odtbrain Documentation

Release 0.4.3

Paul Müller

Jun 25, 2021

CONTENTS

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 0.4.3.

CHAPTER

ONE

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 $\widehat{U}_{\mathrm{B},\phi_0}(\mathbf{k}_\mathrm{D})$ of the scattered field $u_\mathrm{B}(\mathbf{r_D})$, 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 $\widehat{F}(\mathbf{k})$ of the object function $f(\mathbf{r})$ [KS01], [Wol69].

$$\widehat{F}(k_\mathrm{m}(\mathbf{s}-\mathbf{s_0})) = -\sqrt{\frac{2}{\pi}}\frac{ik_\mathrm{m}}{a_0} M \widehat{U}_{\mathrm{B},\phi_0}(\mathbf{k}_\mathrm{D}) \exp(-ik_\mathrm{m} M l_\mathrm{D})$$

In this notation, k_m is the wave number, $\mathbf{s_0}$ is the norm vector pointing at ϕ_0, $M = \sqrt{1-s_\mathrm{x}^2}$ (2D) and $M = \sqrt{1-s_\mathrm{x}^2-s_\mathrm{y}^2}$ (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].

CHAPTER

TWO

CODE REFERENCE

2.1 Data conversion methods

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

$$u_\mathrm{B}(\mathbf{r}) = u_0(\mathbf{r}) \ln\left(\frac{u_\mathrm{R}(\mathbf{r})}{u_0(\mathbf{r})} + 1\right)$$

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

Parameters

- **uSin** (*2d or 3d complex ndarray*) – The sinogram of the complex wave $u_\mathrm{R}(\mathbf{r}) + u_0(\mathbf{r})$. 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* $= \frac{u_\mathrm{R}(\mathbf{r})}{u_0(\mathbf{r})} + 1$). 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 $u_\mathrm{B}(\mathbf{r})$.

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

$$f(\mathbf{r}) = k_\mathrm{m}^2 \left[\left(\frac{n(\mathbf{r})}{n_\mathrm{m}} \right)^2 - 1 \right]$$

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

$$n(\mathbf{r}) = n_\mathrm{m} \sqrt{\frac{f(\mathbf{r})}{k_\mathrm{m}^2} + 1}$$

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.

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

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 $u_0(\mathbf{r}) = a_0 \exp(-ik_\mathrm{m}\mathbf{s_0}\mathbf{r})$ reads:

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

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

$$G(\mathbf{r}-\mathbf{r}') = \frac{i}{4} H_0^{(1)}(k_\mathrm{m}\,|\mathbf{r}-\mathbf{r}'|)$$

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

$$\widehat{F}(k_\mathrm{m}(\mathbf{s}-\mathbf{s_0})) = -\sqrt{\frac{2}{\pi}} \frac{ik_\mathrm{m}}{a_0} M \widehat{U}_{\mathrm{B},\phi_0}(k_\mathrm{Dx}) \exp(-ik_\mathrm{m} M l_\mathrm{D})$$

where $\widehat{F}(k_\mathrm{x}, k_\mathrm{z})$ is the Fourier transformed object function and $\widehat{U}_{\mathrm{B},\phi_0}(k_\mathrm{Dx})$ 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:

$$\begin{aligned} k_\mathrm{m}(\mathbf{s}-\mathbf{s_0}) &= k_\mathrm{Dx}\,\mathbf{t}_\perp + k_\mathrm{m}(M-1)\,\mathbf{s_0} \\ \mathbf{s_0} &= (p_0,\, M_0) = (-\sin\phi_0,\, \cos\phi_0) \\ \mathbf{t}_\perp &= (-M_0,\, p_0) = (\cos\phi_0,\, \sin\phi_0) \end{aligned}$$

2.2.1 Method summary

backpropagate_2d(uSin, angles, res, nm[, …])	2D backpropagation with the Fourier diffraction theorem
fourier_map_2d(uSin, angles, res, nm[, lD, …])	2D Fourier mapping with the Fourier diffraction theorem
integrate_2d(uSin, angles, res, nm[, lD, …])	(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].

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

with the forward $\mathrm{FFT}_{1\mathrm{D}}$ and inverse $\mathrm{FFT}_{1\mathrm{D}}^{-1}$ 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 $|k_\mathrm{Dx}|$, and the propagation distance $(z_{\phi_j} - l_\mathrm{D})$.

Parameters

- **uSin** (*(A,N) ndarray*) – Two-dimensional sinogram of line recordings $u_{\mathrm{B},\phi_j}(x_\mathrm{D})$ divided by the incident plane wave $u_0(l_\mathrm{D})$ 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.

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

  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 *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`).

2.2.3 Fourier mapping

odtbrain.**fourier_map_2d**(*uSin*, *angles*, *res*, *nm*, *lD=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 $u_{\mathrm{B},\phi_j}(x_\mathrm{D})$ divided by the incident plane wave $u_0(l_\mathrm{D})$ 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.
- **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 *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`).

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*, *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 $u_{\mathrm{B},\phi_j}(x_\mathrm{D})$ divided by the incident plane wave $u_0(l_\mathrm{D})$ 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.

- **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 *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 3D inversion

The first Born approximation for a 3D scattering problem with a plane wave $u_0(\mathbf{r}) = a_0 \exp(-ik_\mathrm{m}\mathbf{s_0}\mathbf{r})$ reads:

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

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

$$G(\mathbf{r}-\mathbf{r}') = \frac{ik_\mathrm{m}}{8\pi^2} \iint dpdq \frac{1}{M} \exp\{ik_\mathrm{m}\left[p(x-x') + q(y-y') + M(z-z')\right]\}$$

with

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

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

$$\widehat{F}(k_\mathrm{m}(\mathbf{s}-\mathbf{s_0})) = -\sqrt{\frac{2}{\pi}} \frac{ik_\mathrm{m}}{a_0} M \widehat{U}_{\mathrm{B},\phi_0}(k_\mathrm{Dx}, k_\mathrm{Dy}) \exp(-ik_\mathrm{m} M l_\mathrm{D})$$

where $\widehat{F}(k_\mathrm{x}, k_\mathrm{y}, k_\mathrm{z})$ is the Fourier transformed object function and $\widehat{U}_{\mathrm{B},\phi_0}(k_\mathrm{Dx}, k_\mathrm{Dy})$ 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:

$$
\begin{aligned}
k_\mathrm{m}(\mathbf{s} - \mathbf{s_0}) &= k_\mathrm{Dx}\,\mathbf{t}_\perp + k_\mathrm{m}(M-1)\,\mathbf{s_0} \\
\mathbf{s} &= (p, q, M) \\
\mathbf{s_0} &= (p_0, q_0, M_0) = (-\sin\phi_0,\, 0,\, \cos\phi_0) \\
\mathbf{t}_\perp &= \left(\cos\phi_0,\, \frac{k_\mathrm{Dy}}{k_\mathrm{Dx}},\, \sin\phi_0\right)^\top
\end{aligned}
$$

2.3.1 Method summary

backpropagate_3d(uSin, angles, res, nm[, ...])	3D backpropagation
backpropagate_3d_tilted(uSin, angles, res, nm)	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].

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

with the forward FFT_2D and inverse $\mathrm{FFT}_\mathrm{2D}^{-1}$ 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 $|k_\mathrm{Dx}|$, and the propagation distance $(z_{\phi_j} - l_\mathrm{D})$.

Parameters

- **uSin** (*(A, Ny, Nx) ndarray*) – Three-dimensional sinogram of plane recordings $u_{\mathrm{B},\phi_j}(x_\mathrm{D}, y_\mathrm{D})$ divided by the incident plane wave $u_0(l_\mathrm{D})$ 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.

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

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 geater 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 approximat 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.
- **intp_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.
- **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].

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

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

New in version 0.1.2.

Parameters

- **uSin** (*(A, Ny, Nx) ndarray*) – Three-dimensional sinogram of plane recordings $u_{\mathrm{B},\phi_j}(x_\mathrm{D},y_\mathrm{D})$ divided by the incident plane wave $u_0(l_\mathrm{D})$ 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.

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

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.
- **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.
- **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. Nx!=Ny 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 *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`).

