# radontea Documentation

***Release 0.4.13***

**Paul Müller**

**Oct 18, 2022**

# CONTENTS

Radontea is a Python library for computerized tomographic image reconstruction. Radontea implements several iterative and non-iterative reconstruction algorithms. This is the documentaion of radontea version 0.4.13.

CHAPTER

ONE

# INTRODUCTION

There are several methods to compute the inverse **Radon** transform. The module `radontea` implements some of them. I focused on code readability and thorough comments. The result is a collection of algorithms that are suitable for **tea**ching the basics of computerized tomography.

## 1.1 Recommended literature

- Aninash C. Kak and Malcom Slaney. *Principles of Computerized Tomographic Imaging*. Ed. by Robert E. O'Malley. SIAM, 2001, p. 327. ISBM: 089871494X.
- Johann Radon. *Über die Bestimmung von Funktionen durch ihre Integralwerte längs gewisser Mannigfaltigkeiten*. Tech. rep. Leipzig: Berichte über die Verhandlungen der Königlich-Sächsischen Gesellschaft der Wissenschaften zu Leipzig, 1917, pp. 262–277.
- R A Crowther, D J DeRosier, and A Klug. *The Reconstruction of a Three-Dimensional Structure from Projections and its Application to Electron Microscopy*. In: Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences 317.1530 (1970), pp. 319–340. doi: 10.1098/rspa.1970.0119.

## 1.2 Obtaining radontea

If you have Python and `numpy` installed, simply run

```
pip install radontea
```

The source code of radontea is available at https://github.com/RI-imaging/radontea.

## 1.3 Citing radontea

Please cite this package if you are using it in a scientific publication.

This package should be cited like this (replace "x.x.x" with the actual version of radontea that you used):

**cite**

Paul Müller (2013) *radontea: Python algorithms for the inversion of the Radon transform* (Version x.x.x) [Software]. Available at https://pypi.python.org/pypi/radontea/

You can find out what version you are using by typing (in a Python console):

```
>>> import radontea
>>> radontea.__version__
'0.1.4'
```

CHAPTER

TWO

# CODE REFERENCE

## 2.1 Parallel beam geometry

| | |
|---|---|
| *radon_parallel*(arr, angles[, count, max_count]) | Compute the Radon transform (sinogram) of a circular image. |
| *backproject*(sinogram, angles[, filtering, ...]) | 2D backprojection algorithm |
| *fourier_map*(sinogram, angles[, intp_method, ...]) | 2D Fourier mapping with the Fourier slice theorem |
| *integrate*(sinogram, angles[, count, max_count]) | 2D sum-reconstruction with the Fourier slice theorem |
| *art*(sinogram, angles[, initial, iterations, ...]) | Algebraic Reconstruction Technique |
| *sart*(sinogram, angles[, initial, ...]) | Simultaneous Algebraic Reconstruction Technique |

### 2.1.1 Radon transform

radontea.**radon_parallel**(*arr: numpy.ndarray*, *angles: numpy.ndarray*, *count=None*, *max_count=None*) → numpy.ndarray

Compute the Radon transform (sinogram) of a circular image.

The scipy Radon transform performs this operation on the entire image, whereas this implementation requires an input image that has gray-scale values of 0 outside of a circle with diameter equal to the image size.

**Parameters**

- **arr** (*ndarray, shape (N,N)*) – the input image.
- **angles** (*ndarray, length A*) – angles or projections in radians
- **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.

**Returns** **outarr** – Sinogram of the input image. The i'th row contains the projection data of the i'th angle.

**Return type** ndarray of floats, shape (A, N)

**See also:**

**scipy.ndimage.interpolation.rotate** The interpolator used to rotate the image.

### 2.1.2 Non-iterative reconstruction

Computes the inverse Radon transform with non-iterative techniques. The linear system of equations that describes the forward process can be inverted with several algorithms, most notably the backprojection algorithm *radontea.backproject()*. The reconstruction is based on the Fourier slice theorem. A Fourier-based interpolation algorithm is implemented in *radontea.fourier_map()*.

