float*) – Increase padding size of the input data. A value greater than one will trigger padding to the second-next power of two. For example, a value of 1.75 will lead to a padded size of 256 for an initial size of 144, whereas it will lead to a padded size of 512 for an initial size of 150. Values greater than 2 are allowed. This parameter may greatly increase memory usage!
- **padval** (*float or "edge"*) – The value used for padding. This is important for the Rytov approximation, where an approximate zero in the phase might translate to $2i$ due to the unwrapping algorithm. In that case, this value should be a multiple of $2i$. If *padval* is “edge”, then the edge values are used for padding (see documentation of `numpy.pad()`). If *padval* is a float, then padding is done with a linear ramp.
- **interp_order** (*int between 0 and 5*) – Order of the interpolation for rotation. See `scipy.ndimage.rotate()` for details.
- **dtype** (*dtype object or argument for numpy.dtype()*) – The data type that is used for calculations (float or double). Defaults to `numpy.float_`.
- **num_cores** (*int*) – The number of cores to use for parallel operations. This value defaults to the number of cores on the system.
- **save_memory** (*bool*) – Saves memory at the cost of longer computation time.
New in version 0.1.5.
- **copy** (*bool*) – Copy input sinogram *uSin* for data processing. If *copy* is set to *False*, then *uSin* will be overridden.
New in version 0.1.5.
- **count** (*multiprocessing.Value or None*) – Can be used to monitor the progress of the algorithm. Initially, the value of *max_count.value* is incremented by the total number of steps. At each step, the value of *count.value* is incremented.
- **max_count** (*multiprocessing.Value or None*) – Can be used to monitor the progress of the algorithm. Initially, the value of *max_count.value* is incremented by the total number of steps. At each step, the value of *count.value* is incremented.
- **verbose** (*int*) – Increment to increase verbosity.

Returns **f** – Reconstructed object function $f(\mathbf{r})$ as defined by the Helmholtz equation. $f(x, z) =$

$$k_m^2 \left(\left(\frac{n(x, z)}{n_m} \right)^2 - 1 \right)$$

Return type ndarray of shape (Nx, Ny, Nx), complex if *onlyreal*==*False*

See also:

odt_to_ri conversion of the object function $f(\mathbf{r})$ to refractive index $n(\mathbf{r})$

Notes

Do not use the parameter *lD* in combination with the Rytov approximation - the propagation is not correctly described. Instead, numerically refocus the sinogram prior to converting it to Rytov data (using e.g. [odtbrain.sinogram_as_rytov\(\)](#)) with a numerical focusing algorithm (available in the Python package [nrefocus](#)).

2.3.3 Backpropagation with tilted axis of rotation

```
odtbrain.backpropagate_3d_tilted(uSin, angles, res, nm, lD=0, tilted_axis=[0, 1, 0], coords=None,
                                weight_angles=True, onlyreal=False, padding=(True, True),
                                padfac=1.75, padval='edge', intp_order=2, dtype=None, num_cores=2,
                                save_memory=False, copy=True, count=None, max_count=None,
                                verbose=0)
```

3D backpropagation with a tilted axis of rotation

Three-dimensional diffraction tomography reconstruction algorithm for scattering of a plane wave $u_0(\mathbf{r}) = u_0(x, y, z)$ by a dielectric object with refractive index $n(x, y, z)$.

This method implements the 3D backpropagation algorithm with a rotational axis that is tilted by θ_{tilt} w.r.t. the imaging plane [MSCG15].

$$f(\mathbf{r}) = -\frac{ik_m}{2\pi} \sum_{j=1}^N \Delta\phi_0 D_{-\phi_j}^{\text{tilt}} \left\{ \text{FFT}_{2D}^{-1} \left\{ |k_{Dx} \cdot \cos \theta_{\text{tilt}}| \frac{\text{FFT}_{2D} \{u_{B,\phi_j}(x_D, y_D)\}}{u_0(l_D)} \exp[ik_m(M-1) \cdot (z_{\phi_j} - l_D)] \right\} \right\}$$

with a modified rotational operator $D_{-\phi_j}^{\text{tilt}}$ and a different filter in Fourier space $|k_{Dx} \cdot \cos \theta_{\text{tilt}}|$ when compared to [backpropagate_3d\(\)](#).

New in version 0.1.2.

Parameters

- **uSin** ((*A*, *Ny*, *Nx*) *ndarray*) – Three-dimensional sinogram of plane recordings $u_{B,\phi_j}(x_D, y_D)$ divided by the incident plane wave $u_0(l_D)$ measured at the detector.
- **angles** (*ndarray of shape (A,3) or 1D array of length A*) – If the shape is (A,3), then *angles* consists of vectors on the unit sphere that correspond to the direction of illumination and acquisition (s_0). If the shape is (A,), then *angles* is a one-dimensional array of angles in radians that determines the angular position ϕ_j . In both cases, *tilted_axis* must be set according to the tilt of the rotational axis.
- **res** (*float*) – Vacuum wavelength of the light λ in pixels.
- **nm** (*float*) – Refractive index of the surrounding medium n_m .
- **lD** (*float*) – Distance from center of rotation to detector plane l_D in pixels.
- **tilted_axis** (*list of floats*) – The coordinates [x, y, z] on a unit sphere representing the tilted axis of rotation. The default is (0,1,0), which corresponds to a rotation about the y-axis and follows the behavior of [odtbrain.backpropagate_3d\(\)](#).
- **coords** (*None [(3, M) ndarray]*) – Only compute the output image at these coordinates. This keyword is reserved for future versions and is not implemented yet.
- **weight_angles** (*bool*) – If *True*, weights each backpropagated projection with a factor proportional to the angular distance between the neighboring projections.

$$\Delta\phi_0 \mapsto \Delta\phi_j = \frac{\phi_{j+1} - \phi_{j-1}}{2}$$

This currently only works when *angles* has the shape (A,).

- **onlyreal** (*bool*) – If *True*, only the real part of the reconstructed image will be returned. This saves computation time.
- **padding** (*tuple of bool*) – Pad the input data to the second next power of 2 before Fourier transforming. This reduces artifacts and speeds up the process for input image sizes that are not powers of 2. The default is padding in x and y: *padding=(True, True)*. For padding only in x-direction (e.g. for cylindrical symmetries), set *padding* to *(True, False)*. To turn off padding, set it to *(False, False)*.
- **padfac** (*float*) – Increase padding size of the input data. A value greater than one will trigger padding to the second-next power of two. For example, a value of 1.75 will lead to a padded size of 256 for an initial size of 144, whereas it will lead to a padded size of 512 for an initial size of 150. Values greater than 2 are allowed. This parameter may greatly increase memory usage!
- **padval** (*float or "edge"*) – The value used for padding. This is important for the Rytov approximation, where an approximate zero in the phase might translate to $2i$ due to the unwrapping algorithm. In that case, this value should be a multiple of $2i$. If *padval* is “edge”, then the edge values are used for padding (see documentation of `numpy.pad()`). If *padval* is a float, then padding is done with a linear ramp.
- **interp_order** (*int between 0 and 5*) – Order of the interpolation for rotation. See `scipy.ndimage.affine_transform()` for details.
- **dtype** (*dtype object or argument for numpy.dtype()*) – The data type that is used for calculations (float or double). Defaults to *numpy.float_*.
- **num_cores** (*int*) – The number of cores to use for parallel operations. This value defaults to the number of cores on the system.
- **save_memory** (*bool*) – Saves memory at the cost of longer computation time.
New in version 0.1.5.
- **copy** (*bool*) – Copy input sinogram *uSin* for data processing. If *copy* is set to *False*, then *uSin* will be overridden.
New in version 0.1.5.
- **count** (*multiprocessing.Value or None*) – Can be used to monitor the progress of the algorithm. Initially, the value of *max_count.value* is incremented by the total number of steps. At each step, the value of *count.value* is incremented.
- **max_count** (*multiprocessing.Value or None*) – Can be used to monitor the progress of the algorithm. Initially, the value of *max_count.value* is incremented by the total number of steps. At each step, the value of *count.value* is incremented.
- **verbose** (*int*) – Increment to increase verbosity.

Returns **f** – Reconstructed object function $f(\mathbf{r})$ as defined by the Helmholtz equation. $f(x, z) =$

$$k_m^2 \left(\left(\frac{n(x, z)}{n_m} \right)^2 - 1 \right)$$

Return type ndarray of shape (Nx, Ny, Nx), complex if *onlyreal==False*

See also:

odt_to_ri conversion of the object function $f(\mathbf{r})$ to refractive index $n(\mathbf{r})$

Notes

This implementation can deal with projection angles that are not distributed along a circle about the rotational axis. If there are slight deviations from this circle, simply pass the 3D rotational positions instead of the 1D angles to the *angles* argument. In principle, this should improve the reconstruction. The general problem here is that the backpropagation algorithm requires a ramp filter in Fourier space that is oriented perpendicular to the rotational axis. If the sample does not rotate about a single axis, then a 1D parametric representation of this rotation must be found to correctly determine the filter in Fourier space. Such a parametric representation could e.g. be a spiral between the poles of the unit sphere (but this kind of rotation is probably difficult to implement experimentally).

If you have input images with rectangular shape, e.g. $N_x \neq N_y$ and the rotational axis deviates by approximately $\pi/2$ from the axis (0,1,0), then data might get cropped in the reconstruction volume. You can avoid that by rotating your input data and the rotational axis by $\pi/2$. For instance, change `tilted_axis`` from [1,0,0] to [0,1,0] and `np.rot90` the sinogram images.

Do not use the parameter *ID* in combination with the Rytov approximation - the propagation is not correctly described. Instead, numerically refocus the sinogram prior to converting it to Rytov data (using e.g. `odtbrain.sinogram_as_rytov()` with a numerical focusing algorithm (available in the Python package `nrefocus`).

2.4 3D Apple core correction

The missing apple core (in Fourier space) leads to ringing and blurring artifacts in optical diffraction tomography [VDYH09]. This module contains basic functions that can be used to attenuate these artifacts.

New in version 0.3.0.

Changed in version 0.4.0.

`odtbrain.apple.apple_core_3d(shape, res, nm)`

Return a binary array with the apple core in 3D

Parameters

- **shape** (*list-like*, *length* 3) – Shape of the reconstruction volume for which to compute the apple core; The second (y-) axis is assumed to be the axis of symmetry (according to ODTbrain standard notation)
- **res** (*float*) – Size of the vacuum wave length λ in pixels
- **nm** (*float*) – Refractive index of the medium n_m

Returns **core** – The mask is *True* for positions within the apple core

Return type 3D ndarray

`odtbrain.apple.constraint_nn(data, mask=None, bg_shell=None)`

Non-negativity constraint

`odtbrain.apple.constraint_sh(data, mask=None, bg_shell=None)`

Symmetric histogram background data constraint

`odtbrain.apple.correct(f, res, nm, method='nn', mask=None, bg_shell_width=None, enforce_envelope=0.95, max_iter=100, min_diff=0.01, count=None, max_count=None)`

Fill the missing apple core of the object function

Parameters

- **f** (3D ndarray) – Complex objec function $f(\mathbf{r})$
- **res** (*float*) – Size of the vacuum wave length λ in pixels

- **nm** (*float*) – Refractive index of the medium n_m that surrounds the object in $n(\mathbf{r})$
- **method** (*str*) – One of:
 - “nn”: non-negativity constraint ($f > 0$). This method resembles classic missing apple core correction.
 - “sh”: symmetric histogram constraint (background data in f). This method works well for sparse-gradient data (e.g. works better than “nn” for simulated data), but might result in stripe-like artifacts when applied to experimental data.

The imaginary part of the refractive index is suppressed in both cases. Note that these constraints are soft, i.e. after the final inverse Fourier transform, the conditions might not be met.

- **mask** (*3D boolean ndarray, or None*) – Optional, defines background region(s) used for enforcing *method*. If a boolean ndarray, the values set to *True* define the used background regions.
- **bg_shell_width** (*float*) – Optional, defines the width of an ellipsoid shell (outer radii matching image shape) that is used additionally for enforcing *method*.
- **enforce_envelope** (*float in interval [0, 1] or False*) – Set the suppression factor for frequencies that are above the envelope function; disabled if set to *False* or 0
- **max_iter** (*int*) – Maximum number of iterations to perform
- **min_diff** (*float*) – Stopping criterion computed as the relative difference (relative to the first iteration *norm*) of the changes applied during the current iteration *cur_diff*: `np.abs(cur_diff/norm) < min_diff`
- **count** (*multiprocessing.Value*) – May be used for tracking progress. At each iteration *count.value* is incremented by one.
- **max_count** (*multiprocessing.Value*) – May be used for tracking progress; is incremented initially.