2.4 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

CHAPTER

THREE

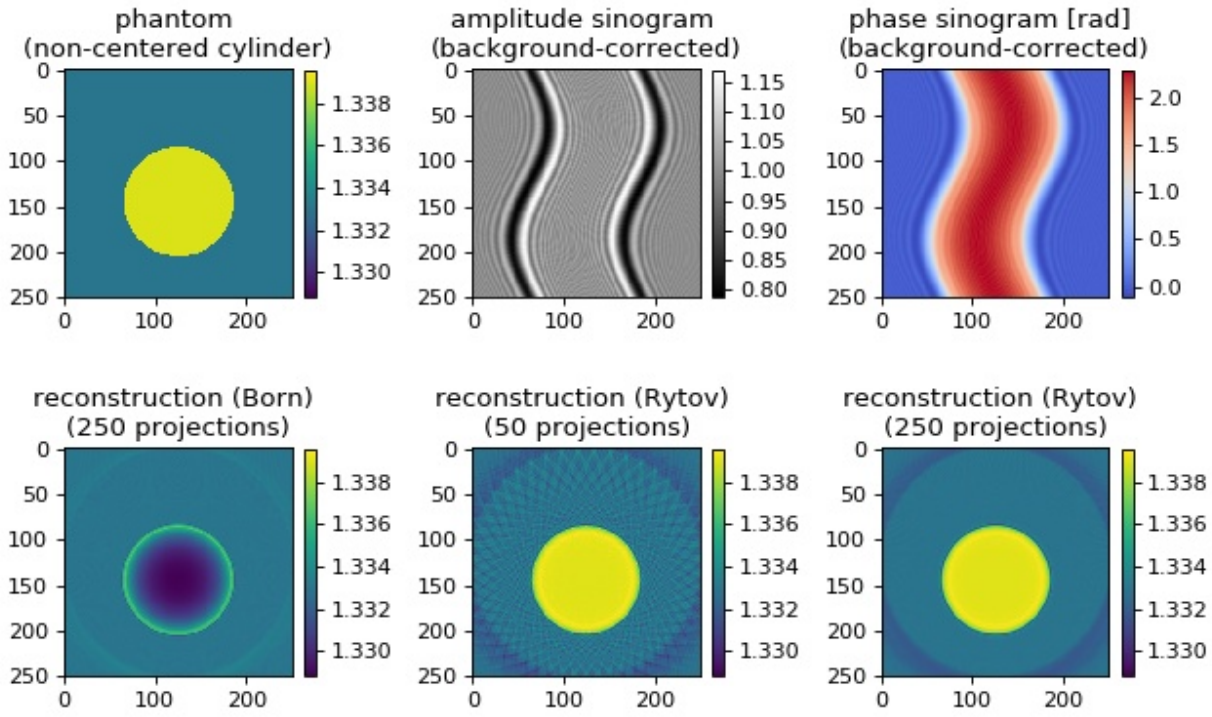
EXAMPLES

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 softare 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`

```
import matplotlib.pylab as plt
import numpy as np

import odtbrain as odt

from example_helper import load_data


# simulation data
sino, angles, cfg = load_data("mie_2d_noncentered_cylinder_A250_R2.zip",
                              f_sino_imag="sino_imag.txt",
                              f_sino_real="sino_real.txt",
                              f_angles="mie_angles.txt",
                              f_info="mie_info.txt")
A, size = sino.shape

# background sinogram computed with Mie theory
# miefield.GetSinogramCylinderRotation(radius, nmed, nmed, lD, lC, size, A,res)
u0 = load_data("mie_2d_noncentered_cylinder_A250_R2.zip",
               f_sino_imag="u0_imag.txt",
               f_sino_real="u0_real.txt")
# create 2d array
u0 = np.tile(u0, size).reshape(A, size).transpose()

# background field necessary to compute initial born field
# u0_single = mie.GetFieldCylinder(radius, nmed, nmed, lD, size, res)
u0_single = load_data("mie_2d_noncentered_cylinder_A250_R2.zip",
                      f_sino_imag="u0_single_imag.txt",
                      f_sino_real="u0_single_real.txt")

print("Example: Backpropagation from 2D Mie simulations")
print("Refractive index of medium:", cfg["nmed"])
print("Measurement position from object center:", cfg["lD"])
print("Wavelength sampling:", cfg["res"])
print("Performing backpropagation.")

# Set measurement parameters
# Compute scattered field from cylinder
radius = cfg["radius"]  # wavelengths
nmed = cfg["nmed"]
ncyl = cfg["ncyl"]

lD = cfg["lD"]  # measurement distance in wavelengths
lC = cfg["lC"]  # displacement from center of image
size = cfg["size"]
res = cfg["res"]  # px/wavelengths
A = cfg["A"]  # number of projections

x = np.arange(size) - size / 2
X, Y = np.meshgrid(x, x)
rad_px = radius * res
phantom = np.array(((Y - lC * res)**2 + X**2) < rad_px**2,
                   dtype=np.float) * (ncyl - nmed) + nmed
```

(continues on next page)

(continued from previous page)

```
# Born
u_sinB = (sino / u0 * u0_single - u0_single)  # fake born
fB = odt.backpropagate_2d(u_sinB, angles, res, nmed, lD * res)
nB = odt.odt_to_ri(fB, res, nmed)

# Rytov
u_sinR = odt.sinogram_as_rytov(sino / u0)
fR = odt.backpropagate_2d(u_sinR, angles, res, nmed, lD * res)
nR = odt.odt_to_ri(fR, res, nmed)

# Rytov 50
u_sinR50 = odt.sinogram_as_rytov((sino / u0)[::5, :])
fR50 = odt.backpropagate_2d(u_sinR50, angles[::5], res, nmed, lD * res)
nR50 = odt.odt_to_ri(fR50, res, nmed)

# Plot sinogram phase and amplitude
ph = odt.sinogram_as_radon(sino / u0)

am = np.abs(sino / u0)

# prepare plot
vmin = np.min(np.array([phantom, nB.real, nR50.real, nR.real]))
vmax = np.max(np.array([phantom, nB.real, nR50.real, nR.real]))

fig, axes = plt.subplots(2, 3, figsize=(8, 5))
axes = np.array(axes).flatten()

phantommap = axes[0].imshow(phantom, vmin=vmin, vmax=vmax)
axes[0].set_title("phantom \n(non-centered cylinder)")

amplmap = axes[1].imshow(am, cmap="gray")
axes[1].set_title("amplitude sinogram \n(background-corrected)")

phasemap = axes[2].imshow(ph, cmap="coolwarm")
axes[2].set_title("phase sinogram [rad] \n(background-corrected)")

axes[3].imshow(nB.real, vmin=vmin, vmax=vmax)
axes[3].set_title("reconstruction (Born) \n(250 projections)")

axes[4].imshow(nR50.real, vmin=vmin, vmax=vmax)
axes[4].set_title("reconstruction (Rytov) \n(50 projections)")

axes[5].imshow(nR.real, vmin=vmin, vmax=vmax)
axes[5].set_title("reconstruction (Rytov) \n(250 projections)")

# color bars
cbkwargs = {"fraction": 0.045}
plt.colorbar(phantommap, ax=axes[0], **cbkwargs)
plt.colorbar(amplmap, ax=axes[1], **cbkwargs)
plt.colorbar(phasemap, ax=axes[2], **cbkwargs)
plt.colorbar(phantommap, ax=axes[3], **cbkwargs)
```

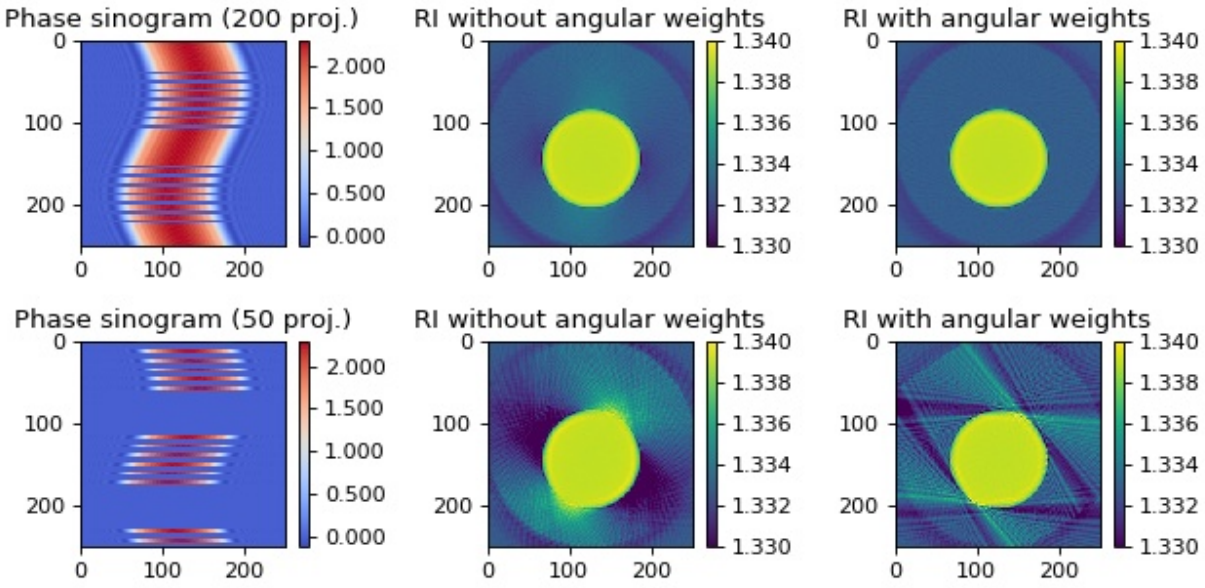
(continues on next page)

(continued from previous page)

```
plt.colorbar(phantommap, ax=axes[4], **cbkwargs)
plt.colorbar(phantommap, ax=axes[5], **cbkwargs)

plt.tight_layout()
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 softare 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

```
import matplotlib.pylab as plt
import numpy as np
import unwrap

import odtbrain as odt

from example_helper import load_data


sino, angles, cfg = load_data("mie_2d_noncentered_cylinder_A250_R2.zip",
                              f_angles="mie_angles.txt",
                              f_sino_real="sino_real.txt",
                              f_sino_imag="sino_imag.txt",
```

(continues on next page)

(continued from previous page)

```
                                f_info="mie_info.txt")
A, size = sino.shape

# background sinogram computed with Mie theory
# miefield.GetSinogramCylinderRotation(radius, nmed, nmed, lD, lC, size, A,res)
u0 = load_data("mie_2d_noncentered_cylinder_A250_R2.zip",
               f_sino_imag="u0_imag.txt",
               f_sino_real="u0_real.txt")
# create 2d array
u0 = np.tile(u0, size).reshape(A, size).transpose()

# background field necessary to compute initial born field
# u0_single = mie.GetFieldCylinder(radius, nmed, nmed, lD, size, res)
u0_single = load_data("mie_2d_noncentered_cylinder_A250_R2.zip",
                      f_sino_imag="u0_single_imag.txt",
                      f_sino_real="u0_single_real.txt")


print("Example: Backpropagation from 2D FDTD simulations")
print("Refractive index of medium:", cfg["nmed"])
print("Measurement position from object center:", cfg["lD"])
print("Wavelength sampling:", cfg["res"])
print("Performing backpropagation.")