#### 2.1.2.1 Backprojection

radontea.**backproject**(*sinogram: numpy.ndarray*, *angles: numpy.ndarray*, *filtering: str = 'ramp'*, *weight_angles: bool = True*, *padding: bool = True*, *padval: float = 0*, *count=None*, *max_count=None*, *verbose: int = 0*) → numpy.ndarray

2D backprojection algorithm

Computes the inverse of the Radon transform using filtered backprojection.

**Parameters**

- **sinogram** (*ndarray, shape (A,N)*) – Two-dimensional sinogram of line recordings.
- **angles** (*(A,) ndarray*) – Angular positions of the *sinogram* in radians.
- **filtering** (*{'ramp', 'shepp-logan', 'cosine', 'hamming', }* – 'hann'}, optional Specifies the Fourier filter. Either of
  - "ramp": mathematically correct reconstruction
  - "shepp-logan"
  - "cosine"
  - "hamming"
  - "hann"
- **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 0.1.9.
- **padding** (*bool, optional*) – 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. 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** **out** – The reconstructed image.

**Return type** ndarray

#### Examples

```
>>> import numpy as np
>>> from radontea import backproject
>>> N = 5
>>> A = 7
>>> x = np.linspace(-N/2, N/2, N)
>>> projection = np.exp(-x**2)
>>> sinogram = np.tile(projection, A).reshape(A,N)
>>> angles = np.linspace(0, np.pi, A, endpoint=False)
>>> backproject(sinogram, angles)
array([[ 0.03813283, -0.01972347, -0.02221885,  0.03822303,  0.01903376],
       [ 0.04033526, -0.01591647,  0.02262173,  0.04203002,  0.02123619],
       [ 0.02658797,  0.11375576,  0.41525594,  0.17170226,  0.0074889 ],
       [ 0.04008672,  0.11139209,  0.27650193,  0.16933859,  0.02098764],
       [ 0.02140445,  0.0334597 ,  0.07691547,  0.0914062 ,  0.00230537]])
```

### 2.1.2.2 Fourier mapping

radontea.**fourier_map**(*sinogram: numpy.ndarray*, *angles: numpy.ndarray*, *intp_method: str = 'cubic'*, *count=None*, *max_count=None*) → numpy.ndarray

2D Fourier mapping with the Fourier slice theorem

Computes the inverse of the Radon transform using Fourier interpolation. Warning: This is the naive reconstruction that assumes that the image is rotated through the upper left pixel nearest to the actual center of the image. We do not have this problem for odd images, only for even images.

**Parameters**

- **sinogram** (*(A,N) ndarray*) – Two-dimensional sinogram of line recordings.
- **angles** (*(A,) ndarray*) – Angular positions of the *sinogram* in radians equally distributed from zero to PI.
- **intp_method** (*{'cubic', 'nearest', 'linear'}, optional*) – Method of interpolation. For more information see *scipy.interpolate.griddata*. One of
  - ”nearest”: instead of interpolating, use the points closest to the input data
  - ”linear”: bilinear interpolation between data points
  - ”cubic”: interpolate using a two-dimensional poolynimial surface
- **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.

**Returns** **out** – The reconstructed image.

**Return type** ndarray

**See also:**

`scipy.interpolate.griddata` interpolation method used

#### 2.1.2.3 Slow integration

radontea.**integrate**(*sinogram:* numpy.ndarray, *angles:* numpy.ndarray, *count=None*, *max_count=None*) → numpy.ndarray

2D sum-reconstruction with the Fourier slice theorem

Computes the inverse of the Radon transform by computing the integral in real space.

**Parameters**

- **sinogram** (*(A,N) ndarray*) – Two-dimensional sinogram of line recordings.
- **angles** (*(A,) ndarray*) – Angular positions of the *sinogram* in radians equally distributed from zero to PI.
- **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.

**Returns** **out** – The reconstructed image.

**Return type** ndarray

### 2.1.3 Iterative reconstruction

Inversion of Radon-based tomography methods using iterative algorithms. The convegence of these algorithms might be slow. The implementation is not optimized.

