gratopy

Release 0.1.0

Kristian Bredies, Richard Huber

Mar 29, 2023

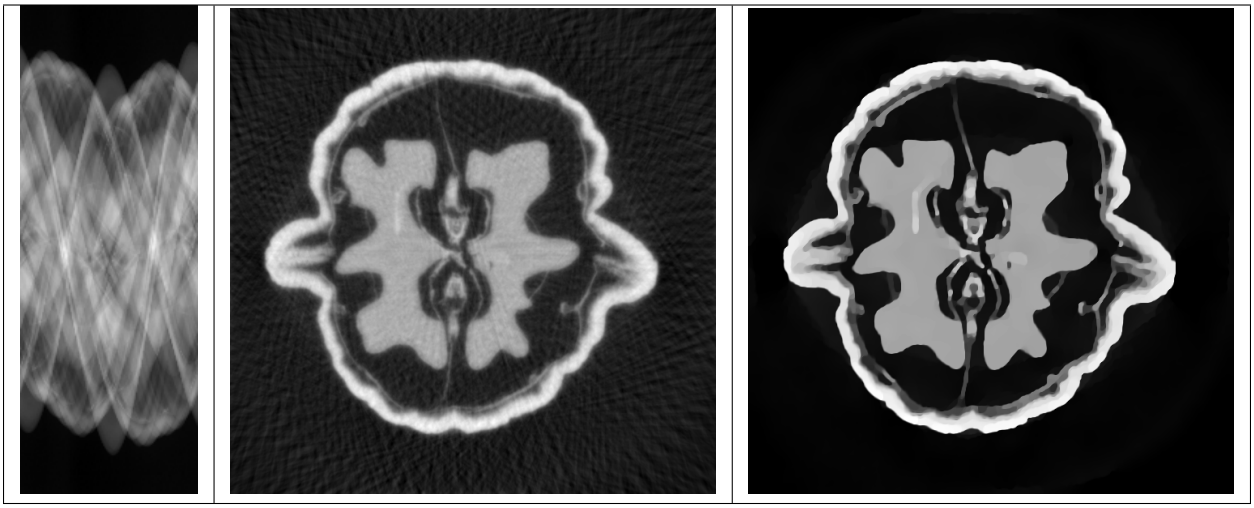
CONTENTS:

The gratopy (**Gr**az **a**ccelerated **to**mographic projections for **Py**thon) toolbox is a Python3 software package for the efficient and high-quality computation of Radon transforms, fanbeam transforms as well as the associated backprojections. The included operators are based on pixel-driven projection methods which were shown to possess favorable approximation properties. The toolbox offers a powerful parallel OpenCL/GPU implementation which admits high execution speed and allows for seamless integration into PyOpenCL. Gratopy can efficiently be combined with other PyOpenCL code and is well-suited for the development of iterative tomographic reconstruction approaches, in particular, for those involving optimization algorithms.

Highlights

- Easy-to-use tomographic projection toolbox.
- High-quality 2D projection operators.
- Fast projection due to custom OpenCL/GPU implementation.
- Seamless integration into PyOpenCL.
- Basic iterative reconstruction schemes included (Landweber, CG, total variation).
- Comprehensive documentation, tests and example code.

Table 1: The fanbeam projection of a walnut and gratopy's Landweber and total variation reconstructions (from left to right).



CHAPTER

ONE

INSTALLATION

Gratopy supports common Python package distribution frameworks such as setuptools or pip.

1.1 Installation in Python

The gratopy toolbox can easily be installed using pip

```
pip install gratopy
```

Alternatively, the release can be downloaded from https://github.com/kbredies/gratopy and installed (after unpacking inside the corresponding folder) via

```
pip install .
```

Also, setuptools can be used for installation via

```
python setup.py install
```

In case installation fails due to the dependency on other packages (see requirements.txt), it is advised to install the packages by hand before retrying to install gratopy. In particular, the PyOpenCL package may require some additional effort as it depends on additional drivers and C libraries which might needed to be installed by hand. We refer to the documentation of PyOpenCL.

1.2 Testing correct installation

The release archive (or GitHub repository) includes a `tests` folder which contains a variety of tests that allow to observe visually and numerically whether gratopy was installed correctly and works as desired.