# Set measurement parameters
# Compute scattered field from cylinder
radius = cfg["radius"]  # wavelengths
nmed = cfg["nmed"]
ncyl = cfg["ncyl"]

lD = cfg["lD"]  # measurement distance in wavelengths
lC = cfg["lC"]  # displacement from center of image
size = cfg["size"]
res = cfg["res"]  # px/wavelengths
A = cfg["A"]  # number of projections

x = np.arange(size) - size / 2.0
X, Y = np.meshgrid(x, x)
rad_px = radius * res
phantom = np.array(((Y - lC * res)**2 + X**2) < rad_px **
                   2, dtype=np.float) * (ncyl - nmed) + nmed

u_sinR = odt.sinogram_as_rytov(sino / u0)

# Rytov 200 projections
# remove 50 projections from total of 250 projections
remove200 = np.argsort(angles % .0002)[:50]
angles200 = np.delete(angles, remove200, axis=0)
u_sinR200 = np.delete(u_sinR, remove200, axis=0)
ph200 = unwrap.unwrap(np.angle(sino / u0))
ph200[remove200] = 0

```

(continues on next page)

(continued from previous page)

```
fR200 = odt.backpropagate_2d(u_sinR200, angles200, res, nmed, lD*res)
nR200 = odt.odt_to_ri(fR200, res, nmed)
fR200nw = odt.backpropagate_2d(u_sinR200, angles200, res, nmed, lD*res,
                               weight_angles=False)
nR200nw = odt.odt_to_ri(fR200nw, res, nmed)

# Rytov 50 projections
remove50 = np.argsort(angles % .0002)[:200]
angles50 = np.delete(angles, remove50, axis=0)
u_sinR50 = np.delete(u_sinR, remove50, axis=0)
ph50 = unwrap.unwrap(np.angle(sino / u0))
ph50[remove50] = 0

fR50 = odt.backpropagate_2d(u_sinR50, angles50, res, nmed, lD*res)
nR50 = odt.odt_to_ri(fR50, res, nmed)
fR50nw = odt.backpropagate_2d(u_sinR50, angles50, res, nmed, lD*res,
                              weight_angles=False)
nR50nw = odt.odt_to_ri(fR50nw, res, nmed)

# prepare plot
kw_ri = {"vmin": 1.330,
         "vmax": 1.340}

kw_ph = {"vmin": np.min(np.array([ph200, ph50])),
         "vmax": np.max(np.array([ph200, ph50])),
         "cmap": "coolwarm"}

fig, axes = plt.subplots(2, 3, figsize=(8, 4))
axes = np.array(axes).flatten()

phmap = axes[0].imshow(ph200, **kw_ph)
axes[0].set_title("Phase sinogram (200 proj.)")

rimap = axes[1].imshow(nR200nw.real, **kw_ri)
axes[1].set_title("RI without angular weights")

axes[2].imshow(nR200.real, **kw_ri)
axes[2].set_title("RI with angular weights")

axes[3].imshow(ph50, **kw_ph)
axes[3].set_title("Phase sinogram (50 proj.)")

axes[4].imshow(nR50nw.real, **kw_ri)
axes[4].set_title("RI without angular weights")

axes[5].imshow(nR50.real, **kw_ri)
axes[5].set_title("RI with angular weights")

# color bars
cbkwargs = {"fraction": 0.045,
            "format": "%.3f"}
plt.colorbar(phmap, ax=axes[0], **cbkwargs)
```

(continues on next page)

(continued from previous page)

```
plt.colorbar(phmap, ax=axes[3], **cbkwargs)
plt.colorbar(rimap, ax=axes[1], **cbkwargs)
plt.colorbar(rimap, ax=axes[2], **cbkwargs)
plt.colorbar(rimap, ax=axes[5], **cbkwargs)
plt.colorbar(rimap, ax=axes[4], **cbkwargs)

plt.tight_layout()
plt.show()
```

3.1.3 Mie cylinder with incomplete angular coverage

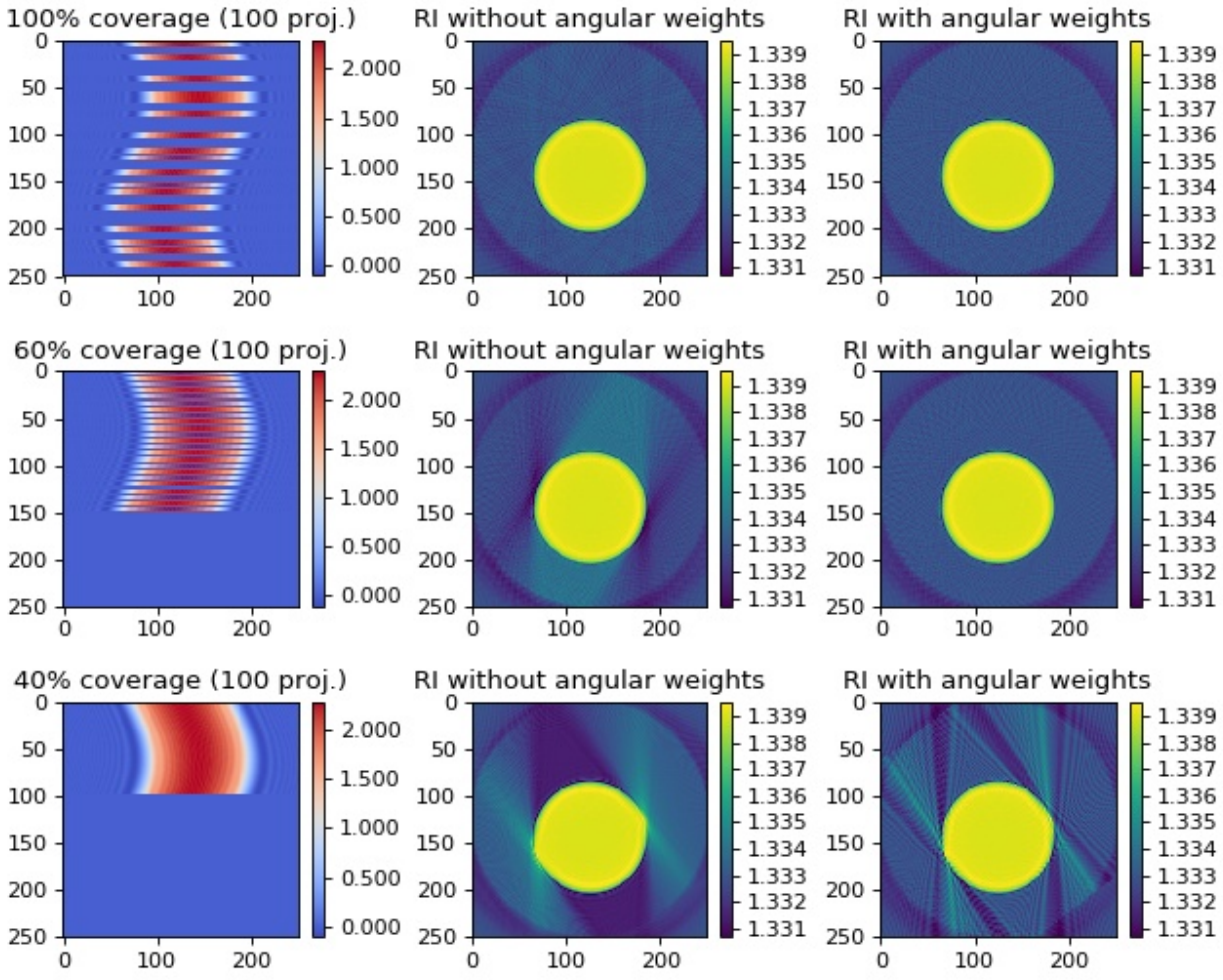
This example illustrates how the backpropagation algorithm of ODTbrain handles incomplete angular coverage. All examples use 100 projections at 100%, 60%, and 40% total angular coverage. The keyword argument *weight_angles* that invokes angular weighting is set to *True* by default. The *in silico* data set was created with the softare 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

```
import matplotlib.pylab as plt
import numpy as np

import odtbrain as odt

from example_helper import load_data

sino, angles, cfg = load_data("mie_2d_noncentered_cylinder_A250_R2.zip",
                              f_angles="mie_angles.txt",
                              f_sino_real="sino_real.txt",
                              f_sino_imag="sino_imag.txt",
                              f_info="mie_info.txt")
A, size = sino.shape

