
odtbrain Documentation

Release 0.2.0

Paul Müller

May 23, 2018

Contents:

1	Introduction	3
1.1	Installation	3
1.2	Theoretical background	3
1.3	Fields of Application	4
1.4	How to cite	4
2	Code reference	5
2.1	Pre- and post-processing	5
2.1.1	Pre-processing models	5
2.1.2	Post-processing (Refractive index retrieval)	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
3	Examples	17
3.1	2D examples	17
3.1.1	Mie off-center cylinder	17
3.1.2	Mie cylinder with unevenly spaced angles	20
3.1.3	Mie cylinder with incomplete angular coverage	23
3.1.4	FDTD cell phantom	26
3.2	3D examples	28
3.2.1	Mie sphere	28
3.2.2	FDTD cell phantom	30
3.2.3	FDTD cell phantom with tilted axis of rotation	33
3.2.4	FDTD cell phantom with tilted and rolled axis of rotation	35
4	Bibliography	39
5	Indices and tables	41
	Bibliography	43

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 0.2.0.

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*].

2.1 Pre- and post-processing

<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 Pre-processing models

Tomographic data sets consist of detector images for different rotational positions ϕ_0 of the object. Pre-processing in this context 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 Post-processing (Refractive index retrieval)

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 $2\pi i$ due to the unwrapping algorithm. In that case, this value should be a multiple of $2\pi i$. If *padval* is *None*, then the edge values are used for padding (see documentation of `numpy.pad()`).
- **max_count** (*count*,) – 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 2π , but an equidistant coverage over 2π 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 2π coverage with the union set $\{X\} \cup \{X+\pi\}$.

- **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.
- **max_count** (*count*,) – 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 *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`).

2.2.4 Direct sum

`odtbrain.integrate_2d(uSin, angles, res, nm, lD=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 .
- **lD** (*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.
- **max_count** (*count*,) – 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=None, intp_order=2, dtype=None, num_cores=4, 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*) – The value used for padding. This is important for the Rytov approximation, where an approximat zero in the phase might translate to $2\pi i$ due to the unwrapping algorithm. In that case, this value should be a multiple of $2\pi i$. If *padval* is *None*, then the edge values are used for padding (see documentation of `numpy.pad()`).
- **interp_order** (*int between 0 and 5*) – Order of the interpolation for rotation. See `scipy.ndimage.interpolation.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.
- **max_count** (*count,*) – 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 *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`).

2.3.3 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, on-
lyreal=False, padding=(True, True), padfac=1.75, pad-
val=None, intp_order=2, dtype=None, num_cores=4,
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 for 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*) – The value used for padding. This is important for the Rytov approximation, where an approximate zero in the phase might translate to $2\pi i$ due to the unwrapping algorithm. In that case, this value should be a multiple of $2\pi i$. If *padval* is *None*, then the edge values are used for padding (see documentation of `numpy.pad()`).
- **intp_order** (*int between 0 and 5*) – Order of the interpolation for rotation. See `scipy.ndimage.interpolation.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.

- **max_count** (*count,*) – 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`).

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

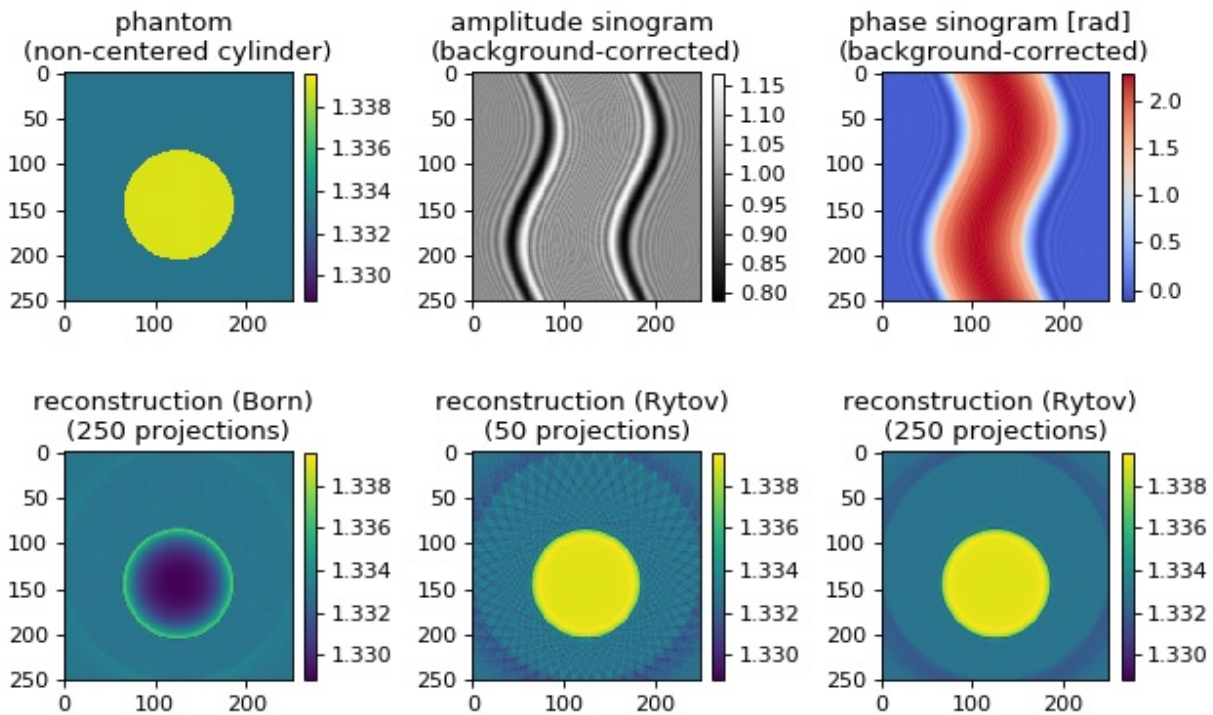
The *in silico* data set was created with the software [miefield](#). The data are 1D projections of an off-center cylinder of constant refractive index. The Born approximation is error-prone due to a relatively large radius of the cylinder (30 wavelengths) and a refractive index difference of 0.006 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.

The figure shows different reconstruction approaches for a cylinder from data computed with Mie theory.