#### 2.1.3.1 ART

radontea.**art**(*sinogram:* numpy.ndarray, *angles:* numpy.ndarray, *initial: Optional[numpy.ndarray] = None*, *iterations:* int *= 1*, *count=None*, *max_count=None*) → numpy.ndarray

Algebraic Reconstruction Technique

The Algebraic Reconstruction Technique (ART) iteratively computes the inverse of the Radon transform in two dimensions. The reconstruction technique uses *rays* of the diameter of one pixel to iteratively solve the system of linear equations that describe the projection process. The binary weighting factors are

- 1, if the center of the a pixel is within the *ray*
- 0, else

**Parameters**

- **sinogram** (*ndarrayy, shape (A,N)*) – Two-dimensional sinogram of line recordings.
- **angles** (*ndarray, length A*) – Angular positions of the *sinogram* in radians. The angles at which the sinogram slices were recorded do not have to be distributed equidistantly as in *backproject()*. The angles are internaly converted to modulo PI.
- **initial** (*ndarray, shape (N,N), optional*) – The initial guess for the solution.
- **iterations** (*int*) – Number of iterations to perform.

- **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.

#### Notes

For theoretical backround, see Kak, A. C., & Slaney, M.. *Principles of Computerized Tomographic Imaging*, SIAM, (2001)

Sec. 7.2: *"ART reconstrutions usually suffer from salt and pepper noise, which is caused by the inconsitencies introuced in the set of equations by the approximations commonly used for* $w_{ik}$ *'s."*

### 2.1.3.2 SART

radontea.**sart**(*sinogram: numpy.ndarray*, *angles: numpy.ndarray*, *initial: Optional[numpy.ndarray] = None*, *iterations: int = 1*, *count=None*, *max_count=None*)

Simultaneous Algebraic Reconstruction Technique

SART computes an inverse of the Radon transform in two dimensions. The reconstruction technique uses "rays" of the diameter of one pixel to iteratively solve the system of linear equations that describe the image. The weighting factors are bilinear elements. At the beginning and end of each ray, only partial weights are used. The pixel values of the image are updated only after each iteration is complete.

**Parameters**

- **sinogram** (*ndarrayy, shape (A,N)*) – Two-dimensional sinogram of line recordings.
- **angles** (*ndarray, length A*) – Angular positions of the *sinogram* in radians. The angles at which the sinogram slices were recorded do not have to be distributed equidistantly as in backprojection techniques. The angles are internaly converted to modulo PI.
- **initial** (*ndarray, shape (N,N), optional*) – the initial guess for the solution.
- **iterations** (*integer*) – Number of iterations to perform.
- **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.

#### Notes

**Algebraic reconstruction technique (ART) (see *art*):** Iterations are performed over each ray of each projection. Weighting factors are binary (1 if center of pixel is within ray, 0 else). This leads to salt and pepper noise.

**Simultaneous iterative reconstruction technique (SIRT):** Same idea as ART, but for each iteration, the change of the image f is computed for all rays and projections separately and the weights are applied simultaneously after each iteration. The result is a slower convergence but the final image is also less noisy.

This implementation does NOT use a hamming window to filter the data and to emphasize points at the center of the recon- struction region.

For theoretical backround, see Kak, A. C., & Slaney, M.. *Principles of Computerized Tomographic Imaging*, SIAM, (2001)

Sec 7.4: *"[SART] seems to combine the best of ART and SIRT. [...] Here are the main features if SART: First, [...] the traditional pixel basis is abandonded in favor of bilinear elements [e.g. interpolation]. Also, for a circular reconstruction region, only partial weights are assigned to the first and last picture elements on the individual rays. To further reduce the noise [...], the correction terms are simultaneously applied for all the rays in one projection [...]."*

## 2.2 Fan beam geometry

| | |
|---|---|
| *fan.radon_fan*(arr, det_size[, det_spacing, ...]) | Compute the Radon transform for a fan beam geometry |
| *fan.get_det_coords*(size, spacing) | Compute pixel-center positions for 2D detector |
| *fan.get_fan_coords*(size, spacing, distance, ...) | Compute equispaced angular coordinates for 2d detector |
| *fan.lino2sino*(linogram, lds[, stepsize, ...]) | Convert linogram to sinogram for an equispaced detector. |
| *fan.fan_rec*(linogram, lds, method[, ...]) | 2D synthetic aperture reconstruction |

### 2.2.1 Coordinate transforms

radontea.fan.**get_det_coords**(*size: int*, *spacing: float*) → numpy.ndarray
Compute pixel-center positions for 2D detector

The centers of the pixels of a detector are usually not aligned to a pixel grid. If we center the detector at the origin, odd images will have a pixel at zero, whereas even images will have two pixels next to the actual center.