# background sinogram computed with Mie theory
# miefield.GetSinogramCylinderRotation(radius, nmed, nmed, lD, lC, size, A,res)
u0 = load_data("mie_2d_noncentered_cylinder_A250_R2.zip",
               f_sino_imag="u0_imag.txt",
               f_sino_real="u0_real.txt")
# create 2d array
u0 = np.tile(u0, size).reshape(A, size).transpose()

# background field necessary to compute initial born field
# u0_single = mie.GetFieldCylinder(radius, nmed, nmed, lD, size, res)
u0_single = load_data("mie_2d_noncentered_cylinder_A250_R2.zip",
                      f_sino_imag="u0_single_imag.txt",
                      f_sino_real="u0_single_real.txt")
```

(continues on next page)


100% coverage (100 proj.)
RI without angular weights
RI with angular weights
60% coverage (100 proj.)
RI without angular weights
RI with angular weights
40% coverage (100 proj.)
RI without angular weights
RI with angular weights
0
50
100
150
200
250
0
100
200
2.000
1.500
1.000
0.500
0.000
1.339
1.338
1.337
1.336
1.335
1.334
1.333
1.332
1.331

(continued from previous page)

```
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=np.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(
79      u_sinRmiss, anglesmiss, res, nmed, lD * res, weight_angles=False)
```

(continues on next page)

(continued from previous page)

```
nRmissnw = odt.odt_to_ri(fRmissnw, res, nmed)

# Rytov 100 projections less than 180
removebad = 249 - np.arange(150)
anglesbad = np.delete(angles, removebad, axis=0)
u_sinRbad = np.delete(u_sinR, removebad, axis=0)
phbad = odt.sinogram_as_radon(sino / u0)
phbad[removebad] = 0

fRbad = odt.backpropagate_2d(u_sinRbad, anglesbad, res, nmed, lD * res)
nRbad = odt.odt_to_ri(fRbad, res, nmed)
fRbadnw = odt.backpropagate_2d(
    u_sinRbad, anglesbad, res, nmed, lD * res, weight_angles=False)
nRbadnw = odt.odt_to_ri(fRbadnw, res, nmed)

# prepare plot
kw_ri = {"vmin": np.min(np.array([phantom, nRmiss.real, nReven.real])),
         "vmax": np.max(np.array([phantom, nRmiss.real, nReven.real]))}

kw_ph = {"vmin": np.min(np.array([pheven, phmiss])),
         "vmax": np.max(np.array([pheven, phmiss])),
         "cmap": "coolwarm"}

fig, axes = plt.subplots(3, 3, figsize=(8, 6.5))

axes[0, 0].set_title("100% coverage ({} proj.)".format(angleseven.shape[0]))
phmap = axes[0, 0].imshow(pheven, **kw_ph)

axes[0, 1].set_title("RI without angular weights")
rimap = axes[0, 1].imshow(nRevennw.real, **kw_ri)

axes[0, 2].set_title("RI with angular weights")
rimap = axes[0, 2].imshow(nReven.real, **kw_ri)

axes[1, 0].set_title("60% coverage ({} proj.)".format(anglesmiss.shape[0]))
axes[1, 0].imshow(phmiss, **kw_ph)

axes[1, 1].set_title("RI without angular weights")
axes[1, 1].imshow(nRmissnw.real, **kw_ri)

axes[1, 2].set_title("RI with angular weights")
axes[1, 2].imshow(nRmiss.real, **kw_ri)

axes[2, 0].set_title("40% coverage ({} proj.)".format(anglesbad.shape[0]))
axes[2, 0].imshow(phbad, **kw_ph)

axes[2, 1].set_title("RI without angular weights")
axes[2, 1].imshow(nRbadnw.real, **kw_ri)

axes[2, 2].set_title("RI with angular weights")
axes[2, 2].imshow(nRbad.real, **kw_ri)

```

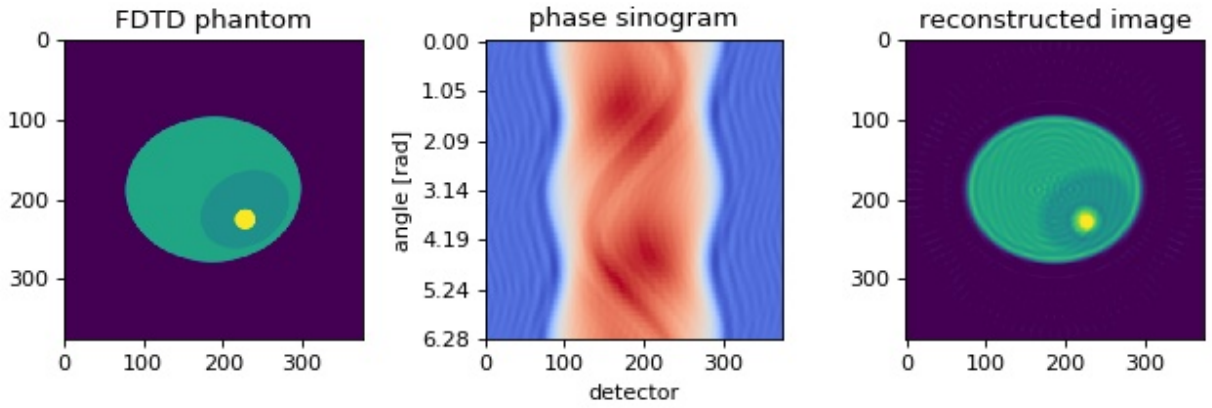
(continues on next page)

(continued from previous page)

```
# color bars
cbkwargs = {"fraction": 0.045,
            "format": "%.3f"}
plt.colorbar(phmap, ax=axes[0, 0], **cbkwargs)
plt.colorbar(phmap, ax=axes[1, 0], **cbkwargs)
plt.colorbar(phmap, ax=axes[2, 0], **cbkwargs)
plt.colorbar(rimap, ax=axes[0, 1], **cbkwargs)
plt.colorbar(rimap, ax=axes[1, 1], **cbkwargs)
plt.colorbar(rimap, ax=axes[2, 1], **cbkwargs)
plt.colorbar(rimap, ax=axes[0, 2], **cbkwargs)
plt.colorbar(rimap, ax=axes[1, 2], **cbkwargs)
plt.colorbar(rimap, ax=axes[2, 2], **cbkwargs)

plt.tight_layout()
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

```
import matplotlib.pylab as plt
import numpy as np
import odtbrain as odt

from example_helper import load_data


sino, angles, phantom, cfg = load_data("fdtd_2d_sino_A100_R13.zip",
                                       f_angles="fdtd_angles.txt",
                                       f_sino_imag="fdtd_imag.txt",
                                       f_sino_real="fdtd_real.txt",
                                       f_info="fdtd_info.txt",
```

(continues on next page)