`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")
```

(continues on next page)



(continued from previous page)

```

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 print("Example: Backpropagation from 2D Mie simulations")
32 print("Refractive index of medium:", cfg["nmed"])
33 print("Measurement position from object center:", cfg["lD"])
34 print("Wavelength sampling:", cfg["res"])
35 print("Performing backpropagation.")
36
37 # Set measurement parameters
38 # Compute scattered field from cylinder
39 radius = cfg["radius"] # wavelengths
40 nmed = cfg["nmed"]
41 ncyl = cfg["ncyl"]
42

```

(continues on next page)

(continued from previous page)

```

43 lD = cfg["lD"] # measurement distance in wavelengths
44 lC = cfg["lC"] # displacement from center of image
45 size = cfg["size"]
46 res = cfg["res"] # px/wavelengths
47 A = cfg["A"] # number of projections
48
49 x = np.arange(size) - size / 2
50 X, Y = np.meshgrid(x, x)
51 rad_px = radius * res
52 phantom = np.array(((Y - lC * res)**2 + X**2) < rad_px**2,
53                    dtype=np.float) * (ncyl - nmed) + nmed
54
55 # Born
56 u_sinB = (sino / u0 * u0_single - u0_single) # fake born
57 fB = odt.backpropagate_2d(u_sinB, angles, res, nmed, lD * 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, lD * 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, lD * 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

```

(continues on next page)

(continued from previous page)

```

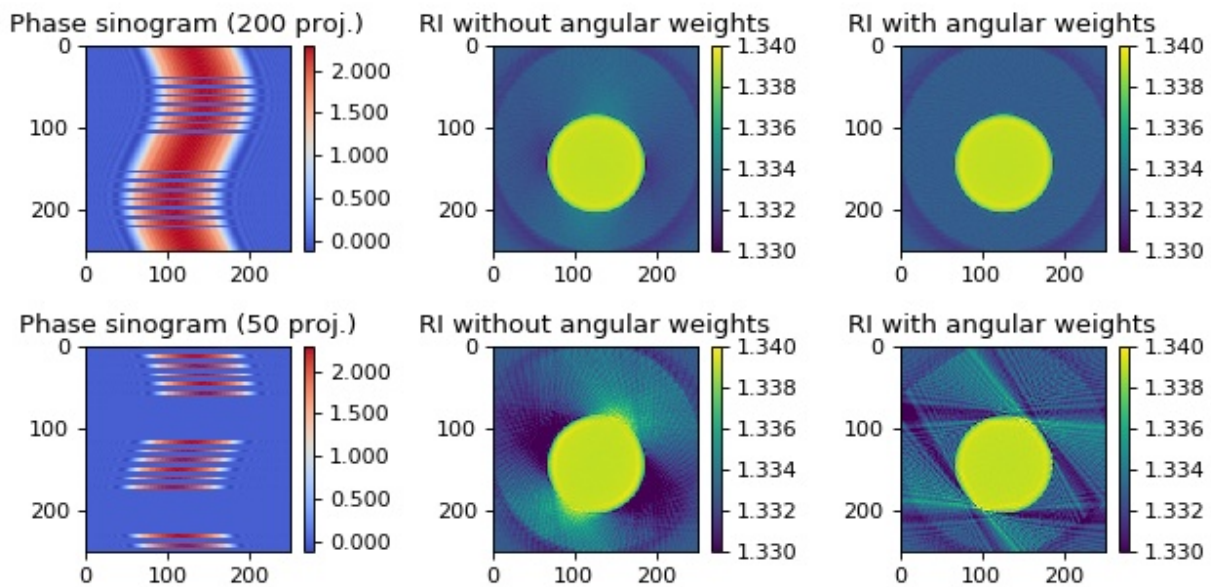
100 # color bars
101 cbkwargs = {"fraction": 0.045}
102 plt.colorbar(phantommap, ax=axes[0], **cbkwargs)
103 plt.colorbar(amplmap, ax=axes[1], **cbkwargs)
104 plt.colorbar(phasemap, ax=axes[2], **cbkwargs)
105 plt.colorbar(phantommap, ax=axes[3], **cbkwargs)
106 plt.colorbar(phantommap, ax=axes[4], **cbkwargs)
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.

The illustrates the impact of angular weighting on backpropagation with the Rytov approximation.



backprop_from_mie_2d_weights_angles.py

```

1 import matplotlib.pyplot as plt
2 import numpy as np
3 import unwrap
4
5 import odtbrain as odt
6

```

(continues on next page)

(continued from previous page)

```

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",
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=np.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.unwrap(np.angle(sino / u0))

```

(continues on next page)

(continued from previous page)

```

64 ph200[remove200] = 0
65
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.unwrap(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
93 fig, axes = plt.subplots(2, 3, figsize=(8, 4))
94 axes = np.array(axes).flatten()
95
96 phmap = axes[0].imshow(ph200, **kw_ph)
97 axes[0].set_title("Phase sinogram (200 proj.)")
98
99 rimap = axes[1].imshow(nR200nw.real, **kw_ri)
100 axes[1].set_title("RI without angular weights")
101
102 axes[2].imshow(nR200.real, **kw_ri)
103 axes[2].set_title("RI with angular weights")
104
105 axes[3].imshow(ph50, **kw_ph)
106 axes[3].set_title("Phase sinogram (50 proj.)")
107
108 axes[4].imshow(nR50nw.real, **kw_ri)
109 axes[4].set_title("RI without angular weights")
110
111 axes[5].imshow(nR50.real, **kw_ri)
112 axes[5].set_title("RI with angular weights")
113
114 # color bars
115 cbkwargs = {"fraction": 0.045,
116            "format": "%.3f"}
117 plt.colorbar(phmap, ax=axes[0], **cbkwargs)
118 plt.colorbar(phmap, ax=axes[3], **cbkwargs)
119 plt.colorbar(rimap, ax=axes[1], **cbkwargs)
120 plt.colorbar(rimap, ax=axes[2], **cbkwargs)

```

(continues on next page)