Notes

Internally, the Fourier transform is performed with single-precision floating point values (complex64).

`odtbrain.apple.count_to_half(array)`

Determination of half-initial value index

Return first index at which array values decrease below 1/2 of the initial initial value `array[0]`.

`odtbrain.apple.ellipsoid_shell(shape, width=20)`

Return background ellipsoid shell

`odtbrain.apple.envelope_gauss(ftdata, core)`

Compute a gaussian-filtered envelope, without apple core

Parameters

- **ftdata** (*3D ndarray*) – Fourier transform of the object function data (zero frequency not shifted to center of array)
- **core** (*3D ndarray (same shape as ftdata)*) – Apple core (as defined by `apple_core_3d()`)

Returns *envelope* – Envelope function in Fourier space

Return type 3D ndarray

`odtbrain.apple.spillover_region(shape, shell=0)`
Return boolean array for region outside ellipsoid

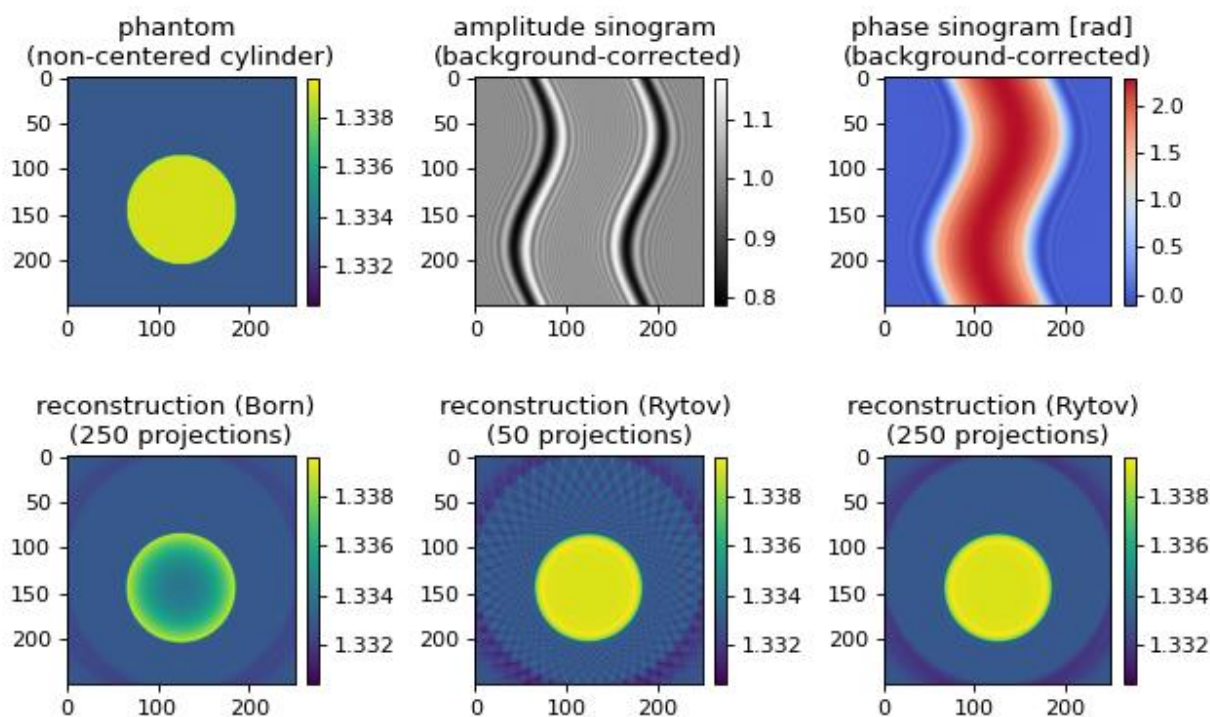
EXAMPLES

3.1 2D examples

These examples require raw data which are automatically downloaded from the source repository by the script `example_helper.py`. Please make sure that this script is present in the example script folder.

3.1.1 Mie off-center cylinder

The *in silico* data set was created with the software `miefield`. The data are 1D projections of an off-center cylinder of constant refractive index. The Born approximation is error-prone due to a relatively large radius of the cylinder (30 wavelengths) and a refractive index difference of 0.006 between cylinder and surrounding medium. The reconstruction of the refractive index with the Rytov approximation is in good agreement with the input data. When only 50 projections are used for the reconstruction, artifacts appear. These vanish when more projections are used for the reconstruction.



`backprop_from_mie_2d_cylinder_offcenter.py`

```

1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 import odtbrain as odt
5
6 from example_helper import load_data
7
8
9 # simulation data
10 sino, angles, cfg = load_data("mie_2d_noncentered_cylinder_A250_R2.zip",
11                               f_sino_imag="sino_imag.txt",
12                               f_sino_real="sino_real.txt",
13                               f_angles="mie_angles.txt",
14                               f_info="mie_info.txt")
15
16 A, size = sino.shape
17
18 # background sinogram computed with Mie theory
19 # miefield.GetSinogramCylinderRotation(radius, nmed, nmed, lD, lC, size, A, res)
20 u0 = load_data("mie_2d_noncentered_cylinder_A250_R2.zip",
21               f_sino_imag="u0_imag.txt",
22               f_sino_real="u0_real.txt")
23
24 # create 2d array
25 u0 = np.tile(u0, size).reshape(A, size).transpose()
26
27 # background field necessary to compute initial born field
28 # u0_single = mie.GetFieldCylinder(radius, nmed, nmed, lD, size, res)
29 u0_single = load_data("mie_2d_noncentered_cylinder_A250_R2.zip",
30                       f_sino_imag="u0_single_imag.txt",
31                       f_sino_real="u0_single_real.txt")
32
33 print("Example: Backpropagation from 2D Mie simulations")
34 print("Refractive index of medium:", cfg["nmed"])
35 print("Measurement position from object center:", cfg["lD"])
36 print("Wavelength sampling:", cfg["res"])
37 print("Performing backpropagation.")
38
39 # Set measurement parameters
40 # Compute scattered field from cylinder
41 radius = cfg["radius"] # wavelengths
42 nmed = cfg["nmed"]
43 ncyl = cfg["ncyl"]
44
45 lD = cfg["lD"] # measurement distance in wavelengths
46 lC = cfg["lC"] # displacement from center of image
47 size = cfg["size"]
48 res = cfg["res"] # px/wavelengths
49 A = cfg["A"] # number of projections
50
51 x = np.arange(size) - size / 2
52 X, Y = np.meshgrid(x, x)
53 rad_px = radius * res
54 phantom = np.array(((Y - lC * res)**2 + X**2) < rad_px**2,
55                    dtype=float) * (ncyl - nmed) + nmed

```

(continues on next page)

(continued from previous page)

```

54
55 # Born
56 u_sinB = (sino / u0 * u0_single - u0_single) # fake born
57 fB = odt.backpropagate_2d(u_sinB, angles, res, nmed, 1D * res)
58 nB = odt.odt_to_ri(fB, res, nmed)
59
60 # Rytov
61 u_sinR = odt.sinogram_as_rytov(sino / u0)
62 fR = odt.backpropagate_2d(u_sinR, angles, res, nmed, 1D * res)
63 nR = odt.odt_to_ri(fR, res, nmed)
64
65 # Rytov 50
66 u_sinR50 = odt.sinogram_as_rytov((sino / u0)[:5, :])
67 fR50 = odt.backpropagate_2d(u_sinR50, angles[:5], res, nmed, 1D * res)
68 nR50 = odt.odt_to_ri(fR50, res, nmed)
69
70 # Plot sinogram phase and amplitude
71 ph = odt.sinogram_as_radon(sino / u0)
72
73 am = np.abs(sino / u0)
74
75 # prepare plot
76 vmin = np.min(np.array([phantom, nB.real, nR50.real, nR.real]))
77 vmax = np.max(np.array([phantom, nB.real, nR50.real, nR.real]))
78
79 fig, axes = plt.subplots(2, 3, figsize=(8, 5))
80 axes = np.array(axes).flatten()
81
82 phantommap = axes[0].imshow(phantom, vmin=vmin, vmax=vmax)
83 axes[0].set_title("phantom \n(non-centered cylinder)")
84
85 amplmap = axes[1].imshow(am, cmap="gray")
86 axes[1].set_title("amplitude sinogram \n(background-corrected)")
87
88 phasemap = axes[2].imshow(ph, cmap="coolwarm")
89 axes[2].set_title("phase sinogram [rad] \n(background-corrected)")
90
91 axes[3].imshow(nB.real, vmin=vmin, vmax=vmax)
92 axes[3].set_title("reconstruction (Born) \n(250 projections)")
93
94 axes[4].imshow(nR50.real, vmin=vmin, vmax=vmax)
95 axes[4].set_title("reconstruction (Rytov) \n(50 projections)")
96
97 axes[5].imshow(nR.real, vmin=vmin, vmax=vmax)
98 axes[5].set_title("reconstruction (Rytov) \n(250 projections)")
99
100 # color bars
101 cbkwargs = {"fraction": 0.045}
102 plt.colorbar(phantommap, ax=axes[0], **cbkwargs)
103 plt.colorbar(amplmap, ax=axes[1], **cbkwargs)
104 plt.colorbar(phasemap, ax=axes[2], **cbkwargs)
105 plt.colorbar(phantommap, ax=axes[3], **cbkwargs)

```

(continues on next page)

(continued from previous page)

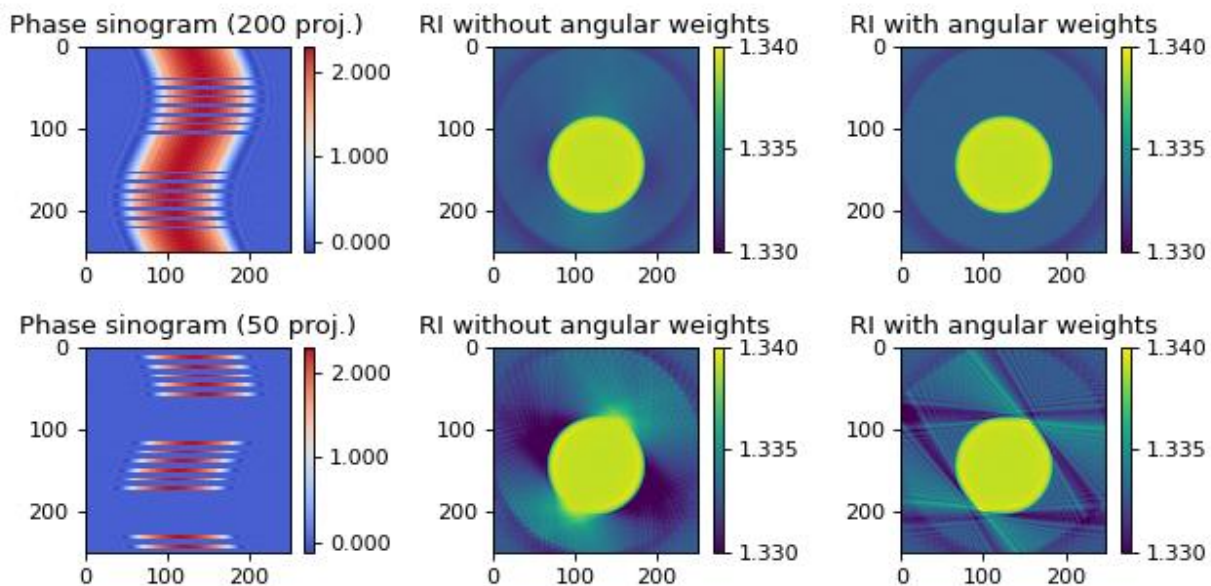
```

106 plt.colorbar(phantommap, ax=axes[4], **cbkwargs)
107 plt.colorbar(phantommap, ax=axes[5], **cbkwargs)
108
109 plt.tight_layout()
110 plt.show()