(continued from previous page)

```
                                    f_phantom="fdtd_phantom.txt",
                                    )

print("Example: Backpropagation from 2D FDTD simulations")
print("Refractive index of medium:", cfg["nm"])
print("Measurement position from object center:", cfg["lD"])
print("Wavelength sampling:", cfg["res"])
print("Performing backpropagation.")

# Apply the Rytov approximation
sino_rytov = odt.sinogram_as_rytov(sino)

# perform backpropagation to obtain object function f
f = odt.backpropagate_2d(uSin=sino_rytov,
                         angles=angles,
                         res=cfg["res"],
                         nm=cfg["nm"],
                         lD=cfg["lD"] * cfg["res"]
                         )

# compute refractive index n from object function
n = odt.odt_to_ri(f, res=cfg["res"], nm=cfg["nm"])

# compare phantom and reconstruction in plot
fig, axes = plt.subplots(1, 3, figsize=(8, 2.8))

axes[0].set_title("FDTD phantom")
axes[0].imshow(phantom, vmin=phantom.min(), vmax=phantom.max())
sino_phase = np.unwrap(np.angle(sino), axis=1)

axes[1].set_title("phase sinogram")
axes[1].imshow(sino_phase, vmin=sino_phase.min(), vmax=sino_phase.max(),
               aspect=sino.shape[1] / sino.shape[0],
               cmap="coolwarm")
axes[1].set_xlabel("detector")
axes[1].set_ylabel("angle [rad]")

axes[2].set_title("reconstructed image")
axes[2].imshow(n.real, vmin=phantom.min(), vmax=phantom.max())

# set y ticks for sinogram
labels = np.linspace(0, 2 * np.pi, len(axes[1].get_yticks()))
labels = ["{:.2f}".format(i) for i in labels]
axes[1].set_yticks(np.linspace(0, len(angles), len(labels)))
axes[1].set_yticklabels(labels)

plt.tight_layout()
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.

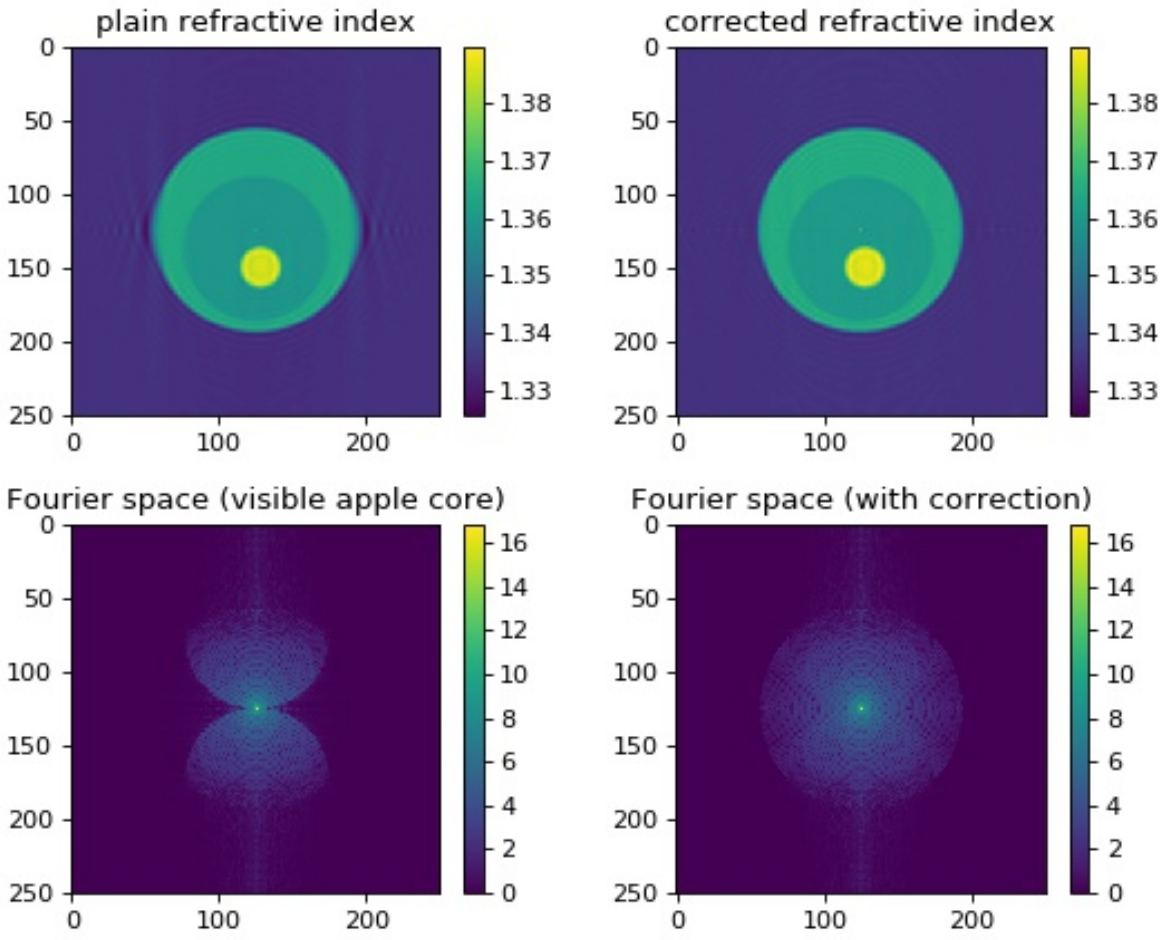
This example generates an artificial sinogram using the Python library cellsino (The example parameters are reused from this example). The sinogram is then reconstructed with the backpropagation algorithm and the missing apple core correction is applied.

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

```
import matplotlib.pylab as plt
import numpy as np

import cellsino
import odtbrain as odt


if __name__ == "__main__":
    # number of sinogram angles
    num_ang = 160
    # sinogram acquisition angles
    angles = np.linspace(0, 2*np.pi, num_ang, endpoint=False)
    # detector grid size
    grid_size = (250, 250)
    # vacuum wavelength [m]
    wavelength = 550e-9
    # pixel size [m]
    pixel_size = 0.08e-6
    # refractive index of the surrounding medium
    medium_index = 1.335

    # initialize cell phantom
    phantom = cellsino.phantoms.SimpleCell()

    # initialize sinogram with geometric parameters
    sino = cellsino.Sinogram(phantom=phantom,
```

(continues on next page)


plain refractive index
corrected refractive index
Fourier space (visible apple core)
Fourier space (with correction)
0
50
100
150
200
250
1.38
1.37
1.36
1.35
1.34
1.33
16
14
12
10
8
6
4
2

(continued from previous page)

```
                            wavelength=wavelength,
                            pixel_size=pixel_size,
                            grid_size=grid_size)

    # compute sinogram (field with Rytov approximation and fluorescence)
    sino = sino.compute(angles=angles, propagator="rytov", mode="field")

    # reconstruction of refractive index
    sino_rytov = odt.sinogram_as_rytov(sino)
    f = odt.backpropagate_3d(uSin=sino_rytov,
                            angles=angles,
                            res=wavelength/pixel_size,
                            nm=medium_index)

    ri = odt.odt_to_ri(f=f,
                       res=wavelength/pixel_size,
                       nm=medium_index)

    # apple core correction
    fc = odt.apple.correct(f=f,
                           res=wavelength/pixel_size,
                           nm=medium_index,
                           method="sh")

    ric = odt.odt_to_ri(f=fc,
                        res=wavelength/pixel_size,
                        nm=medium_index)

    # plotting
    idx = ri.shape[2] // 2

    # log-scaled power spectra
    ft = np.log(1 + np.abs(np.fft.fftshift(np.fft.fftn(ri))))
    ftc = np.log(1 + np.abs(np.fft.fftshift(np.fft.fftn(ric))))

    plt.figure(figsize=(7, 5.5))

    plotkwri = {"vmax": ri.real.max(),
                "vmin": ri.real.min(),
                "interpolation": "none",
                }

    plotkwft = {"vmax": ft.max(),
                "vmin": 0,
                "interpolation": "none",
                }

    ax1 = plt.subplot(221, title="plain refractive index")
    mapper = ax1.imshow(ri[:, :, idx].real, **plotkwri)
    plt.colorbar(mappable=mapper, ax=ax1)

    ax2 = plt.subplot(222, title="corrected refractive index")
```

(continues on next page)

(continued from previous page)

```
    mapper = ax2.imshow(ric[:, :, idx].real, **plotkwri)
    plt.colorbar(mappable=mapper, ax=ax2)

    ax3 = plt.subplot(223, title="Fourier space (visible apple core)")
    mapper = ax3.imshow(ft[:, :, idx], **plotkwft)
    plt.colorbar(mappable=mapper, ax=ax3)

    ax4 = plt.subplot(224, title="Fourier space (with correction)")
    mapper = ax4.imshow(ftc[:, :, idx], **plotkwft)
    plt.colorbar(mappable=mapper, ax=ax4)

    plt.tight_layout()
    plt.show()
```

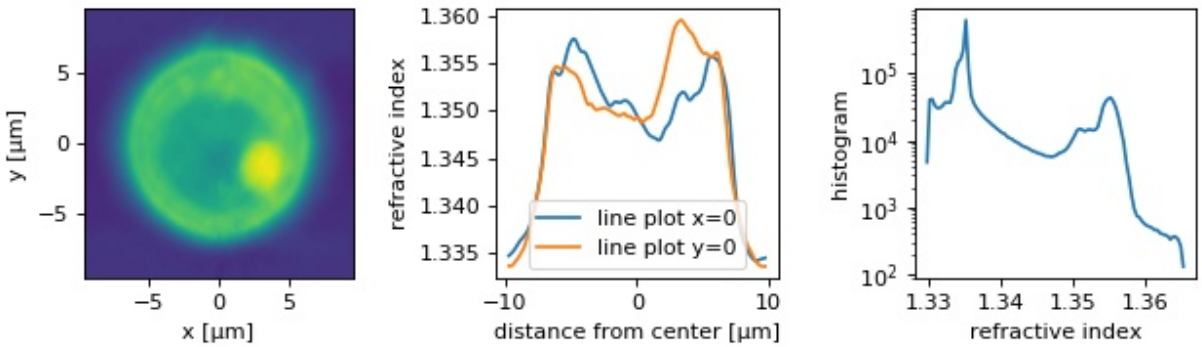
3.2.2 HL60 cell

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

```
import pathlib
import tarfile
import tempfile

import matplotlib.pylab as plt
import numpy as np
```

(continues on next page)

(continued from previous page)

```
import odtbrain as odt
import qpimage

from example_helper import get_file, extract_lzma


if __name__ == "__main__":
    # ascertain the data
    path = get_file("qlsi_3d_hl60-cell_A140.tar.lzma")
    tarf = extract_lzma(path)
    tdir = tempfile.mkdtemp(prefix="odtbrain_example_")

    with tarfile.open(tarf) as tf:
        tf.extract("series.h5", path=tdir)
        angles = np.loadtxt(tf.extractfile("angles.txt"))

    # extract the complex field sinogram from the qpimage series data
    h5file = pathlib.Path(tdir) / "series.h5"
    with qpimage.QPSeries(h5file=h5file, h5mode="r") as qps:
        qp0 = qps[0]
        meta = qp0.meta
        sino = np.zeros((len(qps), qp0.shape[0], qp0.shape[1]),
                        dtype=np.complex)
        for ii in range(len(qps)):
            sino[ii] = qps[ii].field

    # perform backgpropagation
    u_sinR = odt.sinogram_as_rytov(sino)
    res = meta["wavelength"] / meta["pixel size"]
    nm = meta["medium index"]

    fR = odt.backpropagate_3d(uSin=u_sinR,
                              angles=angles,
                              res=res,
                              nm=nm)

    ri = odt.odt_to_ri(fR, res, nm)