(continued from previous page)

```

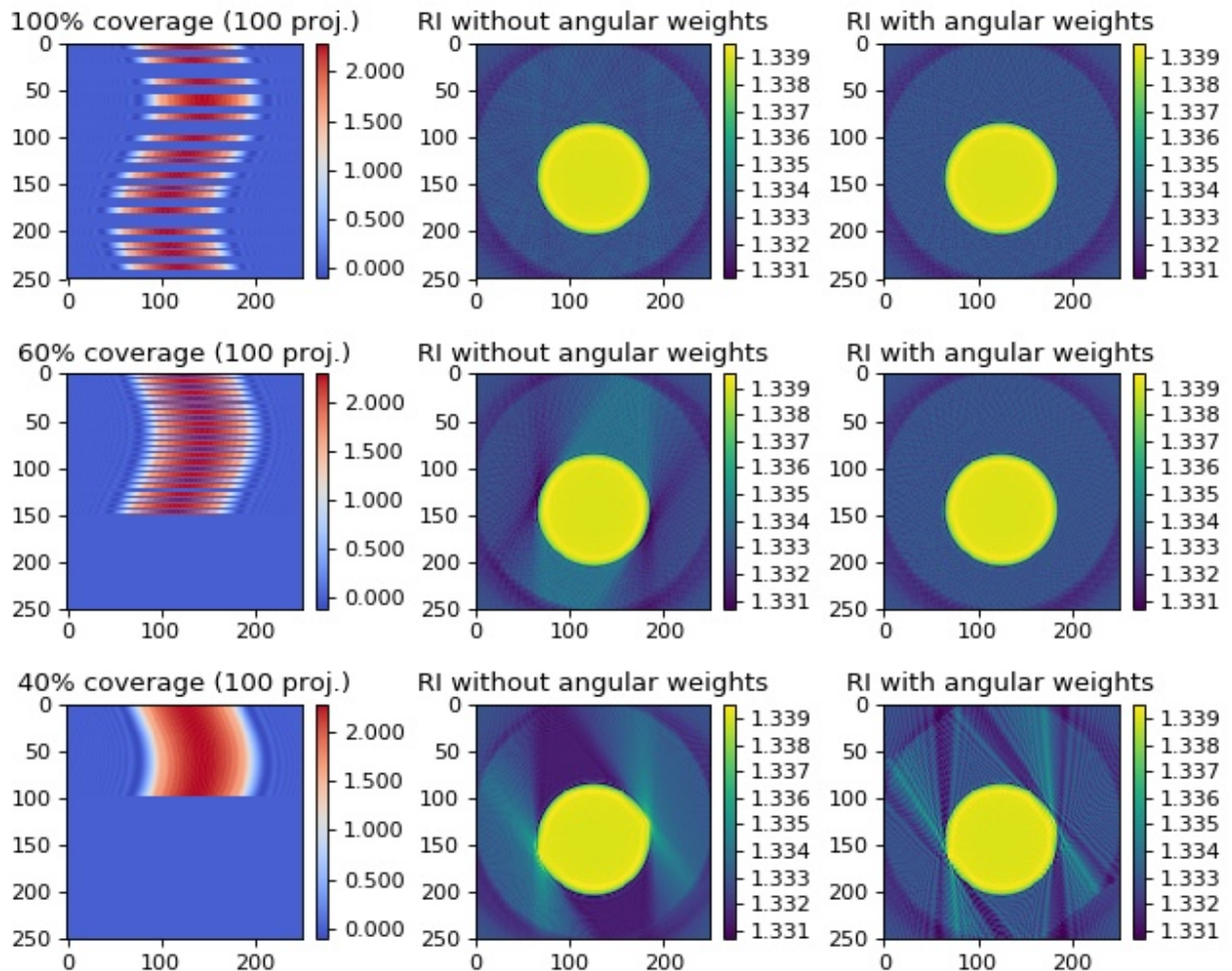
121 plt.colorbar(rimap, ax=axes[5], **cbkwargs)
122 plt.colorbar(rimap, ax=axes[4], **cbkwargs)
123
124 plt.tight_layout()
125 plt.show()

```

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",
29                        f_sino_real="u0_single_real.txt")
30
31  print("Example: Backpropagation from 2D FDTD simulations")
32  print("Refractive index of medium:", cfg["nmed"])
33  print("Measurement position from object center:", cfg["lD"])
34  print("Wavelength sampling:", cfg["res"])
35  print("Performing backpropagation.")
36
37  # Set measurement parameters
38  # Compute scattered field from cylinder
39  radius = cfg["radius"] # wavelengths
40  nmed = cfg["nmed"]
41  ncyl = cfg["ncyl"]
42
43  lD = cfg["lD"] # measurement distance in wavelengths
44  lC = cfg["lC"] # displacement from center of image
45  size = cfg["size"]
46  res = cfg["res"] # px/wavelengths
47  A = cfg["A"] # number of projections
48
49  x = np.arange(size) - size / 2.0
50  X, Y = np.meshgrid(x, x)
51  rad_px = radius * res
52  phantom = np.array(((Y - lC * res)**2 + X**2) < rad_px **
53                     2, dtype=np.float) * (ncyl - nmed) + nmed
54
55  u_sinR = odt.sinogram_as_rytov(sino / u0)
56
57  # Rytov 100 projections evenly distributed
58  removeeven = np.argsort(angles % .002)[:150]

```

(continues on next page)

(continued from previous page)

```

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, 1D * res)
63 nReven = odt.odt_to_ri(fReven, res, nmed)
64 fRevennw = odt.backpropagate_2d(
65     u_sinReven, angleseven, res, nmed, 1D * 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, 1D * res)
77 nRmiss = odt.odt_to_ri(fRmiss, res, nmed)
78 fRmissnw = odt.backpropagate_2d(
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
103 fig, axes = plt.subplots(3, 3, figsize=(8, 6.5))
104
105 axes[0, 0].set_title("100% coverage ({} proj.)".format(angleseven.shape[0]))
106 phmap = axes[0, 0].imshow(pheven, **kw_ph)
107
108 axes[0, 1].set_title("RI without angular weights")
109 rimap = axes[0, 1].imshow(nRevennw.real, **kw_ri)
110
111 axes[0, 2].set_title("RI with angular weights")
112 rimap = axes[0, 2].imshow(nReven.real, **kw_ri)
113

```

(continues on next page)

(continued from previous page)