```

3.1.2 Mie cylinder with unevenly spaced angles

Angular weighting can significantly improve reconstruction quality when the angular projections are sampled at non-equidistant intervals [TPM81]. The *in silico* data set was created with the software [miefield](#). The data are 1D projections of a non-centered cylinder of constant refractive index 1.339 embedded in water with refractive index 1.333. The first column shows the used sinograms (missing angles are displayed as zeros) that were created from the original sinogram with 250 projections. The second column shows the reconstruction without angular weights and the third column shows the reconstruction with angular weights. The keyword argument `weight_angles` was introduced in version 0.1.1.



backprop_from_mie_2d_weights_angles.py

```

1  import matplotlib.pyplot as plt
2  import numpy as np
3  from skimage.restoration import unwrap_phase
4
5  import odtbrain as odt
6
7  from example_helper import load_data
8
9
10 sino, angles, cfg = load_data("mie_2d_noncentered_cylinder_A250_R2.zip",
11                               f_angles="mie_angles.txt",
12                               f_sino_real="sino_real.txt",
13                               f_sino_imag="sino_imag.txt",

```

(continues on next page)

(continued from previous page)

```

14         f_info="mie_info.txt")
15 A, size = sino.shape
16
17 # background sinogram computed with Mie theory
18 # miefield.GetSinogramCylinderRotation(radius, nmed, nmed, lD, lC, size, A, res)
19 u0 = load_data("mie_2d_noncentered_cylinder_A250_R2.zip",
20               f_sino_imag="u0_imag.txt",
21               f_sino_real="u0_real.txt")
22 # create 2d array
23 u0 = np.tile(u0, size).reshape(A, size).transpose()
24
25 # background field necessary to compute initial born field
26 # u0_single = mie.GetFieldCylinder(radius, nmed, nmed, lD, size, res)
27 u0_single = load_data("mie_2d_noncentered_cylinder_A250_R2.zip",
28                      f_sino_imag="u0_single_imag.txt",
29                      f_sino_real="u0_single_real.txt")
30
31
32 print("Example: Backpropagation from 2D FDTD simulations")
33 print("Refractive index of medium:", cfg["nmed"])
34 print("Measurement position from object center:", cfg["lD"])
35 print("Wavelength sampling:", cfg["res"])
36 print("Performing backpropagation.")
37
38 # Set measurement parameters
39 # Compute scattered field from cylinder
40 radius = cfg["radius"] # wavelengths
41 nmed = cfg["nmed"]
42 ncyl = cfg["ncyl"]
43
44 lD = cfg["lD"] # measurement distance in wavelengths
45 lC = cfg["lC"] # displacement from center of image
46 size = cfg["size"]
47 res = cfg["res"] # px/wavelengths
48 A = cfg["A"] # number of projections
49
50 x = np.arange(size) - size / 2.0
51 X, Y = np.meshgrid(x, x)
52 rad_px = radius * res
53 phantom = np.array(((Y - lC * res)**2 + X**2) < rad_px **
54                    2, dtype=float) * (ncyl - nmed) + nmed
55
56 u_sinR = odt.sinogram_as_rytov(sino / u0)
57
58 # Rytov 200 projections
59 # remove 50 projections from total of 250 projections
60 remove200 = np.argsort(angles % .0002)[:50]
61 angles200 = np.delete(angles, remove200, axis=0)
62 u_sinR200 = np.delete(u_sinR, remove200, axis=0)
63 ph200 = unwrap_phase(np.angle(sino / u0))
64 ph200[remove200] = 0
65

```

(continues on next page)

(continued from previous page)

```

66 fR200 = odt.backpropagate_2d(u_sinR200, angles200, res, nmed, 1D*res)
67 nR200 = odt.odt_to_ri(fR200, res, nmed)
68 fR200nw = odt.backpropagate_2d(u_sinR200, angles200, res, nmed, 1D*res,
69                               weight_angles=False)
70 nR200nw = odt.odt_to_ri(fR200nw, res, nmed)
71
72 # Rytov 50 projections
73 remove50 = np.argsort(angles % .0002)[:200]
74 angles50 = np.delete(angles, remove50, axis=0)
75 u_sinR50 = np.delete(u_sinR, remove50, axis=0)
76 ph50 = unwrap_phase(np.angle(sino / u0))
77 ph50[remove50] = 0
78
79 fR50 = odt.backpropagate_2d(u_sinR50, angles50, res, nmed, 1D*res)
80 nR50 = odt.odt_to_ri(fR50, res, nmed)
81 fR50nw = odt.backpropagate_2d(u_sinR50, angles50, res, nmed, 1D*res,
82                               weight_angles=False)
83 nR50nw = odt.odt_to_ri(fR50nw, res, nmed)
84
85 # prepare plot
86 kw_ri = {"vmin": 1.330,
87         "vmax": 1.340}
88
89 kw_ph = {"vmin": np.min(np.array([ph200, ph50])),
90         "vmax": np.max(np.array([ph200, ph50])),
91         "cmap": "coolwarm",
92         "interpolation": "none"}
93
94 fig, axes = plt.subplots(2, 3, figsize=(8, 4))
95 axes = np.array(axes).flatten()
96
97 phmap = axes[0].imshow(ph200, **kw_ph)
98 axes[0].set_title("Phase sinogram (200 proj.)")
99
100 rimap = axes[1].imshow(nR200nw.real, **kw_ri)
101 axes[1].set_title("RI without angular weights")
102
103 axes[2].imshow(nR200.real, **kw_ri)
104 axes[2].set_title("RI with angular weights")
105
106 axes[3].imshow(ph50, **kw_ph)
107 axes[3].set_title("Phase sinogram (50 proj.)")
108
109 axes[4].imshow(nR50nw.real, **kw_ri)
110 axes[4].set_title("RI without angular weights")
111
112 axes[5].imshow(nR50.real, **kw_ri)
113 axes[5].set_title("RI with angular weights")
114
115 # color bars
116 cbkwargs = {"fraction": 0.045,
117            "format": "%.3f"}

```

(continues on next page)

(continued from previous page)

```

118 plt.colorbar(phmap, ax=axes[0], **cbkwargs)
119 plt.colorbar(phmap, ax=axes[3], **cbkwargs)
120 plt.colorbar(rimap, ax=axes[1], **cbkwargs)
121 plt.colorbar(rimap, ax=axes[2], **cbkwargs)
122 plt.colorbar(rimap, ax=axes[5], **cbkwargs)
123 plt.colorbar(rimap, ax=axes[4], **cbkwargs)
124
125 plt.tight_layout()
126 plt.show()

```

3.1.3 Mie cylinder with incomplete angular coverage

This example illustrates how the backpropagation algorithm of ODTbrain handles incomplete angular coverage. All examples use 100 projections at 100%, 60%, and 40% total angular coverage. The keyword argument *weight_angles* that invokes angular weighting is set to *True* by default. The *in silico* data set was created with the software *miefield*. The data are 1D projections of a non-centered cylinder of constant refractive index 1.339 embedded in water with refractive index 1.333. The first column shows the used sinograms (missing angles are displayed as zeros) that were created from the original sinogram with 250 projections. The second column shows the reconstruction without angular weights and the third column shows the reconstruction with angular weights. The keyword argument *weight_angles* was introduced in version 0.1.1.

A 180 degree coverage results in a good reconstruction of the object. Angular weighting as implemented in the backpropagation algorithm of ODTbrain automatically addresses uneven and incomplete angular coverage.

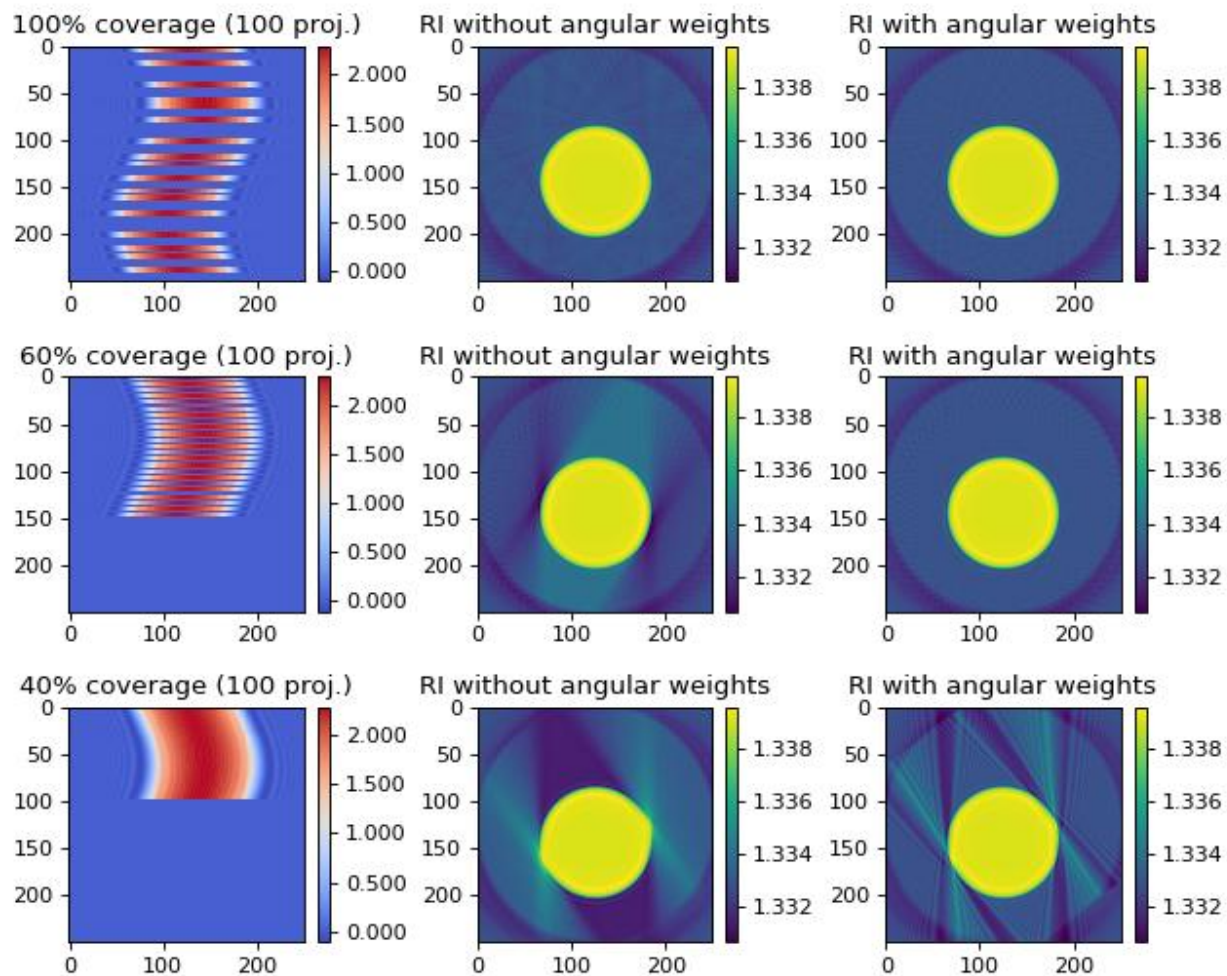
backprop_from_mie_2d_incomplete_coverage.py

```

1  import matplotlib.pyplot as plt
2  import numpy as np
3
4  import odtbrain as odt
5
6  from example_helper import load_data
7
8  sino, angles, cfg = load_data("mie_2d_noncentered_cylinder_A250_R2.zip",
9                               f_angles="mie_angles.txt",
10                              f_sino_real="sino_real.txt",
11                              f_sino_imag="sino_imag.txt",
12                              f_info="mie_info.txt")
13
14  A, size = sino.shape
15
16  # background sinogram computed with Mie theory
17  # miefield.GetSinogramCylinderRotation(radius, nmed, nmed, lD, lC, size, A, res)
18  u0 = load_data("mie_2d_noncentered_cylinder_A250_R2.zip",
19                f_sino_imag="u0_imag.txt",
20                f_sino_real="u0_real.txt")
21
22  # create 2d array
23  u0 = np.tile(u0, size).reshape(A, size).transpose()
24
25  # background field necessary to compute initial born field
26  # u0_single = mie.GetFieldCylinder(radius, nmed, nmed, lD, size, res)
27  u0_single = load_data("mie_2d_noncentered_cylinder_A250_R2.zip",
28                        f_sino_imag="u0_single_imag.txt",

```

(continues on next page)