    # plot results
    ext = meta["pixel size"] * 1e6 * 70
    kw = {"vmin": ri.real.min(),
          "vmax": ri.real.max(),
          "extent": [-ext, ext, -ext, ext]}
    fig, axes = plt.subplots(1, 3, figsize=(8, 2.5))
    axes[0].imshow(ri[70, :, :].real, **kw)
    axes[0].set_xlabel("x [µm]")
    axes[0].set_ylabel("y [µm]")

    x = np.linspace(-ext, ext, 140)
    axes[1].plot(x, ri[70, :, 70], label="line plot x=0")
    axes[1].plot(x, ri[70, 70, :], label="line plot y=0")
    axes[1].set_xlabel("distance from center [µm]")
```

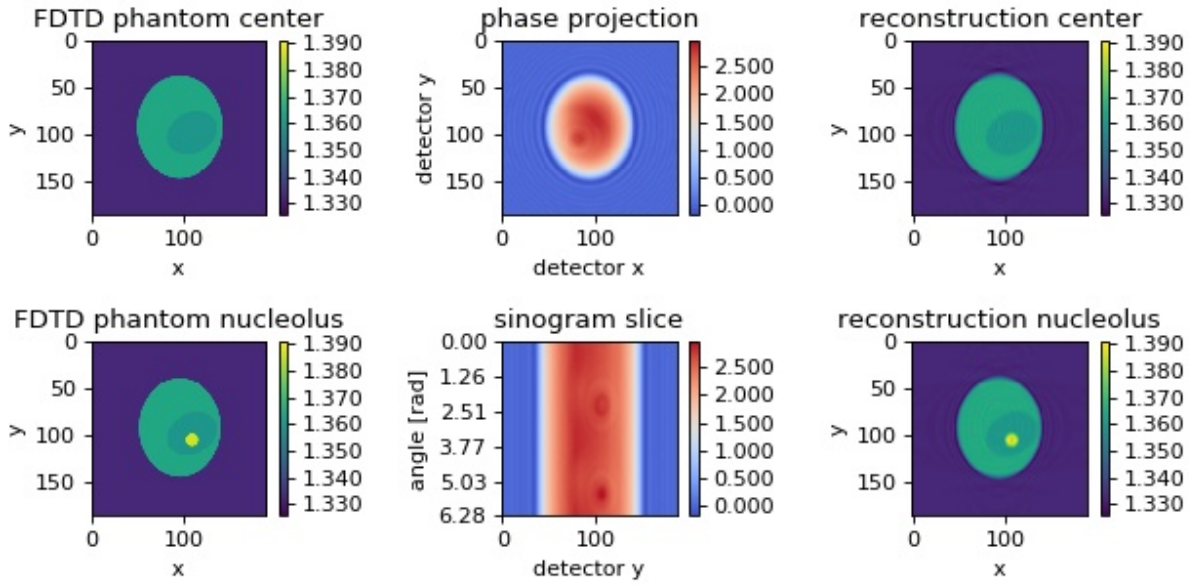
(continues on next page)

(continued from previous page)

```
    axes[1].set_ylabel("refractive index")
    axes[1].legend()

    hist, xh = np.histogram(ri.real, bins=100)
    axes[2].plot(xh[1:], hist)
    axes[2].set_yscale('log')
    axes[2].set_xlabel("refractive index")
    axes[2].set_ylabel("histogram")

    plt.tight_layout()
    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

```
import matplotlib.pylab as plt
import numpy as np

import odtbrain as odt

from example_helper import load_data


```

(continues on next page)

(continued from previous page)

```
if __name__ == "__main__":
    sino, angles, phantom, cfg = \
        load_data("fdtd_3d_sino_A180_R6.500.tar.lzma")

    A = angles.shape[0]

    print("Example: Backpropagation from 3D FDTD simulations")
    print("Refractive index of medium:", cfg["nm"])
    print("Measurement position from object center:", cfg["lD"])
    print("Wavelength sampling:", cfg["res"])
    print("Number of projections:", A)
    print("Performing backpropagation.")

    # Apply the Rytov approximation
    sinoRytov = odt.sinogram_as_rytov(sino)

    # perform backpropagation to obtain object function f
    f = odt.backpropagate_3d(uSin=sinoRytov,
                             angles=angles,
                             res=cfg["res"],
                             nm=cfg["nm"],
                             lD=cfg["lD"]
                             )

    # compute refractive index n from object function
    n = odt.odt_to_ri(f, res=cfg["res"], nm=cfg["nm"])

    sx, sy, sz = n.shape
    px, py, pz = phantom.shape

    sino_phase = np.angle(sino)

    # compare phantom and reconstruction in plot
    fig, axes = plt.subplots(2, 3, figsize=(8, 4))
    kwri = {"vmin": n.real.min(), "vmax": n.real.max()}
    kwph = {"vmin": sino_phase.min(), "vmax": sino_phase.max(),
            "cmap": "coolwarm"}

    # Phantom
    axes[0, 0].set_title("FDTD phantom center")
    rimap = axes[0, 0].imshow(phantom[px // 2], **kwri)
    axes[0, 0].set_xlabel("x")
    axes[0, 0].set_ylabel("y")

    axes[1, 0].set_title("FDTD phantom nucleolus")
    axes[1, 0].imshow(phantom[int(px / 2 + 2 * cfg["res"])], **kwri)
    axes[1, 0].set_xlabel("x")
    axes[1, 0].set_ylabel("y")

    # Sinogram
    axes[0, 1].set_title("phase projection")
    phmap = axes[0, 1].imshow(sino_phase[A // 2, :, :], **kwph)
```

(continues on next page)

(continued from previous page)

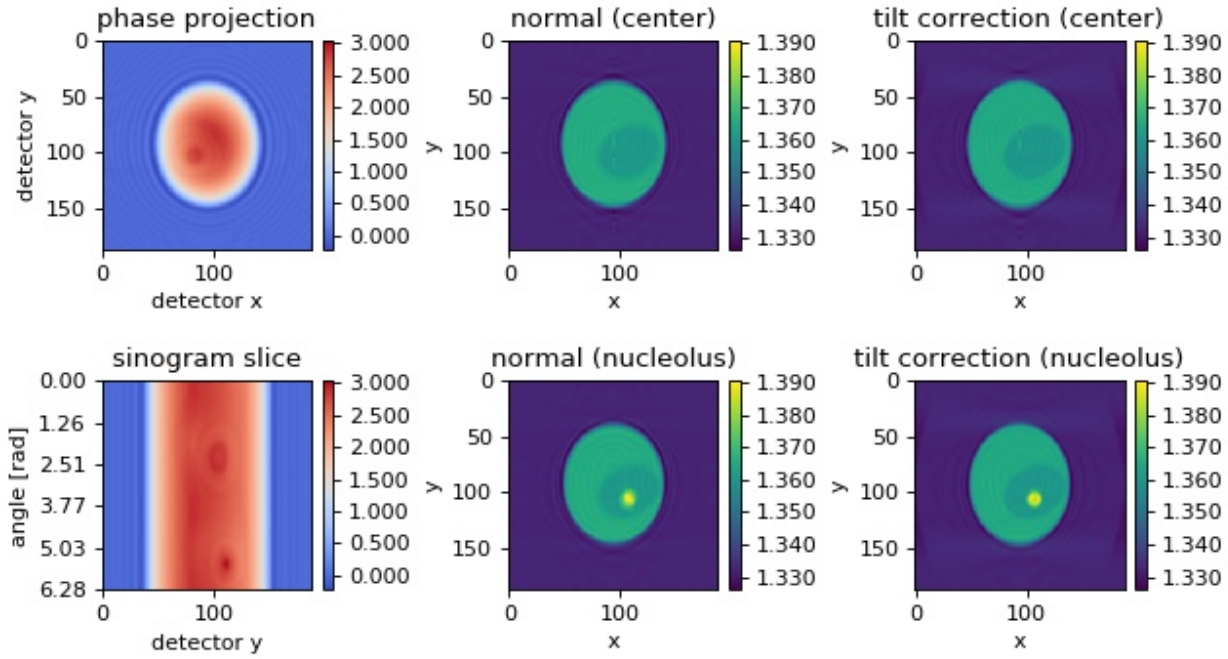
```
    axes[0, 1].set_xlabel("detector x")
    axes[0, 1].set_ylabel("detector y")

    axes[1, 1].set_title("sinogram slice")
    axes[1, 1].imshow(sino_phase[:, :, sino.shape[2] // 2],
                      aspect=sino.shape[1] / sino.shape[0], **kwph)
    axes[1, 1].set_xlabel("detector y")
    axes[1, 1].set_ylabel("angle [rad]")
    # set y ticks for sinogram
    labels = np.linspace(0, 2 * np.pi, len(axes[1, 1].get_yticks()))
    labels = ["{:.2f}".format(i) for i in labels]
    axes[1, 1].set_yticks(np.linspace(0, len(angles), len(labels)))
    axes[1, 1].set_yticklabels(labels)

    axes[0, 2].set_title("reconstruction center")
    axes[0, 2].imshow(n[sx // 2].real, **kwri)
    axes[0, 2].set_xlabel("x")
    axes[0, 2].set_ylabel("y")

    axes[1, 2].set_title("reconstruction nucleolus")
    axes[1, 2].imshow(n[int(sx / 2 + 2 * cfg["res"])].real, **kwri)
    axes[1, 2].set_xlabel("x")
    axes[1, 2].set_ylabel("y")

    # color bars
    cbkwargs = {"fraction": 0.045,
                "format": "%.3f"}
    plt.colorbar(phmap, ax=axes[0, 1], **cbkwargs)
    plt.colorbar(phmap, ax=axes[1, 1], **cbkwargs)
    plt.colorbar(rimap, ax=axes[0, 0], **cbkwargs)
    plt.colorbar(rimap, ax=axes[1, 0], **cbkwargs)
    plt.colorbar(rimap, ax=axes[0, 2], **cbkwargs)
    plt.colorbar(rimap, ax=axes[1, 2], **cbkwargs)

    plt.tight_layout()
    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`