```

114 axes[1, 0].set_title("60% coverage ({} proj.)".format(anglesmiss.shape[0]))
115 axes[1, 0].imshow(phmiss, **kw_ph)
116
117 axes[1, 1].set_title("RI without angular weights")
118 axes[1, 1].imshow(nRmissnw.real, **kw_ri)
119
120 axes[1, 2].set_title("RI with angular weights")
121 axes[1, 2].imshow(nRmiss.real, **kw_ri)
122
123 axes[2, 0].set_title("40% coverage ({} proj.)".format(anglesbad.shape[0]))
124 axes[2, 0].imshow(phbad, **kw_ph)
125
126 axes[2, 1].set_title("RI without angular weights")
127 axes[2, 1].imshow(nRbadnw.real, **kw_ri)
128
129 axes[2, 2].set_title("RI with angular weights")
130 axes[2, 2].imshow(nRbad.real, **kw_ri)
131
132 # color bars
133 cbkwargs = {"fraction": 0.045,
134            "format": "%.3f"}
135 plt.colorbar(phmap, ax=axes[0, 0], **cbkwargs)
136 plt.colorbar(phmap, ax=axes[1, 0], **cbkwargs)
137 plt.colorbar(phmap, ax=axes[2, 0], **cbkwargs)
138 plt.colorbar(rimap, ax=axes[0, 1], **cbkwargs)
139 plt.colorbar(rimap, ax=axes[1, 1], **cbkwargs)
140 plt.colorbar(rimap, ax=axes[2, 1], **cbkwargs)
141 plt.colorbar(rimap, ax=axes[0, 2], **cbkwargs)
142 plt.colorbar(rimap, ax=axes[1, 2], **cbkwargs)
143 plt.colorbar(rimap, ax=axes[2, 2], **cbkwargs)
144
145 plt.tight_layout()
146 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.

The figure shows a 2D reconstruction from FDTD data created by [meep](#) simulations.

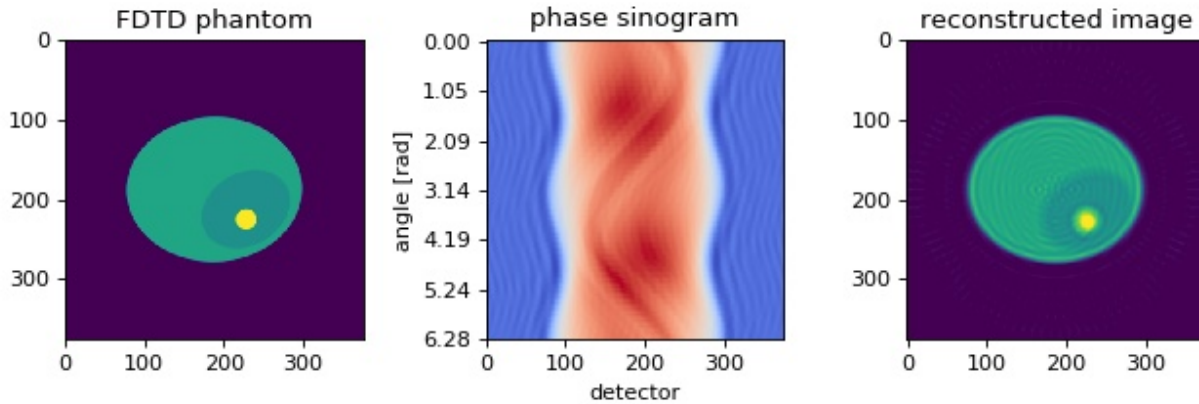
backprop_from_fDTD_2d.py

```

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",
11                                    f_sino_real="fDTD_real.txt",

```

(continues on next page)



(continued from previous page)

```

12         f_info="fddt_info.txt",
13         f_phantom="fddt_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())

```

(continues on next page)

(continued from previous page)

```

52
53 # set y ticks for sinogram
54 labels = np.linspace(0, 2 * np.pi, len(axes[1].get_yticks()))
55 labels = ["{:0.2f}".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

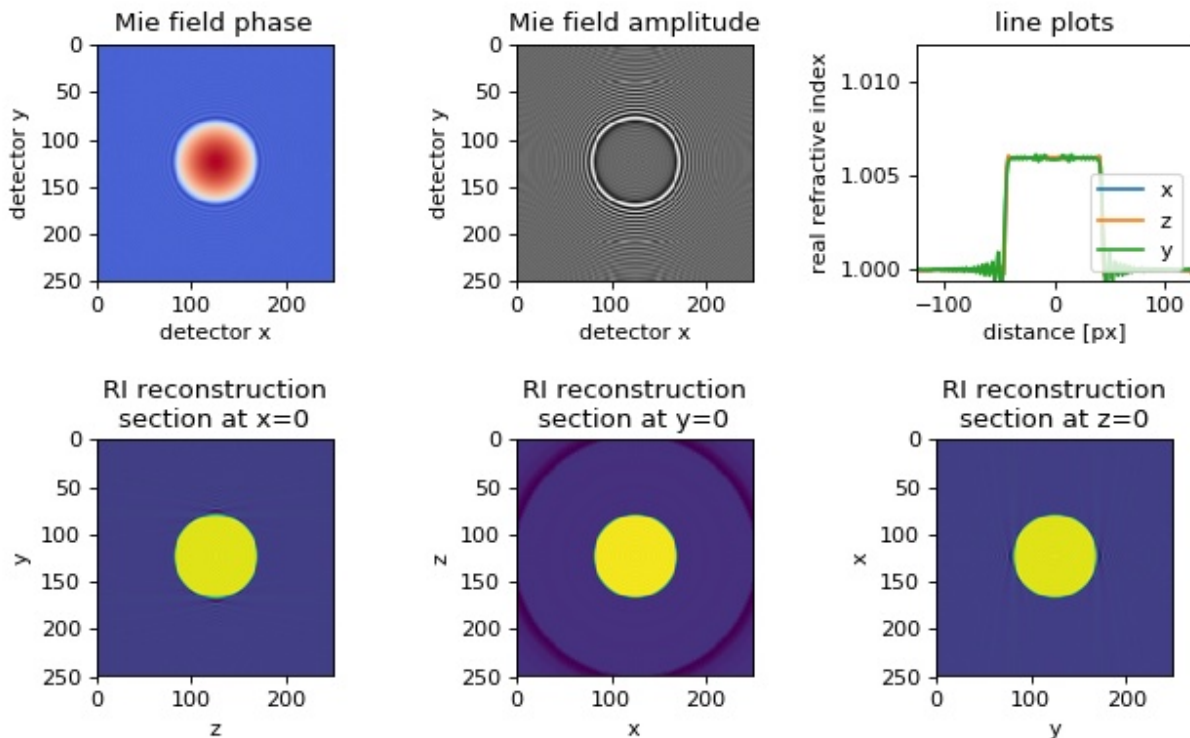
These examples require raw data which are automatically downloaded from the source repository by the script `example_helper.py`. Please make sure that this script is present in the example script folder.

3.2.1 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 `l_D` 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 Ex, cfg = load_data("mie_3d_sphere_field.zip",
11                    f_sino_imag="mie_sphere_imag.txt",
12                    f_sino_real="mie_sphere_real.txt",
13                    f_info="mie_info.txt")
14
15 # Manually set number of angles:
16 A = 200
17
18 print("Example: Backpropagation from 3D Mie scattering")
19 print("Refractive index of medium:", cfg["nm"])
20 print("Measurement position from object center:", cfg["lD"])
21 print("Wavelength sampling:", cfg["res"])
22 print("Number of angles for reconstruction:", A)
23 print("Performing backpropagation.")
24
25 # Reconstruction angles
26 angles = np.linspace(0, 2 * np.pi, A, endpoint=False)
27
28 # Perform focusing
29 Ex = nrefocus.refocus(Ex,
30                       d=-cfg["lD"]*cfg["res"],
31                       nm=cfg["nm"],
32                       res=cfg["res"],
33                       )
34
35 # Create sinogram
36 u_sin = np.tile(Ex.flat, A).reshape(A, int(cfg["size"]), int(cfg["size"]))
37
38 # Apply the Rytov approximation
39 u_sinR = odt.sinogram_as_rytov(u_sin)
40
41 # Backpropagation
42 fR = odt.backpropagate_3d(uSin=u_sinR,
43                           angles=angles,
44                           res=cfg["res"],
45                           nm=cfg["nm"],
46                           lD=0,
47                           padfac=2.1,
48                           save_memory=True)
49
50 # RI computation
51 nR = odt.odt_to_ri(fR, cfg["res"], cfg["nm"])
52
53 # Plotting
54 fig, axes = plt.subplots(2, 3, figsize=(8, 5))
55 axes = np.array(axes).flatten()
56 # field