(continued from previous page)

```

27         f_sino_real="u0_single_real.txt")
28
29 print("Example: Backpropagation from 2D FDTD simulations")
30 print("Refractive index of medium:", cfg["nmed"])
31 print("Measurement position from object center:", cfg["lD"])
32 print("Wavelength sampling:", cfg["res"])
33 print("Performing backpropagation.")
34
35 # Set measurement parameters
36 # Compute scattered field from cylinder
37 radius = cfg["radius"] # wavelengths
38 nmed = cfg["nmed"]
39 ncyl = cfg["ncyl"]
40
41 lD = cfg["lD"] # measurement distance in wavelengths
42 lC = cfg["lC"] # displacement from center of image
43 size = cfg["size"]
44 res = cfg["res"] # px/wavelengths
45 A = cfg["A"] # number of projections
46
47 x = np.arange(size) - size / 2.0
48 X, Y = np.meshgrid(x, x)
49 rad_px = radius * res
50 phantom = np.array(((Y - lC * res)**2 + X**2) < rad_px **
51                    2, dtype=float) * (ncyl - nmed) + nmed
52
53 u_sinR = odt.sinogram_as_rytov(sino / u0)
54
55 # Rytov 100 projections evenly distributed
56 removeeven = np.argsort(angles % .002)[:150]
57 angleseven = np.delete(angles, removeeven, axis=0)
58 u_sinReven = np.delete(u_sinR, removeeven, axis=0)
59 pheven = odt.sinogram_as_radon(sino / u0)
60 pheven[removeeven] = 0
61
62 fReven = odt.backpropagate_2d(u_sinReven, angleseven, res, nmed, lD * res)
63 nReven = odt.odt_to_ri(fReven, res, nmed)
64 fRevennw = odt.backpropagate_2d(
65     u_sinReven, angleseven, res, nmed, lD * res, weight_angles=False)
66 nRevennw = odt.odt_to_ri(fRevennw, res, nmed)
67
68 # Rytov 100 projections more than 180
69 removemiss = 249 - \
70     np.concatenate((np.arange(100), 100 + np.arange(150)[::3]))
71 anglesmiss = np.delete(angles, removemiss, axis=0)
72 u_sinRmiss = np.delete(u_sinR, removemiss, axis=0)
73 phmiss = odt.sinogram_as_radon(sino / u0)
74 phmiss[removemiss] = 0
75
76 fRmiss = odt.backpropagate_2d(u_sinRmiss, anglesmiss, res, nmed, lD * res)
77 nRmiss = odt.odt_to_ri(fRmiss, res, nmed)
78 fRmissnw = odt.backpropagate_2d(

```

(continues on next page)

(continued from previous page)

```

79     u_sinRmiss, anglesmiss, res, nmed, 1D * res, weight_angles=False)
80 nRmissnw = odt.odt_to_ri(fRmissnw, res, nmed)
81
82 # Rytov 100 projections less than 180
83 removebad = 249 - np.arange(150)
84 anglesbad = np.delete(angles, removebad, axis=0)
85 u_sinRbad = np.delete(u_sinR, removebad, axis=0)
86 phbad = odt.sinogram_as_radon(sino / u0)
87 phbad[removebad] = 0
88
89 fRbad = odt.backpropagate_2d(u_sinRbad, anglesbad, res, nmed, 1D * res)
90 nRbad = odt.odt_to_ri(fRbad, res, nmed)
91 fRbadnw = odt.backpropagate_2d(
92     u_sinRbad, anglesbad, res, nmed, 1D * res, weight_angles=False)
93 nRbadnw = odt.odt_to_ri(fRbadnw, res, nmed)
94
95 # prepare plot
96 kw_ri = {"vmin": np.min(np.array([phantom, nRmiss.real, nReven.real])),
97         "vmax": np.max(np.array([phantom, nRmiss.real, nReven.real]))}
98
99 kw_ph = {"vmin": np.min(np.array([pheven, phmiss])),
100         "vmax": np.max(np.array([pheven, phmiss])),
101         "cmap": "coolwarm",
102         "interpolation": "none"}
103
104 fig, axes = plt.subplots(3, 3, figsize=(8, 6.5))
105
106 axes[0, 0].set_title("100% coverage ({} proj.)".format(angleseven.shape[0]))
107 phmap = axes[0, 0].imshow(pheven, **kw_ph)
108
109 axes[0, 1].set_title("RI without angular weights")
110 rimap = axes[0, 1].imshow(nRevennw.real, **kw_ri)
111
112 axes[0, 2].set_title("RI with angular weights")
113 rimap = axes[0, 2].imshow(nReven.real, **kw_ri)
114
115 axes[1, 0].set_title("60% coverage ({} proj.)".format(anglesmiss.shape[0]))
116 axes[1, 0].imshow(phmiss, **kw_ph)
117
118 axes[1, 1].set_title("RI without angular weights")
119 axes[1, 1].imshow(nRmissnw.real, **kw_ri)
120
121 axes[1, 2].set_title("RI with angular weights")
122 axes[1, 2].imshow(nRmiss.real, **kw_ri)
123
124 axes[2, 0].set_title("40% coverage ({} proj.)".format(anglesbad.shape[0]))
125 axes[2, 0].imshow(phbad, **kw_ph)
126
127 axes[2, 1].set_title("RI without angular weights")
128 axes[2, 1].imshow(nRbadnw.real, **kw_ri)
129
130 axes[2, 2].set_title("RI with angular weights")

```

(continues on next page)

(continued from previous page)

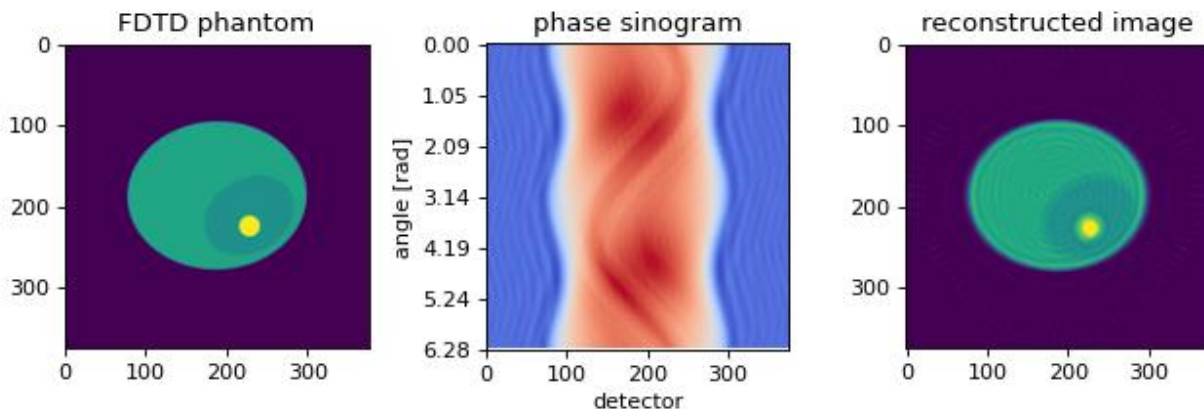
```

131 axes[2, 2].imshow(nRbad.real, **kw_ri)
132
133 # color bars
134 cbkwargs = {"fraction": 0.045,
135             "format": "%.3f"}
136 plt.colorbar(phmap, ax=axes[0, 0], **cbkwargs)
137 plt.colorbar(phmap, ax=axes[1, 0], **cbkwargs)
138 plt.colorbar(phmap, ax=axes[2, 0], **cbkwargs)
139 plt.colorbar(rimap, ax=axes[0, 1], **cbkwargs)
140 plt.colorbar(rimap, ax=axes[1, 1], **cbkwargs)
141 plt.colorbar(rimap, ax=axes[2, 1], **cbkwargs)
142 plt.colorbar(rimap, ax=axes[0, 2], **cbkwargs)
143 plt.colorbar(rimap, ax=axes[1, 2], **cbkwargs)
144 plt.colorbar(rimap, ax=axes[2, 2], **cbkwargs)
145
146 plt.tight_layout()
147 plt.show()

```

3.1.4 FDTD cell phantom

The *in silico* data set was created with the FDTD software [meep](#). The data are 1D projections of a 2D refractive index phantom. The reconstruction of the refractive index with the Rytov approximation is in good agreement with the phantom that was used in the simulation.



backprop_from_fDTD_2d.py

```

1 import matplotlib.pyplot as plt
2 import numpy as np
3 import odtbrain as odt
4
5 from example_helper import load_data
6
7
8 sino, angles, phantom, cfg = load_data("fdtd_2d_sino_A100_R13.zip",
9                                       f_angles="fdtd_angles.txt",
10                                       f_sino_imag="fdtd_imag.txt",

```

(continues on next page)

(continued from previous page)

```

11         f_sino_real="fdtd_real.txt",
12         f_info="fdtd_info.txt",
13         f_phantom="fdtd_phantom.txt",
14     )
15
16     print("Example: Backpropagation from 2D FDTD simulations")
17     print("Refractive index of medium:", cfg["nm"])
18     print("Measurement position from object center:", cfg["lD"])
19     print("Wavelength sampling:", cfg["res"])
20     print("Performing backpropagation.")
21
22     # Apply the Rytov approximation
23     sino_rytov = odt.sinogram_as_rytov(sino)
24
25     # perform backpropagation to obtain object function f
26     f = odt.backpropagate_2d(uSin=sino_rytov,
27                             angles=angles,
28                             res=cfg["res"],
29                             nm=cfg["nm"],
30                             lD=cfg["lD"] * cfg["res"]
31                         )
32
33     # compute refractive index n from object function
34     n = odt.odt_to_ri(f, res=cfg["res"], nm=cfg["nm"])
35
36     # compare phantom and reconstruction in plot
37     fig, axes = plt.subplots(1, 3, figsize=(8, 2.8))
38
39     axes[0].set_title("FDTD phantom")
40     axes[0].imshow(phantom, vmin=phantom.min(), vmax=phantom.max())
41     sino_phase = np.unwrap(np.angle(sino), axis=1)
42
43     axes[1].set_title("phase sinogram")
44     axes[1].imshow(sino_phase, vmin=sino_phase.min(), vmax=sino_phase.max(),
45                   aspect=sino.shape[1] / sino.shape[0],
46                   cmap="coolwarm")
47     axes[1].set_xlabel("detector")
48     axes[1].set_ylabel("angle [rad]")
49
50     axes[2].set_title("reconstructed image")
51     axes[2].imshow(n.real, vmin=phantom.min(), vmax=phantom.max())
52
53     # set y ticks for sinogram
54     labels = np.linspace(0, 2 * np.pi, len(axes[1].get_yticks()))
55     labels = ["{:.2f}".format(i) for i in labels]
56     axes[1].set_yticks(np.linspace(0, len(angles), len(labels)))
57     axes[1].set_yticklabels(labels)
58
59     plt.tight_layout()
60     plt.show()

```

3.2 3D examples

These examples require raw data which are automatically downloaded from the source repository by the script `example_helper.py`. Please make sure that this script is present in the example script folder.

Note: The `if __name__ == "__main__":` guard is necessary on Windows and macOS which *spawn* new processes instead of *forking* the current process. The 3D backpropagation algorithm makes use of `multiprocessing.Pool`.

3.2.1 Missing apple core correction

The missing apple core [VDYH09] is a phenomenon in diffraction tomography that is a result of the fact the the Fourier space is not filled completely when the sample is rotated only about a single axis. The resulting artifacts include ringing and blurring in the reconstruction parallel to the original rotation axis. By enforcing constraints (refractive index real-valued and larger than the surrounding medium), these artifacts can be attenuated.

This example generates an artificial sinogram using the Python library `cellsino` (The example parameters are reused from [this example](#)). The sinogram is then reconstructed with the backpropagation algorithm and the missing apple core correction is applied.