**Parameters**

- **size** (*int*) – Size of the detector (number of detection points).
- **spacing** (*float*) – Distance between two detection points in pixels.

**Returns** **arr** – Pixel coordinates.

**Return type** 1D ndarray

radontea.fan.**get_fan_coords**(*size: int*, *spacing: float*, *distance: float*, *numang: int*) → numpy.ndarray
Compute equispaced angular coordinates for 2d detector

The centers of the pixels of a detector are usually not aligned to a pixel grid. If we center the detector at the origin, odd images will have a pixel at zero, whereas even images will have two pixels next to the actual center. We want to compute an equispaced array of *numang* angles that go between the centers of the outmost far pixels of the detector.

**Parameters**

- **size** (*int*) – Size of the detector (number of detection points).
- **spacing** (*float*) – Distance between two detection points in pixels.
- **distance** (*float*) – Axial distance from the detector to the angular measurement position in pixels.
- **numang** (*int*) – Number of angles.

**Returns** **angles, latpos** – Angles and pixel coordinates at the detector.

**Return type** two 1D ndarrays

### Notes

Actually one would not need to define spacing and distance, but for convenience, these parameters are separated and an arbitrary uint 'pixel' is defined.

radontea.fan.**radon_fan**(*arr*, *det_size: int*, *det_spacing: float = 1*, *shift_size: int = 1*, *lS=1*, *lD=None*, *return_ang: bool = False*, *count=None*, *max_count=None*) → numpy.ndarray

Compute the Radon transform for a fan beam geometry

In contrast to `radon_parallel()`, this function uses (1) a fan-beam geometry (the integral is taken along rays that meet at one point), and (2) translates the object laterally instead of rotating it. The result is sometimes referred to as 'linogram'.

Problem sketch:

```
x
^
|
----> z

source       object   detector

          /              . (+det_size/2, lD)
(0,-lS) ./   (0,0)       .
         \               .
          \              . (-det_size/2, lD)
```

The algorithm computes all angular projections for discrete movements of the object. The position of the object is changed such that its lower boundary starts at (det_size/2, 0) and its upper boundary ends at (-det_size/2, 0) at increments of *shift_size*.

**Parameters**

- **arr** (*ndarray, shape (N,N)*) – the input image.
- **det_size** (*int*) – The total detector size in pixels. The detector centered to the source. The axial position of the detector is the center of the pixels on the far right of the object.
- **det_spacing** (*float*) – Distance between detection points in pixels.
- **shift_size** (*int*) – The amount of pixels that the object is shifted between projections.
- **lS** (*multiples of 0.5*) – Source position relative to the center of *arr*. lS >= 1.
- **lD** (*int*) – Detector position relative to the center *arr*. Default is N/2.
- **return_ang** (*bool*) – Also return the angles corresponding to the detector pixels.

- **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.

**Returns** **outarr** – Linogram of the input image. Where N+det_size determines the lateral position of the sample.

**Return type** ndarray of floats, shape (N+det_size,det_size)

**See also:**

**scipy.ndimage.interpolation.rotate** The interpolator used to rotate the image.

radontea.fan.**lino2sino**(*linogram: numpy.ndarray*, *lds: float*, *stepsize: float = 1*, *det_spacing: float = 1*, *numang: Optional[int] = None*, *retang: bool = False*, *count=None*, *max_count=None*) → numpy.ndarray

Convert linogram to sinogram for an equispaced detector.

**Parameters**

- **linogram** (*real 2d ndarray of shape (D, A*)*) – Linogram from synthetic aperture measurements.
- **lds** (*float*) – Distance from point source to detector in au.
- **stepsize** (*float*) – Translational increment of object in au (stepsize in D).
- **det_spacing** (*float*) – Distance between detector positions in au.
- **numang** (*int*) – Number of equispaced angles, defaults to linogram.shape[1]
- **retang** (*bool*) – Return the corresponding angles for the sinogram.
- **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.