```

(continues on next page)

(continued from previous page)

```

57 axes[0].set_title("Mie field phase")
58 axes[0].set_xlabel("detector x")
59 axes[0].set_ylabel("detector y")
60 axes[0].imshow(np.angle(Ex), cmap="coolwarm")
61 axes[1].set_title("Mie field amplitude")
62 axes[1].set_xlabel("detector x")
63 axes[1].set_ylabel("detector y")
64 axes[1].imshow(np.abs(Ex), cmap="gray")
65
66 # line plot
67 axes[2].set_title("line plots")
68 axes[2].set_xlabel("distance [px]")
69 axes[2].set_ylabel("real refractive index")
70 center = int(cfg["size"] / 2)
71 x = np.arange(cfg["size"]) - center
72 axes[2].plot(x, nR[:, center, center].real, label="x")
73 axes[2].plot(x, nR[center, center, :].real, label="z")
74 axes[2].plot(x, nR[center, :, center].real, label="y")
75 axes[2].legend(loc=4)
76 axes[2].set_xlim((-center, center))
77 dn = abs(cfg["nsph"] - cfg["nm"])
78 axes[2].set_ylim((cfg["nm"] - dn / 10, cfg["nsph"] + dn))
79 axes[2].ticklabel_format(useOffset=False)
80
81 # cross sections
82 axes[3].set_title("RI reconstruction\nsection at x=0")
83 axes[3].set_xlabel("z")
84 axes[3].set_ylabel("y")
85 axes[3].imshow(nR[center, :, :].real)
86
87 axes[4].set_title("RI reconstruction\nsection at y=0")
88 axes[4].set_xlabel("x")
89 axes[4].set_ylabel("z")
90 axes[4].imshow(nR[:, center, :].real)
91
92 axes[5].set_title("RI reconstruction\nsection at z=0")
93 axes[5].set_xlabel("y")
94 axes[5].set_ylabel("x")
95 axes[5].imshow(nR[:, :, center].real)
96
97 plt.tight_layout()
98 plt.show()

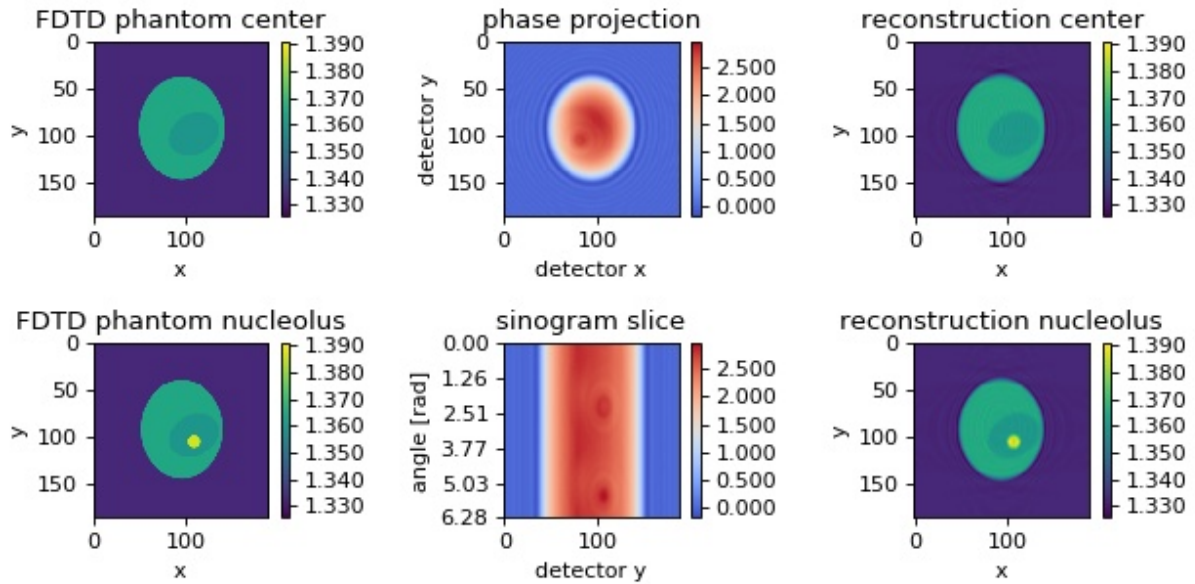
```

3.2.2 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 [\[MSG15a\]](#).