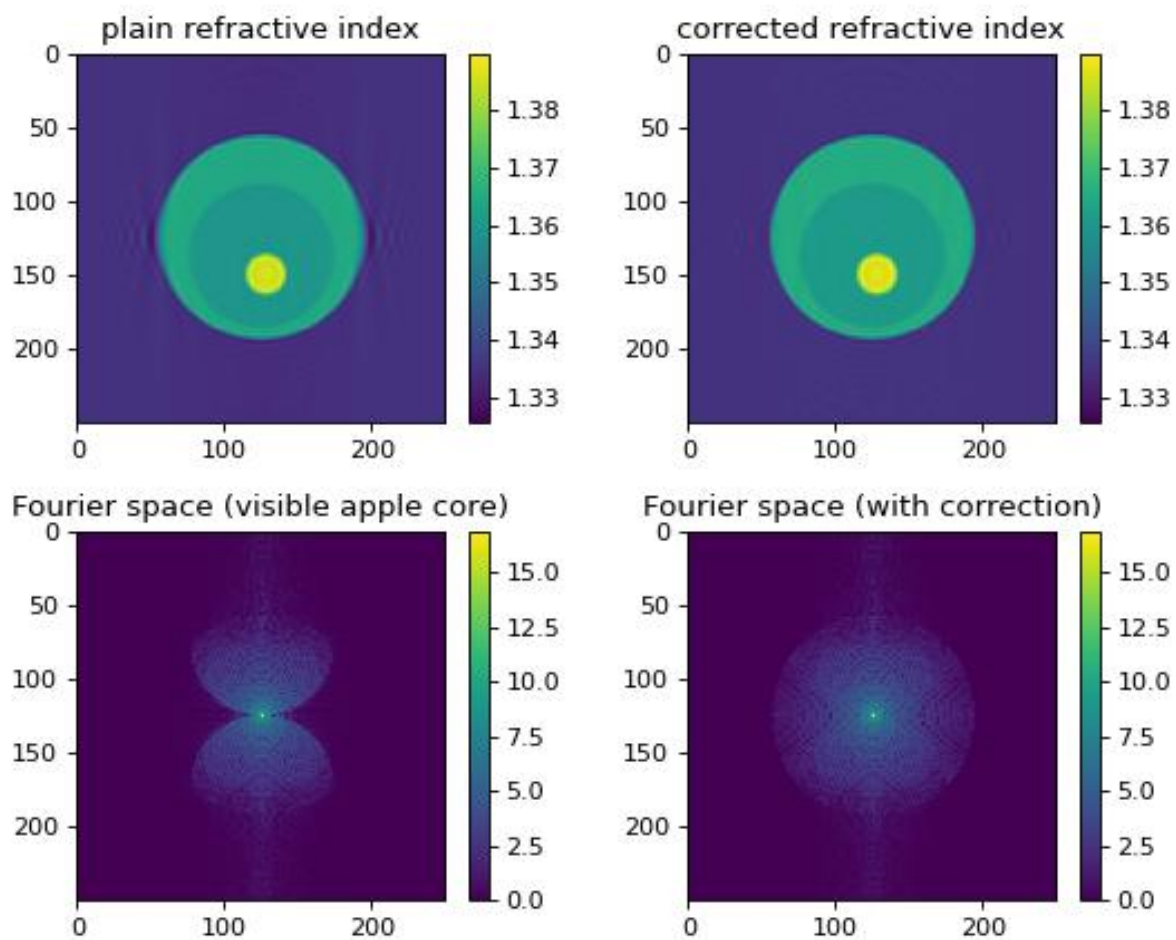
Note: The missing apple core correction `odtbrain.apple.correct()` was implemented in version 0.3.0 and is thus not used in the older examples.

`backprop_from_rytov_3d_phantom_apple.py`

```

1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 import cellsino
5 import odtbrain as odt
6
7
8 if __name__ == "__main__":
9     # number of sinogram angles
10    num_ang = 160
11    # sinogram acquisition angles
12    angles = np.linspace(0, 2*np.pi, num_ang, endpoint=False)
13    # detector grid size
14    grid_size = (250, 250)
15    # vacuum wavelength [m]
16    wavelength = 550e-9
17    # pixel size [m]
18    pixel_size = 0.08e-6
19    # refractive index of the surrounding medium
20    medium_index = 1.335
21
22    # initialize cell phantom
23    phantom = cellsino.phantoms.SimpleCell()
24
25    # initialize sinogram with geometric parameters
26    sino = cellsino.Sinogram(phantom=phantom,
```

(continues on next page)



(continued from previous page)

```

27         wavelength=wavelength,
28         pixel_size=pixel_size,
29         grid_size=grid_size)
30
31     # compute sinogram (field with Rytov approximation and fluorescence)
32     sino = sino.compute(angles=angles, propagator="rytov", mode="field")
33
34     # reconstruction of refractive index
35     sino_rytov = odt.sinogram_as_rytov(sino)
36     f = odt.backpropagate_3d(uSin=sino_rytov,
37                             angles=angles,
38                             res=wavelength/pixel_size,
39                             nm=medium_index)
40
41     ri = odt.odt_to_ri(f=f,
42                      res=wavelength/pixel_size,
43                      nm=medium_index)
44
45     # apple core correction
46     fc = odt.apple.correct(f=f,
47                           res=wavelength/pixel_size,
48                           nm=medium_index,
49                           method="sh")
50
51     ric = odt.odt_to_ri(f=fc,
52                       res=wavelength/pixel_size,
53                       nm=medium_index)
54
55     # plotting
56     idx = ri.shape[2] // 2
57
58     # log-scaled power spectra
59     ft = np.log(1 + np.abs(np.fft.fftshift(np.fft.fftn(ri))))
60     ftc = np.log(1 + np.abs(np.fft.fftshift(np.fft.fftn(ric))))
61
62     plt.figure(figsize=(7, 5.5))
63
64     plotkwri = {"vmax": ri.real.max(),
65               "vmin": ri.real.min(),
66               "interpolation": "none",
67               }
68
69     plotkwft = {"vmax": ft.max(),
70               "vmin": 0,
71               "interpolation": "none",
72               }
73
74     ax1 = plt.subplot(221, title="plain refractive index")
75     mapper = ax1.imshow(ri[:, :, idx].real, **plotkwri)
76     plt.colorbar(mappable=mapper, ax=ax1)
77
78     ax2 = plt.subplot(222, title="corrected refractive index")

```

(continues on next page)

(continued from previous page)

```

79 mapper = ax2.imshow(ric[:, :, idx].real, **plotkwri)
80 plt.colorbar(mappable=mapper, ax=ax2)
81
82 ax3 = plt.subplot(223, title="Fourier space (visible apple core)")
83 mapper = ax3.imshow(ft[:, :, idx], **plotkwft)
84 plt.colorbar(mappable=mapper, ax=ax3)
85
86 ax4 = plt.subplot(224, title="Fourier space (with correction)")
87 mapper = ax4.imshow(ftc[:, :, idx], **plotkwft)
88 plt.colorbar(mappable=mapper, ax=ax4)
89
90 plt.tight_layout()
91 plt.show()

```

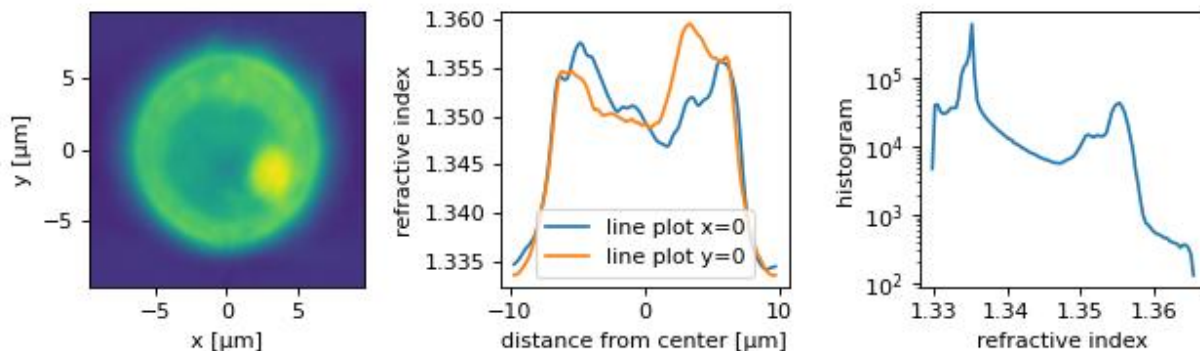
3.2.2 HL60 cell

The quantitative phase data of an HL60 S/4 cell were recorded using QLSI (quadri-wave lateral shearing interferometry). The original dataset was used in a previous publication [SCG+17] to illustrate the capabilities of combined fluorescence and refractive index tomography.

The example data set is already aligned and background-corrected as described in the original publication and the fluorescence data are not included. The lzma-archive contains the sinogram data stored in the `qpimage` file format and the rotational positions of each sinogram image as a text file.

The figure reproduces parts of figure 4 of the original manuscript. Note that minor deviations from the original figure can be attributed to the strong compression (scale offset filter) and due to the fact that the original sinogram images were cropped from 196x196 px to 140x140 px (which in particular affects the background-part of the refractive index histogram).

The raw data is available *on figshare* <<https://doi.org/10.6084/m9.figshare.8055407.v1>> (hl60_sinogram_qpi.h5).



backprop_from_qlsi_3d_hl60.py

```

1 import pathlib
2 import tarfile
3 import tempfile
4
5 import matplotlib.pyplot as plt
6 import numpy as np

```

(continues on next page)

(continued from previous page)

```

7 import odtbrain as odt
8 import qpimage
9
10 from example_helper import get_file, extract_lzma
11
12
13 if __name__ == "__main__":
14     # ascertain the data
15     path = get_file("qlsi_3d_hl60-cell_A140.tar.lzma")
16     tarf = extract_lzma(path)
17     tdir = tempfile.mkdtemp(prefix="odtbrain_example_")
18
19     with tarfile.open(tarf) as tf:
20         tf.extract("series.h5", path=tdir)
21         angles = np.loadtxt(tf.extractfile("angles.txt"))
22
23     # extract the complex field sinogram from the qpimage series data
24     h5file = pathlib.Path(tdir) / "series.h5"
25     with qpimage.QPSeries(h5file=h5file, h5mode="r") as qps:
26         qp0 = qps[0]
27         meta = qp0.meta
28         sino = np.zeros((len(qps), qp0.shape[0], qp0.shape[1]),
29                         dtype=np.complex)
30         for ii in range(len(qps)):
31             sino[ii] = qps[ii].field
32
33     # perform backpropagation
34     u_sinR = odt.sinogram_as_rytov(sino)
35     res = meta["wavelength"] / meta["pixel size"]
36     nm = meta["medium index"]
37
38     fR = odt.backpropagate_3d(uSin=u_sinR,
39                             angles=angles,
40                             res=res,
41                             nm=nm)
42
43     ri = odt.odt_to_ri(fR, res, nm)
44
45     # plot results
46     ext = meta["pixel size"] * 1e6 * 70
47     kw = {"vmin": ri.real.min(),
48          "vmax": ri.real.max(),
49          "extent": [-ext, ext, -ext, ext]}
50     fig, axes = plt.subplots(1, 3, figsize=(8, 2.5))
51     axes[0].imshow(ri[70, :, :].real, **kw)
52     axes[0].set_xlabel("x [ $\mu\text{m}$ ]")
53     axes[0].set_ylabel("y [ $\mu\text{m}$ ]")
54
55     x = np.linspace(-ext, ext, 140)
56     axes[1].plot(x, ri[70, :, 70], label="line plot x=0")
57     axes[1].plot(x, ri[70, 70, :], label="line plot y=0")
58     axes[1].set_xlabel("distance from center [ $\mu\text{m}$ ]")

```

(continues on next page)

(continued from previous page)

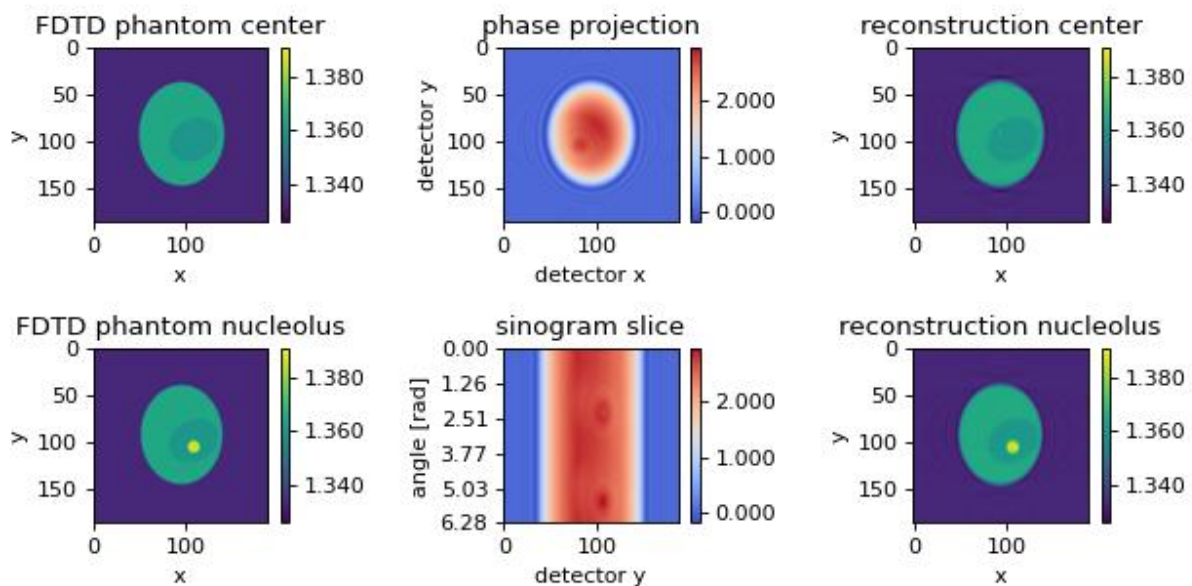
```

59 axes[1].set_ylabel("refractive index")
60 axes[1].legend()
61
62 hist, xh = np.histogram(ri.real, bins=100)
63 axes[2].plot(xh[1:], hist)
64 axes[2].set_yscale('log')
65 axes[2].set_xlabel("refractive index")
66 axes[2].set_ylabel("histogram")
67
68 plt.tight_layout()
69 plt.show()