**Returns** **sinogram** – The distortion-corrected sinogram. If retang is True, then the equispaced angles are returned as well.

**Return type** 2d ndarray of shape (D, A)

### Notes

This function can be used to convert a linogram obtained with fan-beam tomography to a sinogram, which then can be reconstructed with the backprojection or fourier mapping algorithms.

### 2.2.2 Reconstruction

radontea.fan.**fan_rec**(*linogram: numpy.ndarray*, *lds: float*, *method: Callable*, *stepsize=1*, *det_spacing: float = 1*, *numang: Optional[int] = None*, *count=None*, *max_count=None*, ***kwargs*)

2D synthetic aperture reconstruction

Computes the inverse of the fan-beam Radon transform using interpolation of the linogram and one of the inverse algorithms for tomography with the Fourier slice theorem.

**Parameters**

- **linogram** (*2d ndarray of shape (D, A)*) – Input linogram from the synthetic aperture measurement.
- **lds** (*float*) – Distance in pixels between source and detector.
- **method** (*callable*) – Reconstruction method, e.g. *radontea.backproject*.
- **numang** (*int*) – Number of angles to be used for the sinogram. A higher number increases quality, but interpolation takes longer. By default numang = linogram.shape[1].
- **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.
- ****kwargs** (*dict*) – Keyword arguments for *method*.

## 2.3 Volumetric reconstruction

For a slice-wise 3D reconstruction, radontea can use multiprocessing to parallelize the reconstruction process.

### 2.3.1 Convenience functions

radontea.**backproject_3d**(*sinogram: numpy.ndarray*, *angles: numpy.ndarray*, *filtering: str = 'ramp'*, *weight_angles: bool = True*, *padding: bool = True*, *padval: float = 0*, *count=None*, *max_count=None*, *ncpus=None*) → numpy.ndarray

Convenience wrapper for 3D backprojection reconstruction

See *backproject()* for parameter definitions. The additional parameter *ncpus* sets the number of CPUs used.

**Returns**

**out** – The reconstructed volume.

Changed in version 0.4.0: Output indexing now follows the ODTbrain convention. For the the old behavior, use `out.transpose(1, 0, 2)`.

**Return type** ndarray, shape (N,M,N)

radontea.**fourier_map_3d**(*sinogram: numpy.ndarray*, *angles: numpy.ndarray*, *intp_method: str = 'cubic'*, *count=None*, *max_count=None*, *ncpus=None*) → numpy.ndarray

Convenience wrapper for 3D Fourier mapping reconstruction

See *fourier_map()* for parameter definitions. The additional parameter *ncpus* sets the number of CPUs used.

**Returns**

**out** – The reconstructed volume.

Changed in version 0.4.0: Output indexing now follows the ODTbrain convention. For the the old behavior, use `out.transpose(1, 0, 2)`.

**Return type** ndarray, shape (N,M,N)

### 2.3.2 General 3D reconstruction

radontea.**volume_recon**(*func2d: Callable*, *sinogram: Optional[numpy.ndarray] = None*, *angles: Optional[numpy.ndarray] = None*, *count=None*, *max_count=None*, *ncpus=None*, ***kwargs*) → numpy.ndarray

Slice-wise 3D inversion of the Radon transform

Computes the slice-wise 3D inverse of the Radon transform using multiprocessing.

**Parameters**

- **func2d** (*callable*) – A method for the slice-wise reconstruction (e.g. *backproject()*).
- **sinogram** (*ndarray, shape (A,M,N)*) – Three-dimensional sinogram of line recordings. The axis *1* iterates through the *M* slices. The rotation takes place through axis *1*.
- **angles** (*(A,) ndarray*) – Angular positions of the *sinogram* in radians in the interval [0, PI).
- **count** (multiprocessing.Value or *None*) – Can be used to monitor the progress of this function. The value of *max_count.value* is set initially and the value of *count.value* is incremented until it reaches the end of the algorithm (*max_count.value*).
- **max_count** (multiprocessing.Value or *None*) – Can be used to monitor the progress of this function. The value of *max_count.value* is set initially and the value of *count.value* is incremented until it reaches the end of the algorithm (*max_count.value*).
- ****kwargs** (*dict*) – Additional keyword arguments to *func2d*.