The figure shows a 3D reconstruction from FDTD data created with [meep](#) simulations.

`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
9 sino, angles, phantom, cfg = \
10     load_data("fdtd_3d_sino_A180_R6.500.tar.lzma")
11
12 A = angles.shape[0]
13
14 print("Example: Backpropagation from 3D FDTD simulations")
15 print("Refractive index of medium:", cfg["nm"])
16 print("Measurement position from object center:", cfg["lD"])
17 print("Wavelength sampling:", cfg["res"])
18 print("Number of projections:", A)
19 print("Performing backpropagation.")
20
21 # Apply the Rytov approximation
22 sinoRytov = odt.sinogram_as_rytov(sino)
23
24 # perform backpropagation to obtain object function f
25 f = odt.backpropagate_3d(uSin=sinoRytov,
26                          angles=angles,
27                          res=cfg["res"],
28                          nm=cfg["nm"],
29                          lD=cfg["lD"]
30                          )
31
32 # compute refractive index n from object function
33 n = odt.odt_to_ri(f, res=cfg["res"], nm=cfg["nm"])
34

```

(continues on next page)

(continued from previous page)

```

35 sx, sy, sz = n.shape
36 px, py, pz = phantom.shape
37
38 sino_phase = np.angle(sino)
39
40 # compare phantom and reconstruction in plot
41 fig, axes = plt.subplots(2, 3, figsize=(8, 4))
42 kwri = {"vmin": n.real.min(), "vmax": n.real.max()}
43 kwph = {"vmin": sino_phase.min(), "vmax": sino_phase.max(),
44        "cmap": "coolwarm"}
45
46 # Phantom
47 axes[0, 0].set_title("FDTD phantom center")
48 rimap = axes[0, 0].imshow(phantom[px // 2], **kwri)
49 axes[0, 0].set_xlabel("x")
50 axes[0, 0].set_ylabel("y")
51
52 axes[1, 0].set_title("FDTD phantom nucleolus")
53 axes[1, 0].imshow(phantom[int(px / 2 + 2 * cfg["res"])], **kwri)
54 axes[1, 0].set_xlabel("x")
55 axes[1, 0].set_ylabel("y")
56
57 # Sinogram
58 axes[0, 1].set_title("phase projection")
59 phmap = axes[0, 1].imshow(sino_phase[A // 2, :, :], **kwph)
60 axes[0, 1].set_xlabel("detector x")
61 axes[0, 1].set_ylabel("detector y")
62
63 axes[1, 1].set_title("sinogram slice")
64 axes[1, 1].imshow(sino_phase[:, :, sino.shape[2] // 2],
65                  aspect=sino.shape[1] / sino.shape[0], **kwph)
66 axes[1, 1].set_xlabel("detector y")
67 axes[1, 1].set_ylabel("angle [rad]")
68 # set y ticks for sinogram
69 labels = np.linspace(0, 2 * np.pi, len(axes[1, 1].get_yticks()))
70 labels = ["{:0.2f}".format(i) for i in labels]
71 axes[1, 1].set_yticks(np.linspace(0, len(labels), len(labels)))
72 axes[1, 1].set_yticklabels(labels)
73
74 axes[0, 2].set_title("reconstruction center")
75 axes[0, 2].imshow(n[sx // 2].real, **kwri)
76 axes[0, 2].set_xlabel("x")
77 axes[0, 2].set_ylabel("y")
78
79 axes[1, 2].set_title("reconstruction nucleolus")
80 axes[1, 2].imshow(n[int(sx / 2 + 2 * cfg["res"])].real, **kwri)
81 axes[1, 2].set_xlabel("x")
82 axes[1, 2].set_ylabel("y")
83
84 # color bars
85 cbkwargs = {"fraction": 0.045,
86            "format": "%.3f"}
87 plt.colorbar(phmap, ax=axes[0, 1], **cbkwargs)
88 plt.colorbar(phmap, ax=axes[1, 1], **cbkwargs)
89 plt.colorbar(rimap, ax=axes[0, 0], **cbkwargs)
90 plt.colorbar(rimap, ax=axes[1, 0], **cbkwargs)
91 plt.colorbar(rimap, ax=axes[0, 2], **cbkwargs)

```

(continues on next page)

(continued from previous page)

```

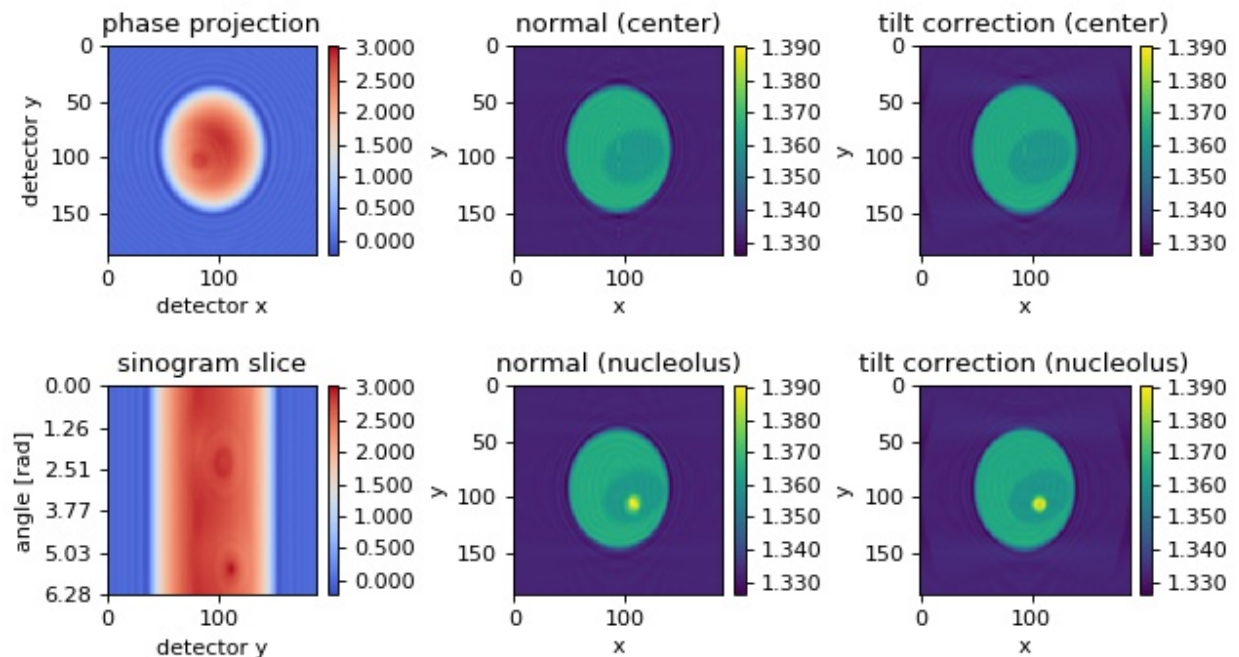
92 plt.colorbar(rimap, ax=axes[1, 2], **cbkwargs)
93
94 plt.tight_layout()
95 plt.show()

```

3.2.3 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 figure shows a 3D reconstruction from FDTD data created by `meep` simulations. 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