```
import matplotlib.pylab as plt
import numpy as np

import odtbrain as odt

from example_helper import load_data


if __name__ == "__main__":
    sino, angles, phantom, cfg = \
        load_data("fdtd_3d_sino_A220_R6.500_tiltyz0.2.tar.lzma")

    A = angles.shape[0]

    print("Example: Backpropagation from 3D FDTD simulations")
    print("Refractive index of medium:", cfg["nm"])
    print("Measurement position from object center:", cfg["lD"])
    print("Wavelength sampling:", cfg["res"])
    print("Axis tilt in y-z direction:", cfg["tilt_yz"])
    print("Number of projections:", A)

    print("Performing normal backpropagation.")
    # Apply the Rytov approximation
    sinoRytov = odt.sinogram_as_rytov(sino)

    # Perform naive backpropagation
    f_naiv = odt.backpropagate_3d(uSin=sinoRytov,
                                  angles=angles,
                                  res=cfg["res"],
```

(continues on next page)

(continued from previous page)

```
                                    nm=cfg["nm"],
                                    lD=cfg["lD"]
                                    )

    print("Performing tilted backpropagation.")
    # Determine tilted axis
    tilted_axis = [0, np.cos(cfg["tilt_yz"]), np.sin(cfg["tilt_yz"])]

    # Perform tilted backpropagation
    f_tilt = odt.backpropagate_3d_tilted(uSin=sinoRytov,
                                         angles=angles,
                                         res=cfg["res"],
                                         nm=cfg["nm"],
                                         lD=cfg["lD"],
                                         tilted_axis=tilted_axis,
                                         )

    # compute refractive index n from object function
    n_naiv = odt.odt_to_ri(f_naiv, res=cfg["res"], nm=cfg["nm"])
    n_tilt = odt.odt_to_ri(f_tilt, res=cfg["res"], nm=cfg["nm"])

    sx, sy, sz = n_tilt.shape
    px, py, pz = phantom.shape

    sino_phase = np.angle(sino)

    # compare phantom and reconstruction in plot
    fig, axes = plt.subplots(2, 3, figsize=(8, 4.5))
    kwri = {"vmin": n_tilt.real.min(), "vmax": n_tilt.real.max()}
    kwph = {"vmin": sino_phase.min(), "vmax": sino_phase.max(),
            "cmap": "coolwarm"}

    # Sinogram
    axes[0, 0].set_title("phase projection")
    phmap = axes[0, 0].imshow(sino_phase[A // 2, :, :], **kwph)
    axes[0, 0].set_xlabel("detector x")
    axes[0, 0].set_ylabel("detector y")

    axes[1, 0].set_title("sinogram slice")
    axes[1, 0].imshow(sino_phase[:, :, sino.shape[2] // 2],
                      aspect=sino.shape[1] / sino.shape[0], **kwph)
    axes[1, 0].set_xlabel("detector y")
    axes[1, 0].set_ylabel("angle [rad]")
    # set y ticks for sinogram
    labels = np.linspace(0, 2 * np.pi, len(axes[1, 1].get_yticks()))
    labels = ["{:.2f}".format(i) for i in labels]
    axes[1, 0].set_yticks(np.linspace(0, len(angles), len(labels)))
    axes[1, 0].set_yticklabels(labels)

    axes[0, 1].set_title("normal (center)")
    rimap = axes[0, 1].imshow(n_naiv[sx // 2].real, **kwri)
    axes[0, 1].set_xlabel("x")
```

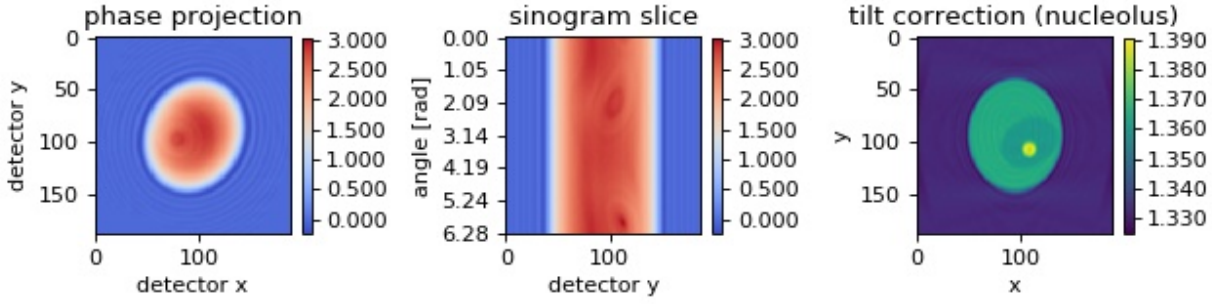
(continues on next page)

(continued from previous page)

```
    axes[0, 1].set_ylabel("y")

    axes[1, 1].set_title("normal (nucleolus)")
    axes[1, 1].imshow(n_naiv[int(sx / 2 + 2 * cfg["res"])].real, **kwri)
    axes[1, 1].set_xlabel("x")
    axes[1, 1].set_ylabel("y")

    axes[0, 2].set_title("tilt correction (center)")
    axes[0, 2].imshow(n_tilt[sx // 2].real, **kwri)
    axes[0, 2].set_xlabel("x")
    axes[0, 2].set_ylabel("y")

    axes[1, 2].set_title("tilt correction (nucleolus)")
    axes[1, 2].imshow(n_tilt[int(sx / 2 + 2 * cfg["res"])].real, **kwri)
    axes[1, 2].set_xlabel("x")
    axes[1, 2].set_ylabel("y")

    # color bars
    cbkwargs = {"fraction": 0.045,
                "format": "%.3f"}
    plt.colorbar(phmap, ax=axes[0, 0], **cbkwargs)
    plt.colorbar(phmap, ax=axes[1, 0], **cbkwargs)
    plt.colorbar(rimap, ax=axes[0, 1], **cbkwargs)
    plt.colorbar(rimap, ax=axes[1, 1], **cbkwargs)
    plt.colorbar(rimap, ax=axes[0, 2], **cbkwargs)
    plt.colorbar(rimap, ax=axes[1, 2], **cbkwargs)

    plt.tight_layout()
    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_fdtd_3d_tilted2.py`

```
import matplotlib.pylab as plt
import numpy as np
from scipy.ndimage import rotate

import odtbrain as odt

from example_helper import load_data


if __name__ == "__main__":
    sino, angles, phantom, cfg = \
        load_data("fdtd_3d_sino_A220_R6.500_tiltyz0.2.tar.lzma")

    # Perform titlt by -.42 rad in detector plane
    rotang = -0.42
    rotkwargs = {"mode": "constant",
                 "order": 2,
                 "reshape": False,
                 }
    for ii in range(len(sino)):
        sino[ii].real = rotate(
            sino[ii].real, np.rad2deg(rotang), cval=1, **rotkwargs)
        sino[ii].imag = rotate(
            sino[ii].imag, np.rad2deg(rotang), cval=0, **rotkwargs)

    A = angles.shape[0]

    print("Example: Backpropagation from 3D FDTD simulations")
    print("Refractive index of medium:", cfg["nm"])
    print("Measurement position from object center:", cfg["lD"])
    print("Wavelength sampling:", cfg["res"])
    print("Axis tilt in y-z direction:", cfg["tilt_yz"])
    print("Number of projections:", A)

    # Apply the Rytov approximation
    sinoRytov = odt.sinogram_as_rytov(sino)

    # Determine tilted axis
    tilted_axis = [0, np.cos(cfg["tilt_yz"]), np.sin(cfg["tilt_yz"])]
    rotmat = np.array([
        [np.cos(rotang), -np.sin(rotang), 0],
        [np.sin(rotang), np.cos(rotang), 0],
        [0, 0, 1],
    ])
    tilted_axis = np.dot(rotmat, tilted_axis)

    print("Performing tilted backpropagation.")
    # Perform tilted backpropagation
    f_tilt = odt.backpropagate_3d_tilted(uSin=sinoRytov,
                                         angles=angles,
                                         res=cfg["res"],
                                         nm=cfg["nm"],
                                         lD=cfg["lD"],
```

(continues on next page)

(continued from previous page)