**Returns**

**out** – The reconstructed volume.

Changed in version 0.4.0: Output indexing now follows the ODTbrain convention. For the the old behavior, use `out.transpose(1, 0, 2)`.

**Return type** ndarray, shape (N,M,N)

CHAPTER

THREE

# EXAMPLES

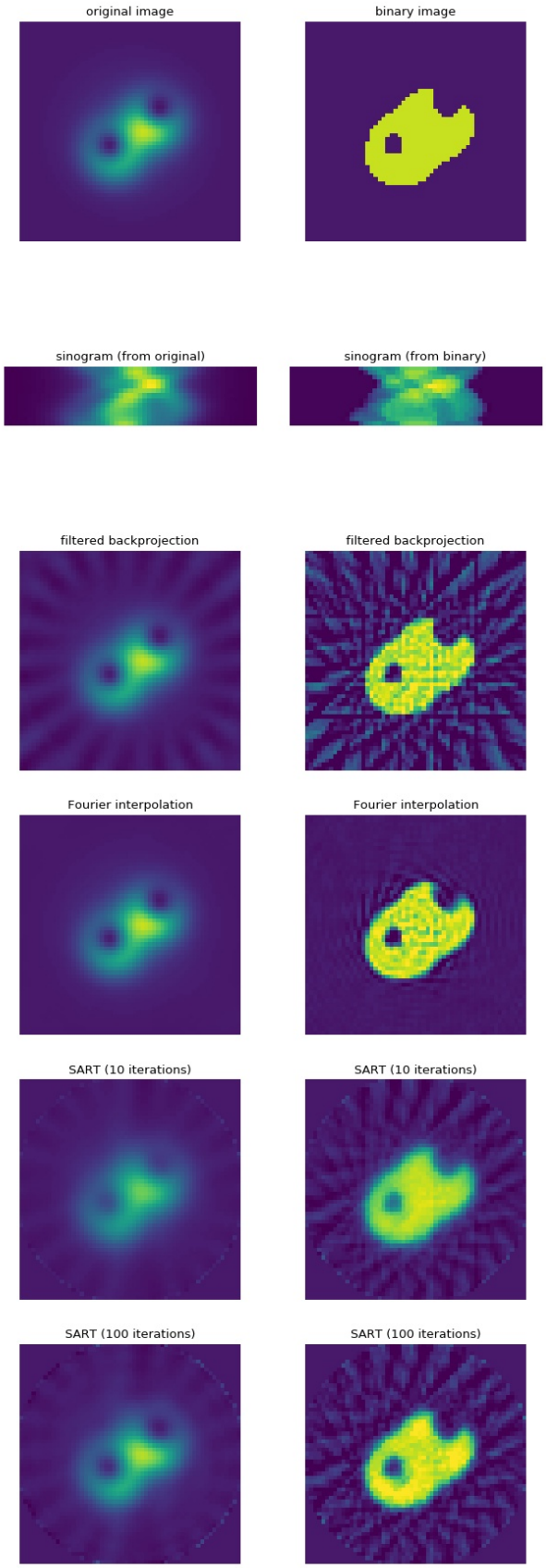
## 3.1 Comparison of parallel-beam reconstruction methods

This example illustrates the performance of the different reconstruction techniques for a parallel-beam geometry. The left column shows the reconstruction of the original image and the right column shows the reconstruction of the corresponding binary images. Note that the SART process could be sped-up by computing an initial guess with a non-iterative method and setting it with the `initial` keyword argument.