```

(continues on next page)

(continued from previous page)

```

8  sino, angles, phantom, cfg = \
9      load_data("fdtd_3d_sino_A220_R6.500_tilt_yz0.2.tar.lzma")
10
11  A = angles.shape[0]
12
13  print("Example: Backpropagation from 3D FDTD simulations")
14  print("Refractive index of medium:", cfg["nm"])
15  print("Measurement position from object center:", cfg["lD"])
16  print("Wavelength sampling:", cfg["res"])
17  print("Axis tilt in y-z direction:", cfg["tilt_yz"])
18  print("Number of projections:", A)
19
20  print("Performing normal backpropagation.")
21  # Apply the Rytov approximation
22  sinoRytov = odt.sinogram_as_rytov(sino)
23
24  # Perform naive backpropagation
25  f_naiv = odt.backpropagate_3d(uSin=sinoRytov,
26                                angles=angles,
27                                res=cfg["res"],
28                                nm=cfg["nm"],
29                                lD=cfg["lD"]
30                                )
31
32  print("Performing tilted backpropagation.")
33  # Determine tilted axis
34  tilted_axis = [0, np.cos(cfg["tilt_yz"]), np.sin(cfg["tilt_yz"])]
35
36  # Perform tilted backpropagation
37  f_tilt = odt.backpropagate_3d_tilted(uSin=sinoRytov,
38                                       angles=angles,
39                                       res=cfg["res"],
40                                       nm=cfg["nm"],
41                                       lD=cfg["lD"],
42                                       tilted_axis=tilted_axis,
43                                       )
44
45  # compute refractive index n from object function
46  n_naiv = odt.odt_to_ri(f_naiv, res=cfg["res"], nm=cfg["nm"])
47  n_tilt = odt.odt_to_ri(f_tilt, res=cfg["res"], nm=cfg["nm"])
48
49  sx, sy, sz = n_tilt.shape
50  px, py, pz = phantom.shape
51
52  sino_phase = np.angle(sino)
53
54  # compare phantom and reconstruction in plot
55  fig, axes = plt.subplots(2, 3, figsize=(8, 4.5))
56  kwri = {"vmin": n_tilt.real.min(), "vmax": n_tilt.real.max()}
57  kwph = {"vmin": sino_phase.min(), "vmax": sino_phase.max(),
58         "cmap": "coolwarm"}
59
60  # Sinogram
61  axes[0, 0].set_title("phase projection")
62  phmap = axes[0, 0].imshow(sino_phase[A // 2, :, :], **kwph)
63  axes[0, 0].set_xlabel("detector x")
64  axes[0, 0].set_ylabel("detector y")

```

(continues on next page)

(continued from previous page)

```

65
66 axes[1, 0].set_title("sinogram slice")
67 axes[1, 0].imshow(sino_phase[:, :, sino.shape[2] // 2],
68                   aspect=sino.shape[1] / sino.shape[0], **kwph)
69 axes[1, 0].set_xlabel("detector y")
70 axes[1, 0].set_ylabel("angle [rad]")
71 # set y ticks for sinogram
72 labels = np.linspace(0, 2 * np.pi, len(axes[1, 1].get_yticks()))
73 labels = ["{:0.2f}".format(i) for i in labels]
74 axes[1, 0].set_yticks(np.linspace(0, len(angles), len(labels)))
75 axes[1, 0].set_yticklabels(labels)
76
77 axes[0, 1].set_title("normal (center)")
78 rimap = axes[0, 1].imshow(n_naiv[sx // 2].real, **kwri)
79 axes[0, 1].set_xlabel("x")
80 axes[0, 1].set_ylabel("y")
81
82 axes[1, 1].set_title("normal (nucleolus)")
83 axes[1, 1].imshow(n_naiv[int(sx / 2 + 2 * cfg["res"])).real, **kwri)
84 axes[1, 1].set_xlabel("x")
85 axes[1, 1].set_ylabel("y")
86
87 axes[0, 2].set_title("tilt correction (center)")
88 axes[0, 2].imshow(n_tilt[sx // 2].real, **kwri)
89 axes[0, 2].set_xlabel("x")
90 axes[0, 2].set_ylabel("y")
91
92 axes[1, 2].set_title("tilt correction (nucleolus)")
93 axes[1, 2].imshow(n_tilt[int(sx / 2 + 2 * cfg["res"])).real, **kwri)
94 axes[1, 2].set_xlabel("x")
95 axes[1, 2].set_ylabel("y")
96
97 # color bars
98 cbkwargs = {"fraction": 0.045,
99            "format": "%0.3f"}
100 plt.colorbar(phmap, ax=axes[0, 0], **cbkwargs)
101 plt.colorbar(phmap, ax=axes[1, 0], **cbkwargs)
102 plt.colorbar(rimap, ax=axes[0, 1], **cbkwargs)
103 plt.colorbar(rimap, ax=axes[1, 1], **cbkwargs)
104 plt.colorbar(rimap, ax=axes[0, 2], **cbkwargs)
105 plt.colorbar(rimap, ax=axes[1, 2], **cbkwargs)
106
107 plt.tight_layout()
108 plt.show()

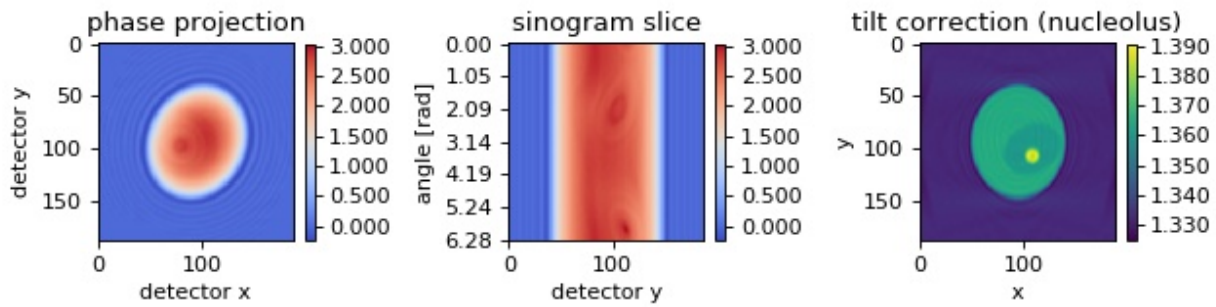
```