```

3.2.3 FDTD cell phantom

The *in silico* data set was created with the FDTD software [meep](#). The data are 2D projections of a 3D refractive index phantom. The reconstruction of the refractive index with the Rytov approximation is in good agreement with the phantom that was used in the simulation. The data are downsampled by a factor of two. The rotational axis is the y-axis. A total of 180 projections are used for the reconstruction. A detailed description of this phantom is given in [MSG15a].



backprop_from_fDTD_3d.py

```

1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 import odtbrain as odt
5
6 from example_helper import load_data
7
8

```

(continues on next page)

(continued from previous page)

```

9  if __name__ == "__main__":
10     sino, angles, phantom, cfg = \
11         load_data("fdtd_3d_sino_A180_R6.500.tar.lzma")
12
13     A = angles.shape[0]
14
15     print("Example: Backpropagation from 3D FDTD simulations")
16     print("Refractive index of medium:", cfg["nm"])
17     print("Measurement position from object center:", cfg["lD"])
18     print("Wavelength sampling:", cfg["res"])
19     print("Number of projections:", A)
20     print("Performing backpropagation.")
21
22     # Apply the Rytov approximation
23     sinoRytov = odt.sinogram_as_rytov(sino)
24
25     # perform backpropagation to obtain object function f
26     f = odt.backpropagate_3d(uSin=sinoRytov,
27                             angles=angles,
28                             res=cfg["res"],
29                             nm=cfg["nm"],
30                             lD=cfg["lD"]
31                             )
32
33     # compute refractive index n from object function
34     n = odt.odt_to_ri(f, res=cfg["res"], nm=cfg["nm"])
35
36     sx, sy, sz = n.shape
37     px, py, pz = phantom.shape
38
39     sino_phase = np.angle(sino)
40
41     # compare phantom and reconstruction in plot
42     fig, axes = plt.subplots(2, 3, figsize=(8, 4))
43     kwri = {"vmin": n.real.min(), "vmax": n.real.max()}
44     kwph = {"vmin": sino_phase.min(), "vmax": sino_phase.max(),
45            "cmap": "coolwarm"}
46
47     # Phantom
48     axes[0, 0].set_title("FDTD phantom center")
49     rimap = axes[0, 0].imshow(phantom[px // 2], **kwri)
50     axes[0, 0].set_xlabel("x")
51     axes[0, 0].set_ylabel("y")
52
53     axes[1, 0].set_title("FDTD phantom nucleolus")
54     axes[1, 0].imshow(phantom[int(px / 2 + 2 * cfg["res"])], **kwri)
55     axes[1, 0].set_xlabel("x")
56     axes[1, 0].set_ylabel("y")
57
58     # Sinogram
59     axes[0, 1].set_title("phase projection")
60     phmap = axes[0, 1].imshow(sino_phase[A // 2, :, :], **kwph)

```

(continues on next page)

(continued from previous page)

```

61 axes[0, 1].set_xlabel("detector x")
62 axes[0, 1].set_ylabel("detector y")
63
64 axes[1, 1].set_title("sinogram slice")
65 axes[1, 1].imshow(sino_phase[:, :, sino.shape[2] // 2],
66                   aspect=sino.shape[1] / sino.shape[0], **kwph)
67 axes[1, 1].set_xlabel("detector y")
68 axes[1, 1].set_ylabel("angle [rad]")
69 # set y ticks for sinogram
70 labels = np.linspace(0, 2 * np.pi, len(axes[1, 1].get_yticks()))
71 labels = ["{:0.2f}".format(i) for i in labels]
72 axes[1, 1].set_yticks(np.linspace(0, len(labels), len(labels)))
73 axes[1, 1].set_yticklabels(labels)
74
75 axes[0, 2].set_title("reconstruction center")
76 axes[0, 2].imshow(n[sx // 2].real, **kwri)
77 axes[0, 2].set_xlabel("x")
78 axes[0, 2].set_ylabel("y")
79
80 axes[1, 2].set_title("reconstruction nucleolus")
81 axes[1, 2].imshow(n[int(sx / 2 + 2 * cfg["res"])].real, **kwri)
82 axes[1, 2].set_xlabel("x")
83 axes[1, 2].set_ylabel("y")
84
85 # color bars
86 cbkwargs = {"fraction": 0.045,
87             "format": "%.3f"}
88 plt.colorbar(phmap, ax=axes[0, 1], **cbkwargs)
89 plt.colorbar(phmap, ax=axes[1, 1], **cbkwargs)
90 plt.colorbar(rimap, ax=axes[0, 0], **cbkwargs)
91 plt.colorbar(rimap, ax=axes[1, 0], **cbkwargs)
92 plt.colorbar(rimap, ax=axes[0, 2], **cbkwargs)
93 plt.colorbar(rimap, ax=axes[1, 2], **cbkwargs)
94
95 plt.tight_layout()
96 plt.show()

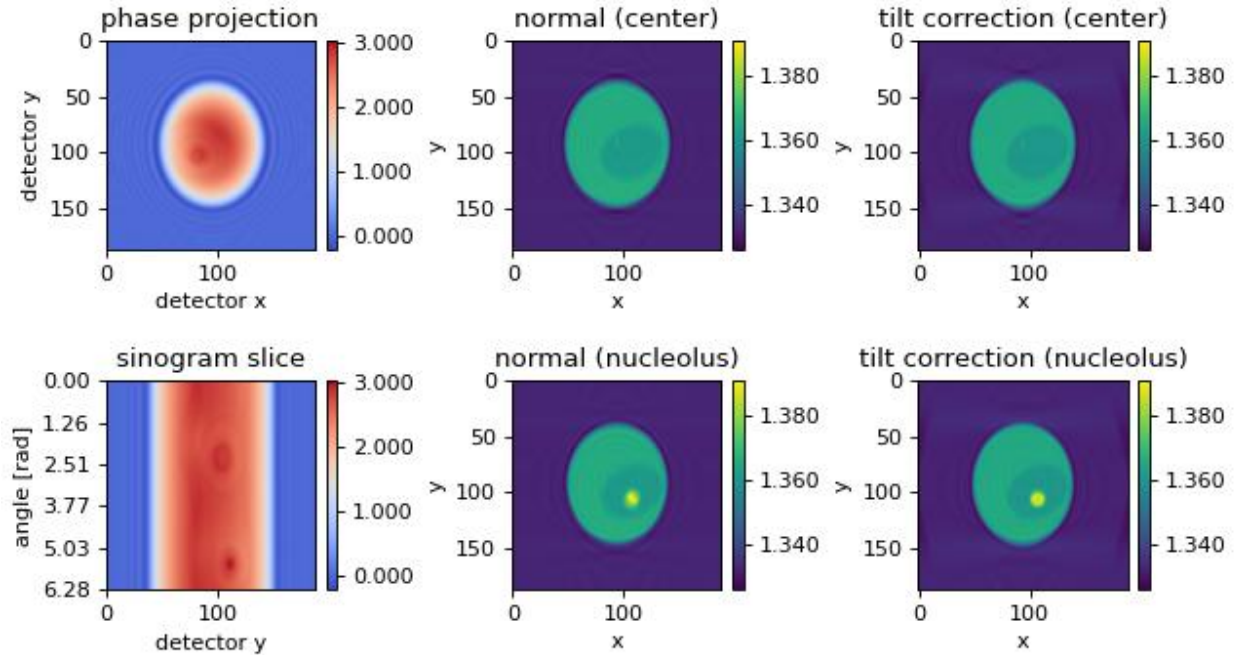
```

3.2.4 FDTD cell phantom with tilted axis of rotation

The *in silico* data set was created with the FDTD software [meep](#). The data are 2D projections of a 3D refractive index phantom that is rotated about an axis which is tilted by 0.2 rad (11.5 degrees) with respect to the imaging plane. The example showcases the method `odtbrain.backpropagate_3d_tilted()` which takes into account such a tilted axis of rotation. The data are downsampled by a factor of two. A total of 220 projections are used for the reconstruction. Note that the information required for reconstruction decreases as the tilt angle increases. If the tilt angle is 90 degrees w.r.t. the imaging plane, then we get a rotating image of a cell (not images of a rotating cell) and tomographic reconstruction is impossible. A brief description of this algorithm is given in [MSCG15].

The first column shows the measured phase, visualizing the tilt (compare to other examples). The second column shows a reconstruction that does not take into account the tilted axis of rotation; the result is a blurry reconstruction. The third column shows the improved reconstruction; the known tilted axis of rotation is used in the reconstruction process.

`backprop_from_fDTD_3d_tilted.py`



```

1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 import odtbrain as odt
5
6 from example_helper import load_data
7
8
9 if __name__ == "__main__":
10     sino, angles, phantom, cfg = \
11         load_data("fdtd_3d_sino_A220_R6.500_tilxyz0.2.tar.lzma")
12
13     A = angles.shape[0]
14
15     print("Example: Backpropagation from 3D FDTD simulations")
16     print("Refractive index of medium:", cfg["nm"])
17     print("Measurement position from object center:", cfg["ld"])
18     print("Wavelength sampling:", cfg["res"])
19     print("Axis tilt in y-z direction:", cfg["tilt_yz"])
20     print("Number of projections:", A)
21
22     print("Performing normal backpropagation.")
23     # Apply the Rytov approximation
24     sinoRytov = odt.sinogram_as_rytov(sino)
25
26     # Perform naive backpropagation
27     f_naiv = odt.backpropagate_3d(uSin=sinoRytov,
28                                   angles=angles,
29                                   res=cfg["res"],

```

(continues on next page)

(continued from previous page)

```

30         nm=cfg["nm"],
31         lD=cfg["lD"]
32     )
33
34     print("Performing tilted backpropagation.")
35     # Determine tilted axis
36     tilted_axis = [0, np.cos(cfg["tilt_yz"]), np.sin(cfg["tilt_yz"])]
37
38     # Perform tilted backpropagation
39     f_tilt = odt.backpropagate_3d_tilted(uSin=sinoRytov,
40                                         angles=angles,
41                                         res=cfg["res"],
42                                         nm=cfg["nm"],
43                                         lD=cfg["lD"],
44                                         tilted_axis=tilted_axis,
45                                         )
46
47     # compute refractive index n from object function
48     n_naiv = odt.odt_to_ri(f_naiv, res=cfg["res"], nm=cfg["nm"])
49     n_tilt = odt.odt_to_ri(f_tilt, res=cfg["res"], nm=cfg["nm"])
50
51     sx, sy, sz = n_tilt.shape
52     px, py, pz = phantom.shape
53
54     sino_phase = np.angle(sino)
55
56     # compare phantom and reconstruction in plot
57     fig, axes = plt.subplots(2, 3, figsize=(8, 4.5))
58     kwri = {"vmin": n_tilt.real.min(), "vmax": n_tilt.real.max()}
59     kwph = {"vmin": sino_phase.min(), "vmax": sino_phase.max(),
60            "cmap": "coolwarm"}
61
62     # Sinogram
63     axes[0, 0].set_title("phase projection")
64     phmap = axes[0, 0].imshow(sino_phase[A // 2, :, :], **kwph)
65     axes[0, 0].set_xlabel("detector x")
66     axes[0, 0].set_ylabel("detector y")
67
68     axes[1, 0].set_title("sinogram slice")
69     axes[1, 0].imshow(sino_phase[:, :, sino.shape[2] // 2],
70                      aspect=sino.shape[1] / sino.shape[0], **kwph)
71     axes[1, 0].set_xlabel("detector y")
72     axes[1, 0].set_ylabel("angle [rad]")
73     # set y ticks for sinogram
74     labels = np.linspace(0, 2 * np.pi, len(axes[1, 1].get_yticks()))
75     labels = ["{:.2f}".format(i) for i in labels]
76     axes[1, 0].set_yticks(np.linspace(0, len(angles), len(labels)))
77     axes[1, 0].set_yticklabels(labels)
78
79     axes[0, 1].set_title("normal (center)")
80     rimap = axes[0, 1].imshow(n_naiv[sx // 2].real, **kwri)
81     axes[0, 1].set_xlabel("x")