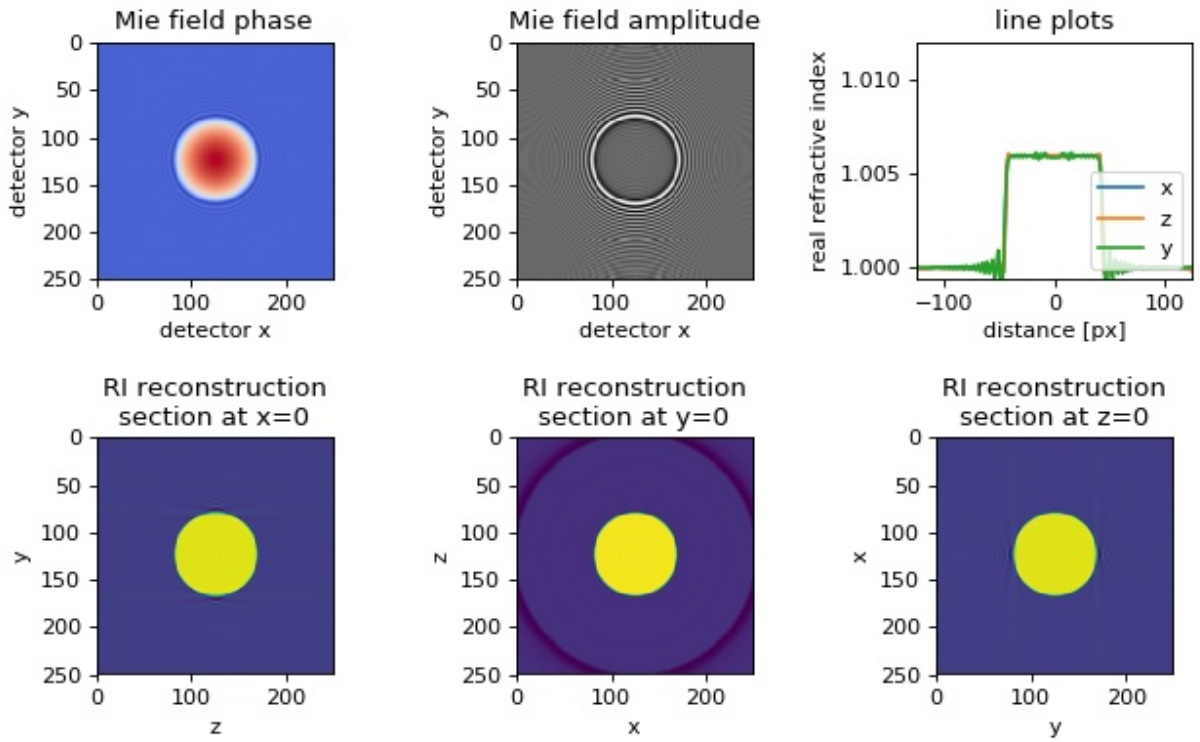
```
                                       tilted_axis=tilted_axis,
                                       )

    # compute refractive index n from object function
    n_tilt = odt.odt_to_ri(f_tilt, res=cfg["res"], nm=cfg["nm"])

    sx, sy, sz = n_tilt.shape
    px, py, pz = phantom.shape

    sino_phase = np.angle(sino)

    # compare phantom and reconstruction in plot
    fig, axes = plt.subplots(1, 3, figsize=(8, 2.4))
    kwri = {"vmin": n_tilt.real.min(), "vmax": n_tilt.real.max()}
    kwph = {"vmin": sino_phase.min(), "vmax": sino_phase.max(),
            "cmap": "coolwarm"}

    # Sinogram
    axes[0].set_title("phase projection")
    phmap = axes[0].imshow(sino_phase[A // 2, :, :], **kwph)
    axes[0].set_xlabel("detector x")
    axes[0].set_ylabel("detector y")

    axes[1].set_title("sinogram slice")
    axes[1].imshow(sino_phase[:, :, sino.shape[2] // 2],
                   aspect=sino.shape[1] / sino.shape[0], **kwph)
    axes[1].set_xlabel("detector y")
    axes[1].set_ylabel("angle [rad]")
    # set y ticks for sinogram
    labels = np.linspace(0, 2 * np.pi, len(axes[1].get_yticks()))
    labels = ["{:.2f}".format(i) for i in labels]
    axes[1].set_yticks(np.linspace(0, len(angles), len(labels)))
    axes[1].set_yticklabels(labels)

    axes[2].set_title("tilt correction (nucleolus)")
    rimap = axes[2].imshow(n_tilt[int(sx / 2 + 2 * cfg["res"])].real, **kwri)
    axes[2].set_xlabel("x")
    axes[2].set_ylabel("y")

    # color bars
    cbkwargs = {"fraction": 0.045,
                "format": "%.3f"}
    plt.colorbar(phmap, ax=axes[0], **cbkwargs)
    plt.colorbar(phmap, ax=axes[1], **cbkwargs)
    plt.colorbar(rimap, ax=axes[2], **cbkwargs)

    plt.tight_layout()
    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 $n_{\mathrm{sph}} = 1.006$, embedded in a medium of refractive index $n_{\mathrm{med}} = 1.0$ onto a detector which is $l_{\mathrm{D}} = 20\lambda$ 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 ($l_{\mathrm{D}} = 0$).

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

```
import matplotlib.pylab as plt
import nrefocus
import numpy as np

import odtbrain as odt

from example_helper import load_data


if __name__ == "__main__":
    Ex, cfg = load_data("mie_3d_sphere_field.zip",
                        f_sino_imag="mie_sphere_imag.txt",
                        f_sino_real="mie_sphere_real.txt",
                        f_info="mie_info.txt")

```

(continues on next page)

(continued from previous page)

```
    # Manually set number of angles:
    A = 200

    print("Example: Backpropagation from 3D Mie scattering")
    print("Refractive index of medium:", cfg["nm"])
    print("Measurement position from object center:", cfg["lD"])
    print("Wavelength sampling:", cfg["res"])
    print("Number of angles for reconstruction:", A)
    print("Performing backpropagation.")

    # Reconstruction angles
    angles = np.linspace(0, 2 * np.pi, A, endpoint=False)

    # Perform focusing
    Ex = nrefocus.refocus(Ex,
                          d=-cfg["lD"]*cfg["res"],
                          nm=cfg["nm"],
                          res=cfg["res"],
                          )

    # Create sinogram
    u_sin = np.tile(Ex.flat, A).reshape(A, int(cfg["size"]), int(cfg["size"]))

    # Apply the Rytov approximation
    u_sinR = odt.sinogram_as_rytov(u_sin)

    # Backpropagation
    fR = odt.backpropagate_3d(uSin=u_sinR,
                              angles=angles,
                              res=cfg["res"],
                              nm=cfg["nm"],
                              lD=0,
                              padfac=2.1,
                              save_memory=True)

    # RI computation
    nR = odt.odt_to_ri(fR, cfg["res"], cfg["nm"])

    # Plotting
    fig, axes = plt.subplots(2, 3, figsize=(8, 5))
    axes = np.array(axes).flatten()
    # field
    axes[0].set_title("Mie field phase")
    axes[0].set_xlabel("detector x")
    axes[0].set_ylabel("detector y")
    axes[0].imshow(np.angle(Ex), cmap="coolwarm")
    axes[1].set_title("Mie field amplitude")
    axes[1].set_xlabel("detector x")
    axes[1].set_ylabel("detector y")
    axes[1].imshow(np.abs(Ex), cmap="gray")

    # line plot
```

(continues on next page)

(continued from previous page)

```
    axes[2].set_title("line plots")
    axes[2].set_xlabel("distance [px]")
    axes[2].set_ylabel("real refractive index")
    center = int(cfg["size"] / 2)
    x = np.arange(cfg["size"]) - center
    axes[2].plot(x, nR[:, center, center].real, label="x")
    axes[2].plot(x, nR[center, center, :].real, label="z")
    axes[2].plot(x, nR[center, :, center].real, label="y")
    axes[2].legend(loc=4)
    axes[2].set_xlim((-center, center))
    dn = abs(cfg["nsph"] - cfg["nm"])
    axes[2].set_ylim((cfg["nm"] - dn / 10, cfg["nsph"] + dn))
    axes[2].ticklabel_format(useOffset=False)

    # cross sections
    axes[3].set_title("RI reconstruction\nsection at x=0")
    axes[3].set_xlabel("z")
    axes[3].set_ylabel("y")
    axes[3].imshow(nR[center, :, :].real)

    axes[4].set_title("RI reconstruction\nsection at y=0")
    axes[4].set_xlabel("x")
    axes[4].set_ylabel("z")
    axes[4].imshow(nR[:, center, :].real)

    axes[5].set_title("RI reconstruction\nsection at z=0")
    axes[5].set_xlabel("y")
    axes[5].set_ylabel("x")
    axes[5].imshow(nR[:, :, center].real)

    plt.tight_layout()
    plt.show()
```

CHAPTER

FOUR

CHANGELOG

List of changes in-between ODTbrain releases.

4.1 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)

4.2 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

4.3 version 0.4.1

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

4.4 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

4.5 version 0.3.0

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

4.6 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

4.7 version 0.2.5

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

4.8 version 0.2.4

- maintenance release

4.9 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.

4.10 version 0.2.2

- docs: minor update and add changelog

4.11 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

4.12 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

4.13 version 0.1.8

- Updated documentation
- Cleaned up examples

4.14 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

4.15 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

4.16 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.

4.17 version 0.1.4

- The exponential term containing the distance between center of rotation and detector *lD* is now multiplied with the factor *M-1* 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 lD=0 for all reconstruction algorithms
- Improvement of documentation

4.18 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*

4.19 version 0.1.2

- Added reconstruction algorithm for tilted axis of rotation

4.20 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

CHAPTER

FIVE

BILBLIOGRAPHY

CHAPTER

SIX

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

INDEX