comparison_parallel.py

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

import radontea
from radontea.logo import get_original

N = 55  # image size
A = 13  # number of sinogram angles
ITA = 10  # number of iterations a
ITB = 100  # number of iterations b

angles = np.linspace(0, np.pi, A)

im = get_original(N)
sino = radontea.radon_parallel(im, angles)
fbp = radontea.backproject(sino, angles)
fintp = radontea.fourier_map(sino, angles).real
sarta = radontea.sart(sino, angles, iterations=ITA)
sartb = radontea.sart(sino, angles, iterations=ITB)

im2 = (im >= (im.max() / 5)) * 255
sino2 = radontea.radon_parallel(im2, angles)
fbp2 = radontea.backproject(sino2, angles)
fintp2 = radontea.fourier_map(sino2, angles).real
sarta2 = radontea.sart(sino2, angles, iterations=ITA)
sartb2 = radontea.sart(sino2, angles, iterations=ITB)

plt.figure(figsize=(8, 22))
pltkw = {"vmin": -20,
         "vmax": 280}

```

(continues on next page)


original image
binary image
sinogram (from original)
sinogram (from binary)
filtered backprojection
filtered backprojection
Fourier interpolation
Fourier interpolation
SART (10 iterations)
SART (10 iterations)
SART (100 iterations)
SART (100 iterations)

(continued from previous page)

```
plt.subplot(6, 2, 1, title="original image")
plt.imshow(im, **pltkw)
plt.axis('off')

plt.subplot(6, 2, 2, title="binary image")
plt.imshow(im2, **pltkw)
plt.axis('off')

plt.subplot(6, 2, 3, title="sinogram (from original)")
plt.imshow(sino)
plt.axis('off')

plt.subplot(6, 2, 4, title="sinogram (from binary)")
plt.imshow(sino2)
plt.axis('off')

plt.subplot(6, 2, 5, title="filtered backprojection")
plt.imshow(fbp, **pltkw)
plt.axis('off')

plt.subplot(6, 2, 6, title="filtered backprojection")
plt.imshow(fbp2, **pltkw)
plt.axis('off')

plt.subplot(6, 2, 7, title="Fourier interpolation")
plt.imshow(fintp, **pltkw)
plt.axis('off')

plt.subplot(6, 2, 8, title="Fourier interpolation")
plt.imshow(fintp2, **pltkw)
plt.axis('off')

plt.subplot(6, 2, 9, title="SART ({} iterations)".format(ITA))
plt.imshow(sarta, **pltkw)
plt.axis('off')

plt.subplot(6, 2, 10, title="SART ({} iterations)".format(ITA))
plt.imshow(sarta2, **pltkw)
plt.axis('off')

plt.subplot(6, 2, 11, title="SART ({} iterations)".format(ITB))
plt.imshow(sartb, **pltkw)
plt.axis('off')

plt.subplot(6, 2, 12, title="SART ({} iterations)".format(ITB))
plt.imshow(sartb2, **pltkw)
plt.axis('off')


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

## 3.2 Volumetric data reconstruction benchmark

This simple example can be used to quantify the speed-up due to multiprocessing on the hardware used. Note that `multiprocessing.cpu_count` does not return the number of physical cores.

benchmark_3d.py

```
from multiprocessing import cpu_count
import time

import numpy as np

import radontea as rt


A = 70    # number of angles
N = 128   # detector size x
M = 24    # detector size y (number of slices)

# generate random data
sino0 = np.random.random((A, N))      # for 2d example
sino = np.random.random((A, M, N))    # for 3d example
sino[:, 0, :] = sino0
angles = np.linspace(0, np.pi, A)     # for both

a = time.time()
data1 = rt.backproject_3d(sino, angles, ncpus=1)
print("time on 1 core:  {:.2f} s".format(time.time() - a))

a = time.time()
data2 = rt.backproject_3d(sino, angles, ncpus=cpu_count())
print("time on {} cores: {:.2f} s".format(cpu_count(), time.time() - a))

assert np.all(data1 == data2), "2D and 3D results don't match"
```

## 3.3 Progress monitoring with progression

The progress of the reconstruction algorithms in radontea can be tracked with other packages such as progression.

```
$python progression_3d.py
00:00:14 [65.2c/s] [============================>              ] TTG 7.00s
```

progression_3d.py

```
from multiprocessing import cpu_count

import numpy as np
import progression as pr

import radontea as rt
```

(continues on next page)