```

(continues on next page)

(continued from previous page)

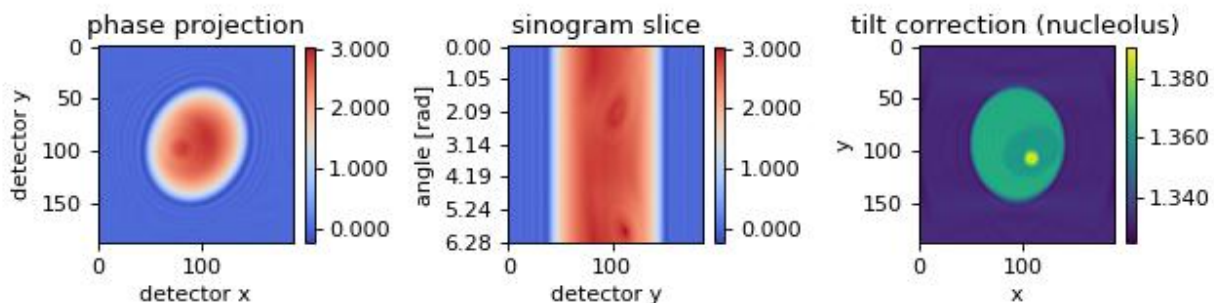
```

82 axes[0, 1].set_ylabel("y")
83
84 axes[1, 1].set_title("normal (nucleolus)")
85 axes[1, 1].imshow(n_naiv[int(sx / 2 + 2 * cfg["res"])]].real, **kwri)
86 axes[1, 1].set_xlabel("x")
87 axes[1, 1].set_ylabel("y")
88
89 axes[0, 2].set_title("tilt correction (center)")
90 axes[0, 2].imshow(n_tilt[sx // 2].real, **kwri)
91 axes[0, 2].set_xlabel("x")
92 axes[0, 2].set_ylabel("y")
93
94 axes[1, 2].set_title("tilt correction (nucleolus)")
95 axes[1, 2].imshow(n_tilt[int(sx / 2 + 2 * cfg["res"])]].real, **kwri)
96 axes[1, 2].set_xlabel("x")
97 axes[1, 2].set_ylabel("y")
98
99 # color bars
100 cbkwargs = {"fraction": 0.045,
101             "format": "%.3f"}
102 plt.colorbar(phmap, ax=axes[0, 0], **cbkwargs)
103 plt.colorbar(phmap, ax=axes[1, 0], **cbkwargs)
104 plt.colorbar(rimap, ax=axes[0, 1], **cbkwargs)
105 plt.colorbar(rimap, ax=axes[1, 1], **cbkwargs)
106 plt.colorbar(rimap, ax=axes[0, 2], **cbkwargs)
107 plt.colorbar(rimap, ax=axes[1, 2], **cbkwargs)
108
109 plt.tight_layout()
110 plt.show()

```

3.2.5 FDTD cell phantom with tilted and rolled axis of rotation

The *in silico* data set was created with the FDTD software [meep](#). The data are 2D projections of a 3D refractive index phantom that is rotated about an axis which is tilted by 0.2 rad (11.5 degrees) with respect to the imaging plane and rolled by -42 rad (-24.1 degrees) within the imaging plane. The data are the same as were used in the previous example. A brief description of this algorithm is given in [\[MSCG15\]](#).



backprop_from_fdttd_3d_tilted2.py

```

1 import matplotlib.pyplot as plt
2 import numpy as np
3 from scipy.ndimage import rotate
4
5 import odtbrain as odt
6
7 from example_helper import load_data
8
9
10 if __name__ == "__main__":
11     sino, angles, phantom, cfg = \
12         load_data("fdtd_3d_sino_A220_R6.500_tilxyz0.2.tar.lzma")
13
14     # Perform tilt by -.42 rad in detector plane
15     rotang = -0.42
16     rotkwargs = {"mode": "constant",
17                 "order": 2,
18                 "reshape": False,
19                 }
20     for ii in range(len(sino)):
21         sino[ii].real = rotate(
22             sino[ii].real, np.rad2deg(rotang), cval=1, **rotkwargs)
23         sino[ii].imag = rotate(
24             sino[ii].imag, np.rad2deg(rotang), cval=0, **rotkwargs)
25
26     A = angles.shape[0]
27
28     print("Example: Backpropagation from 3D FDTD simulations")
29     print("Refractive index of medium:", cfg["nm"])
30     print("Measurement position from object center:", cfg["lD"])
31     print("Wavelength sampling:", cfg["res"])
32     print("Axis tilt in y-z direction:", cfg["tilt_yz"])
33     print("Number of projections:", A)
34
35     # Apply the Rytov approximation
36     sinoRytov = odt.sinogram_as_rytov(sino)
37
38     # Determine tilted axis
39     tilted_axis = [0, np.cos(cfg["tilt_yz"]), np.sin(cfg["tilt_yz"])]
40     rotmat = np.array([
41         [np.cos(rotang), -np.sin(rotang), 0],
42         [np.sin(rotang), np.cos(rotang), 0],
43         [0, 0, 1],
44     ])
45     tilted_axis = np.dot(rotmat, tilted_axis)
46
47     print("Performing tilted backpropagation.")
48     # Perform tilted backpropagation
49     f_tilt = odt.backpropagate_3d_tilted(uSin=sinoRytov,
50                                         angles=angles,
51                                         res=cfg["res"],
52                                         nm=cfg["nm"],
53                                         lD=cfg["lD"],

```

(continues on next page)

(continued from previous page)

```

54         tilted_axis=tilted_axis,
55     )
56
57     # compute refractive index n from object function
58     n_tilt = odt.odt_to_ri(f_tilt, res=cfg["res"], nm=cfg["nm"])
59
60     sx, sy, sz = n_tilt.shape
61     px, py, pz = phantom.shape
62
63     sino_phase = np.angle(sino)
64
65     # compare phantom and reconstruction in plot
66     fig, axes = plt.subplots(1, 3, figsize=(8, 2.4))
67     kwri = {"vmin": n_tilt.real.min(), "vmax": n_tilt.real.max()}
68     kwph = {"vmin": sino_phase.min(), "vmax": sino_phase.max(),
69            "cmap": "coolwarm"}
70
71     # Sinogram
72     axes[0].set_title("phase projection")
73     phmap = axes[0].imshow(sino_phase[A // 2, :, :], **kwph)
74     axes[0].set_xlabel("detector x")
75     axes[0].set_ylabel("detector y")
76
77     axes[1].set_title("sinogram slice")
78     axes[1].imshow(sino_phase[:, :, sino.shape[2] // 2],
79                   aspect=sino.shape[1] / sino.shape[0], **kwph)
80     axes[1].set_xlabel("detector y")
81     axes[1].set_ylabel("angle [rad]")
82     # set y ticks for sinogram
83     labels = np.linspace(0, 2 * np.pi, len(axes[1].get_yticks()))
84     labels = ["{:.2f}".format(i) for i in labels]
85     axes[1].set_yticks(np.linspace(0, len(labels), len(axes[1].get_yticks())))
86     axes[1].set_yticklabels(labels)
87
88     axes[2].set_title("tilt correction (nucleolus)")
89     rimap = axes[2].imshow(n_tilt[int(sx / 2 + 2 * cfg["res"])].real, **kwri)
90     axes[2].set_xlabel("x")
91     axes[2].set_ylabel("y")
92
93     # color bars
94     cbkwargs = {"fraction": 0.045,
95                "format": "%.3f"}
96     plt.colorbar(phmap, ax=axes[0], **cbkwargs)
97     plt.colorbar(phmap, ax=axes[1], **cbkwargs)
98     plt.colorbar(rimap, ax=axes[2], **cbkwargs)
99
100     plt.tight_layout()
101     plt.show()

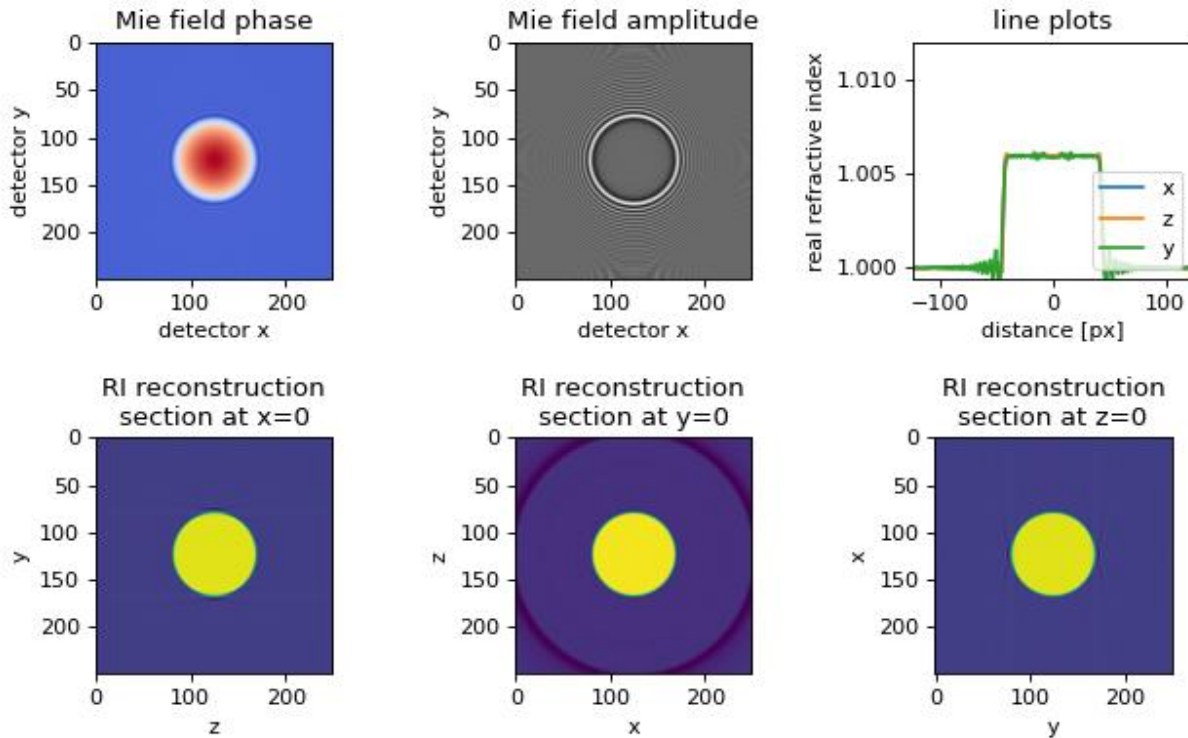
```


3.2.6 Mie sphere

The *in silico* data set was created with the Mie calculation software [GMM-field](#). The data consist of a two-dimensional projection of a sphere with radius $R = 14\lambda$, refractive index $n_{\text{sph}} = 1.006$, embedded in a medium of refractive index $n_{\text{med}} = 1.0$ onto a detector which is $l_D = 20\lambda$ away from the center of the sphere.

The package `nrefocus` must be used to numerically focus the detected field prior to the 3D backpropagation with ODTbrain. In `odtbrain.backpropagate_3d()`, the parameter `lD` must be set to zero ($l_D = 0$).