3.2.4 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 -0.42 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 [\[MSCG15\]](#).

The figure shows the 3D reconstruction from FDTD data created by [meep](#) simulations. The known tilted axis of rotation is used in the reconstruction process.

`backprop_from_fDTD_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 sino, angles, phantom, cfg = \
11     load_data("fdtd_3d_sino_A220_R6.500_tilt_yz0.2.tar.lzma")
12
13 # Perform tilt by -0.42 rad in detector plane
14 rotang = -0.42
15 rotkwargs = {"mode": "constant",
16              "order": 2,
17              "reshape": False,
18              }
19 for ii in range(len(sino)):
20     sino[ii].real = rotate(
21         sino[ii].real, np.rad2deg(rotang), cval=1, **rotkwargs)
22     sino[ii].imag = rotate(
23         sino[ii].imag, np.rad2deg(rotang), cval=0, **rotkwargs)
24
25 A = angles.shape[0]
26
27 print("Example: Backpropagation from 3D FDTD simulations")
28 print("Refractive index of medium:", cfg["nm"])
29 print("Measurement position from object center:", cfg["1D"])
30 print("Wavelength sampling:", cfg["res"])
31 print("Axis tilt in y-z direction:", cfg["tilt_yz"])
32 print("Number of projections:", A)
33
34 # Apply the Rytov approximation
35 sinoRytov = odt.sinogram_as_rytov(sino)
36
37 # Determine tilted axis
38 tilted_axis = [0, np.cos(cfg["tilt_yz"]), np.sin(cfg["tilt_yz"])]
39 rotmat = np.array([
40     [np.cos(rotang), -np.sin(rotang), 0],
41     [np.sin(rotang), np.cos(rotang), 0],
42     [0, 0, 1],
43 ])

```

(continues on next page)

(continued from previous page)

```

44 tilted_axis = np.dot(rotmat, tilted_axis)
45
46 print("Performing tilted backpropagation.")
47 # Perform tilted backpropagation
48 f_tilt = odt.backpropagate_3d_tilted(uSin=sinoRytov,
49                                     angles=angles,
50                                     res=cfg["res"],
51                                     nm=cfg["nm"],
52                                     lD=cfg["lD"],
53                                     tilted_axis=tilted_axis,
54                                     )
55
56 # compute refractive index n from object function
57 n_tilt = odt.odt_to_ri(f_tilt, res=cfg["res"], nm=cfg["nm"])
58
59 sx, sy, sz = n_tilt.shape
60 px, py, pz = phantom.shape
61
62 sino_phase = np.angle(sino)
63
64 # compare phantom and reconstruction in plot
65 fig, axes = plt.subplots(1, 3, figsize=(8, 2.4))
66 kwri = {"vmin": n_tilt.real.min(), "vmax": n_tilt.real.max()}
67 kwph = {"vmin": sino_phase.min(), "vmax": sino_phase.max(),
68         "cmap": "coolwarm"}
69
70 # Sinogram
71 axes[0].set_title("phase projection")
72 phmap = axes[0].imshow(sino_phase[A // 2, :, :], **kwph)
73 axes[0].set_xlabel("detector x")
74 axes[0].set_ylabel("detector y")
75
76 axes[1].set_title("sinogram slice")
77 axes[1].imshow(sino_phase[:, :, sino.shape[2] // 2],
78               aspect=sino.shape[1] / sino.shape[0], **kwph)
79 axes[1].set_xlabel("detector y")
80 axes[1].set_ylabel("angle [rad]")
81 # set y ticks for sinogram
82 labels = np.linspace(0, 2 * np.pi, len(axes[1].get_yticks()))
83 labels = ["{:.2f}".format(i) for i in labels]
84 axes[1].set_yticks(np.linspace(0, len(angles), len(labels)))
85 axes[1].set_yticklabels(labels)
86
87 axes[2].set_title("tilt correction (nucleolus)")
88 rimap = axes[2].imshow(n_tilt[int(sx / 2 + 2 * cfg["res"])].real, **kwri)
89 axes[2].set_xlabel("x")
90 axes[2].set_ylabel("y")
91
92 # color bars
93 cbkwargs = {"fraction": 0.045,
94             "format": "%.3f"}
95 plt.colorbar(phmap, ax=axes[0], **cbkwargs)
96 plt.colorbar(phmap, ax=axes[1], **cbkwargs)
97 plt.colorbar(rimap, ax=axes[2], **cbkwargs)
98
99 plt.tight_layout()
100 plt.show()

```


CHAPTER 4

Bibliography

CHAPTER 5

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.
- [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.
- [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.

B

`backpropagate_2d()` (in module `odtbrain`), [8](#)
`backpropagate_3d()` (in module `odtbrain`), [12](#)
`backpropagate_3d_tilted()` (in module `odtbrain`), [14](#)

F

`fourier_map_2d()` (in module `odtbrain`), [9](#)

I

`integrate_2d()` (in module `odtbrain`), [10](#)

O

`odt_to_ri()` (in module `odtbrain`), [6](#)
`opt_to_ri()` (in module `odtbrain`), [7](#)

S

`sinogram_as_radon()` (in module `odtbrain`), [5](#)
`sinogram_as_rytov()` (in module `odtbrain`), [6](#)