(continued from previous page)

```


A = 55    # number of angles
N = 128   # detector size x
M = 24    # detector size y (number of slices)

# generate random data
sino0 = np.random.random((A, N))
sino = np.random.random((A, M, N))
sino[:, 0, :] = sino0
angles = np.linspace(0, np.pi, A)

count = pr.UnsignedIntValue()
max_count = pr.UnsignedIntValue()


with pr.ProgressBar(count=count,
                    max_count=max_count,
                    interval=0.3
                    ) as pb:
    pb.start()
    rt.volume_recon(func2d=rt.sart, sinogram=sino, angles=angles,
                    ncpus=cpu_count(), count=count, max_count=max_count)
```

CHAPTER

FOUR

# CHANGELOG

List of changes in-between radontea releases.

## 4.1 version 0.4.13

- ci: skip tests that fail on GitHub Actions (Linux/macOS)

## 4.2 version 0.4.12

- maintenance release

## 4.3 version 0.4.11

- ref: cleanup

## 4.4 version 0.4.10

- fix: handle more special cases when computing weights for projections (#7)
- ref: make *util* submodule available upon import

## 4.5 version 0.4.9

- build: setup.py test is deprecated
- build: moved from travisCI to GitHub Actions
- docs: refurbish docs
- tests: allow 2D Fourier mapping test to fail on macOS and Windows (don't know why it fails, probably unstable)
- ref: replace np.complex256 with np.complex128 in _alg_int.py:integrate which does not break any tests but works on Windows which sometimes has not support for longdouble
- ref: replace np.bool with bool due to numpy deprecation warnings

## 4.6 version 0.4.8

- ref: move sinogram generation in tests to separate file, thanks @SZanko
- ref: deprecation warning int for numpy 1.20.0, thanks @SZanko
- ref: more type hints, thanks @SZanko

## 4.7 version 0.4.7

- ref: added PEP 484 type hints, thanks @SZanko (#3)

## 4.8 version 0.4.6

- maintenance release

## 4.9 version 0.4.5

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

## 4.10 version 0.4.4

- fix: correctly increment *count* in *_alg_fmp.py*

## 4.11 version 0.4.3

- fix: correctly compute *max_count* for 3D wrapper

## 4.12 version 0.4.2

- fix: use *Value.get_lock()* when tracking progress
- docs: minor improvements

## 4.13 version 0.4.1

- docs: add index link and include changelog

## 4.14 version 0.4.0

- BREAKING CHANGE: volumetric reconstruction is now done according to the ODTbrain indexing convention. To get the original behavior, transpose the output, i.e. `volout.transpose(1, 0, 2)`.

## 4.15 version 0.3.2

- ci: automated deployment with travis-ci

## 4.16 version 0.3.1

- Convenience fix: Make algorithm source files private

## 4.17 version 0.3.0

- BREAKING CHANGES:
  - Renamed several functions
  - Dropped support for Python 2
- Refactoring:
  - Moved each reconstruction algorithm to a separate file
  - Modified code to comply with PEP8
  - Moved long doc strings from source to docs directory
  - Removed complicated and redundant 3D reconstruction methods
  - Improved 3D algorithm (*threed.py*) to support progress-monitoring

## 4.18 version 0.2.1

- Do not include compiled docs in sdist

## 4.19 version 0.2.0

- Updated docs and examples (#1)
- Moved docs to readthedocs.io

## 4.20 version 0.1.9

- No code changes

## 4.21 version 0.1.8

- Added support for NumPy 1.10

## 4.22 version 0.1.7

- Improved padding for backprojection

## 4.23 version 0.1.6

- Removed a memory leak
- Increase compatibility to jobmanager
- Improved documentation

## 4.24 version 0.1.5

- Fixed an example
- Fixed bug in 3D reconstruction

## 4.25 version 0.1.4

- Added 3D reconstruction methods

## 4.26 version 0.1.3

- Switched to jobamanger package for tracking of progress
- Adde fan-beam helper methods

## 4.27 version 0.1.2

- Renamed callback functions

## 4.28 version 0.1.1

- Mainly code cleanup

## 4.29 version 0.1.0

- Initial GitHub commit

CHAPTER

FIVE

# INDICES AND TABLES

- genindex
- modindex
- search

# INDEX