The figure shows the 3D reconstruction from Mie simulations of a perfect sphere using 200 projections. Missing angle artifacts are visible along the y -axis due to the 2π -only coverage in 3D Fourier space.



backprop_from_mie_3d_sphere.py

```

1 import matplotlib.pyplot as plt
2 import nrefocus
3 import numpy as np
4
5 import odtbrain as odt
6
7 from example_helper import load_data
8
9
10 if __name__ == "__main__":
11     Ex, cfg = load_data("mie_3d_sphere_field.zip",
12                        f_sino_imag="mie_sphere_imag.txt",
13                        f_sino_real="mie_sphere_real.txt",
14                        f_info="mie_info.txt")
15

```

(continues on next page)

(continued from previous page)

```

16  # Manually set number of angles:
17  A = 200
18
19  print("Example: Backpropagation from 3D Mie scattering")
20  print("Refractive index of medium:", cfg["nm"])
21  print("Measurement position from object center:", cfg["lD"])
22  print("Wavelength sampling:", cfg["res"])
23  print("Number of angles for reconstruction:", A)
24  print("Performing backpropagation.")
25
26  # Reconstruction angles
27  angles = np.linspace(0, 2 * np.pi, A, endpoint=False)
28
29  # Perform focusing
30  Ex = nrefocus.refocus(Ex,
31                        d=-cfg["lD"]*cfg["res"],
32                        nm=cfg["nm"],
33                        res=cfg["res"],
34                        )
35
36  # Create sinogram
37  u_sin = np.tile(Ex.flat, A).reshape(A, int(cfg["size"]), int(cfg["size"]))
38
39  # Apply the Rytov approximation
40  u_sinR = odt.sinogram_as_rytov(u_sin)
41
42  # Backpropagation
43  fR = odt.backpropagate_3d(uSin=u_sinR,
44                            angles=angles,
45                            res=cfg["res"],
46                            nm=cfg["nm"],
47                            lD=0,
48                            padfac=2.1,
49                            save_memory=True)
50
51  # RI computation
52  nR = odt.odt_to_ri(fR, cfg["res"], cfg["nm"])
53
54  # Plotting
55  fig, axes = plt.subplots(2, 3, figsize=(8, 5))
56  axes = np.array(axes).flatten()
57  # field
58  axes[0].set_title("Mie field phase")
59  axes[0].set_xlabel("detector x")
60  axes[0].set_ylabel("detector y")
61  axes[0].imshow(np.angle(Ex), cmap="coolwarm")
62  axes[1].set_title("Mie field amplitude")
63  axes[1].set_xlabel("detector x")
64  axes[1].set_ylabel("detector y")
65  axes[1].imshow(np.abs(Ex), cmap="gray")
66
67  # line plot

```

(continues on next page)

(continued from previous page)

```

68 axes[2].set_title("line plots")
69 axes[2].set_xlabel("distance [px]")
70 axes[2].set_ylabel("real refractive index")
71 center = int(cfg["size"] / 2)
72 x = np.arange(cfg["size"]) - center
73 axes[2].plot(x, nR[:, center, center].real, label="x")
74 axes[2].plot(x, nR[center, center, :].real, label="z")
75 axes[2].plot(x, nR[center, :, center].real, label="y")
76 axes[2].legend(loc=4)
77 axes[2].set_xlim((-center, center))
78 dn = abs(cfg["nsph"] - cfg["nm"])
79 axes[2].set_ylim((cfg["nm"] - dn / 10, cfg["nsph"] + dn))
80 axes[2].ticklabel_format(useOffset=False)
81
82 # cross sections
83 axes[3].set_title("RI reconstruction\section at x=0")
84 axes[3].set_xlabel("z")
85 axes[3].set_ylabel("y")
86 axes[3].imshow(nR[center, :, :].real)
87
88 axes[4].set_title("RI reconstruction\section at y=0")
89 axes[4].set_xlabel("x")
90 axes[4].set_ylabel("z")
91 axes[4].imshow(nR[:, center, :].real)
92
93 axes[5].set_title("RI reconstruction\section at z=0")
94 axes[5].set_xlabel("y")
95 axes[5].set_ylabel("x")
96 axes[5].imshow(nR[:, :, center].real)
97
98 plt.tight_layout()
99 plt.show()

```

CHANGELOG

List of changes in-between ODTbrain releases.

4.1 version 0.4.6

- fix: handle more special cases when computing weights for projections
- docs: update example scripts and images

4.2 version 0.4.5

- ci: fix build pipeline

4.3 version 0.4.4

- ref: address scipy deprecation warnings
- ref: replace print statements with warnings (#17)
- ci: switch to Python 3.10
- ci: minor cleanup

4.4 version 0.4.3

- docs: added original meep C++ simulation files in “misc” (#14)
- docs: add `if __name__ == "__main__"` guard to 3D backpropagation examples (#15)

4.5 version 0.4.2

- build: migrate to GitHub Actions
- build: setup.py test is deprecated
- docs: refurbish docs
- ref: change instances of np.int to int due to numpy deprecation warnings

4.6 version 0.4.1

- setup: bump scipy to 1.4.0 (updated QHull in griddata)

4.7 version 0.4.0

- BREAKING CHANGES:
 - renamed submodule *_preproc* to *_prepare_sino*
 - renamed submodule *_postproc* to *_translate_ri*
 - missing apple core correction is now applied to the object function (f) instead of the refractive index (n), which is the physically correct approach
 - default keyword for *padval* is now “edge” instead of *None*; the meaning is retained
- enh: added symmetric histogram apple core correction method “sh”
- fix: using “float32” dtype in 3D backpropagation lead to casting error in numexpr
- enh: improve performance when padding is disabled
- docs: minor update

4.8 version 0.3.0

- feat: basic missing apple core correction (#6)
- docs: reordered 3D examples (decreasing importance)

4.9 version 0.2.6

- fix: make phase unwrapping deterministic
- tests: remove one test of the 2D Fourier mapping algorithm due to instabilities in using `scipy.interpolate.griddata`
- ref: use *empty_aligned* instead of deprecated *n_byte_align_empty*
- docs: add hint for windows users how to run the 3D examples

4.10 version 0.2.5

- fix: reconstruction volume rotated by 180° due to floating point inaccuracies (affects *backpropagate_3d_tilted*).

4.11 version 0.2.4

- maintenance release

4.12 version 0.2.3

- enh: employ slice-wise padding to reduce the memory usage (and possibly the computation time) of
 - basic 3D backpropagation algorithm (#7)
 - 3D backpropagation with a tilted axis of rotation (#9)
- ref: replace asserts with raises (#8)
- ref: multiprocessing-based rotation does not anymore require a variable (`_shared_array`) at the top level of the module; As a result, multiprocessing-rotation should now also work on Windows.

4.13 version 0.2.2

- docs: minor update and add changelog

4.14 version 0.2.1

- fix: Allow sinogram data type other than `complex128`
- docs: Add example with experimental data (#3)
- ci: automated deployment with travis-ci

4.15 version 0.2.0

- BREAKING CHANGES:
 - Dropped support for Python 2
 - Renamed *sum_2d* to *integrate_2d*
- Refactoring (#4, #5):
 - Moved each reconstruction algorithm to a separate file
 - Modified code to comply with PEP8
 - Moved long doc strings from source to docs directory
 - Migrate from `unwrap` to `scikit-image`
 - Cleaned up example scripts

- Bugfixes:
 - Mistake in “negative-modulo” method for determination of 2PI sinogram phase offsets

4.16 version 0.1.8

- Updated documentation
- Cleaned up examples

4.17 version 0.1.7

- Move documentation from GitHub to readthedocs.io
- Add universal wheel on PyPI
- Update tests on travis with new versions of NumPy

4.18 version 0.1.6

- Bugfixes:
 - size of reconstruction volume in z too large for cases where the y-size is larger than the x-size of the sinogram images
 - *backpropagate_3d_tilted* used wrong shape of projections
 - 3D backpropagation methods did not use power-of-two padding size

4.19 version 0.1.5

- Code optimization (speed, memory) with numexpr
- New keyword argument *save_memory* for 3D reconstruction on machines with limited memory
- New keyword argument *copy* for 3D reconstruction to protect input sinogram data.

4.20 version 0.1.4

- The exponential term containing the distance between center of rotation and detector ID is now multiplied with the factor $M-I$ instead of M . This is necessary, because usually the scattered wave is normalized in both amplitude and phase (u_0) and not only amplitude a_0
- Allow angles of shape (A,1) in *backpropagate_3d_tilted*
- Set default value $ID=0$ for all reconstruction algorithms
- Improvement of documentation

4.21 version 0.1.3

- Fixes for *backpropagate_3d_tilted* when *angles* are points on the unit sphere:
 - Make sure each point is normalized
 - Correctly rotate each point w.r.t. *tilted_axis*

4.22 version 0.1.2

- Added reconstruction algorithm for tilted axis of rotation

4.23 version 0.1.1

- Support NumPy 1.10.
- Allow to weight backpropagation using keyword *weight_angles*
- Bugfix: *backpropagate_3d* with keyword *onlyreal=True* did not work
- Bugfix: *sum_2d* did not return correctly shaped array
- Code coverage is now 90%
- Added more examples to the documentation

BILBLIOGRAPHY

INDICES AND TABLES

- `genindex`
- `modindex`
- `search`

BIBLIOGRAPHY

- [KS01] Aninash C. Kak and Malcom G. Slaney. *Principles of Computerized Tomographic Imaging*. SIAM, Philadelphia, USA, 2001. ISBN 089871494X. URL: <http://www.slaney.org/pct/pct-toc.html>, doi:10.1137/1.9780898719277.
- [MSCG15] Paul Müller, Mirjam Schürmann, Chii J Chan, and Jochen Guck. Single-cell diffraction tomography with optofluidic rotation about a tilted axis. *Proc. SPIE*, 9548:95480U–95480U–5, 2015. doi:10.1117/12.2191501.
- [MSG15a] Paul Müller, Mirjam Schürmann, and Jochen Guck. ODTbrain: a Python library for full-view, dense diffraction tomography. *BMC Bioinformatics*, 16(1):1–9, 2015. doi:10.1186/s12859-015-0764-0.
- [MSG15b] Paul Müller, Mirjam Schürmann, and Jochen Guck. The Theory of Diffraction Tomography. *ArXiv e-prints*, 2015. arXiv:1507.00466v2.
- [SCG+17] M. Schürmann, G. Cojoc, S. Girardo, E. Ulbricht, J. Guck, and P. Müller. Three-dimensional correlative single-cell imaging utilizing fluorescence and refractive index tomography. *Journal of Biophotonics*, 11(3):e201700145, aug 2017. doi:10.1002/jbio.201700145.
- [TPM81] K C Tam and V Perez-Mendez. Tomographical imaging with limited-angle input. *J. Opt. Soc. Am.*, 71(5):582–592, 1981. doi:10.1364/JOSA.71.000582.
- [VDYH09] Stanislas Vertu, Jean-Jacques Delaunay, Ichiro Yamada, and Olivier Haeberlé. Diffraction microtomography with sample rotation: influence of a missing apple core in the recorded frequency space. *Central European Journal of Physics*, 7(1):22–31, 2009. doi:10.2478/s11534-008-0154-6.
- [Wol69] Emil Wolf. Three-dimensional structure determination of semi-transparent objects from holographic data. *Optics Communications*, 1(4):153–156, sep 1969. doi:10.1016/0030-4018(69)90052-2.

PYTHON MODULE INDEX

O

`odtbrain.apple`, 16

INDEX

A

`apple_core_3d()` (*in module odtbrain.apple*), 16

B

`backpropagate_2d()` (*in module odtbrain*), 8

`backpropagate_3d()` (*in module odtbrain*), 12

`backpropagate_3d_tilted()` (*in module odtbrain*), 14

C

`constraint_nn()` (*in module odtbrain.apple*), 16

`constraint_sh()` (*in module odtbrain.apple*), 16

`correct()` (*in module odtbrain.apple*), 16

`count_to_half()` (*in module odtbrain.apple*), 17

E

`ellipsoid_shell()` (*in module odtbrain.apple*), 17

`envelope_gauss()` (*in module odtbrain.apple*), 17

F

`fourier_map_2d()` (*in module odtbrain*), 9

I

`integrate_2d()` (*in module odtbrain*), 10

M

module

`odtbrain.apple`, 16

O

`odt_to_ri()` (*in module odtbrain*), 6

`odtbrain.apple`

 module, 16

`opt_to_ri()` (*in module odtbrain*), 7

S

`sinogram_as_radon()` (*in module odtbrain*), 5

`sinogram_as_rytov()` (*in module odtbrain*), 5

`spillover_region()` (*in module odtbrain.apple*), 17