One can perform these tests by using, for instance, pytest

```
pytest
```

or nose

```
nosetests
```

In case multiple OpenCL devices are registered in `pyopencl`, but the default device is not suitably configured for the tests to work, one might need to choose the context to use manually. This a-priori choice of context to use in `pyopencl` can be done via

```
export PYOPENCL_CTX=<context_number>
```

The context number can, for instance, be determined in Python by

```
import pyopencl
pyopencl.create_some_context()
```

following the interactive instructions and observing the console output.

By default, the plots of the tests are disabled, but can be activated, e.g., by

```
export GRATOPY_TEST_PLOT=true
```

Moreover, the *Getting started* guide contains two example code segments which can be executed to quickly check that no errors occur and the output is as desired.

1.3 Requirements

The requirements.txt file specifies Python packages required for the use of gratopy. Amongst them the most relevant are

- pyopencl>=2019.1
- numpy>=1.17.0
- scipy>=1.3.0
- matplotlib>=3.2.0
- Pillow>=6.0.0
- Mako>=1.1.0

Most users aiming for scientific computing applications will probably have these packages already installed as they can be considered standard for numerical computations in Python. Let us again point out that correctly installing PyOpenCL might take some time and effort though, as dependent on the used hardware/GPU, the installation of suitable drivers might be required, see, for instance, https://documen.tician.de/pyopencl/.

CHAPTER

TWO

GETTING STARTED

2.1 Basic principles of gratopy

We start by explaining some recurring relevant quantities and concepts in gratopy, in particular the *ProjectionSettings* class as well as the use of images and sinograms and the connection of forward projection to backprojection in the context of gratopy.

2.1.1 ProjectionSettings

The cornerstone of the gratopy toolbox is formed by the *gratopy.ProjectionSettings* class, which defines the considered geometry, collects all relevant information to create the OpenCL kernels, and precomputes and saves relevant quantities. Thus, virtually all functions of gratopy require an object of this class, usually referred to as **projectionsetting**. In particular, gratopy offers the implementation for two different geometric settings, the **parallel beam** and the **fanbeam** setting.

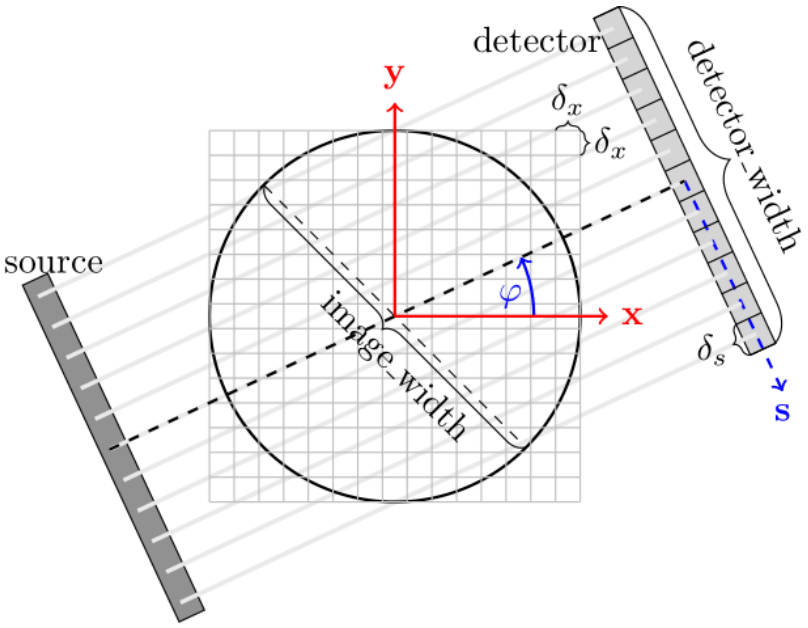
The geometry of the **parallel beam setting** is mainly defined by the **image_width** – the physical diameter of the object in question in arbitrary units, e.g., 3 corresponding to 3cm (or m, etc.) – and the **detector_width** – the physical width of the detector in the same unit –, both parameters of a **projectionsetting**. For most standard examples for the Radon transform, these parameters coincide, i.e., the detector is exactly as wide as the diameter of the imaged object, and thus, captures all rays passing through the object.

The **fanbeam setting** additionally requires **RE** – the physical distance from the source to the center of rotation – and **R** – the physical distance from the source to the detector – to define the geometry, see the figures below.

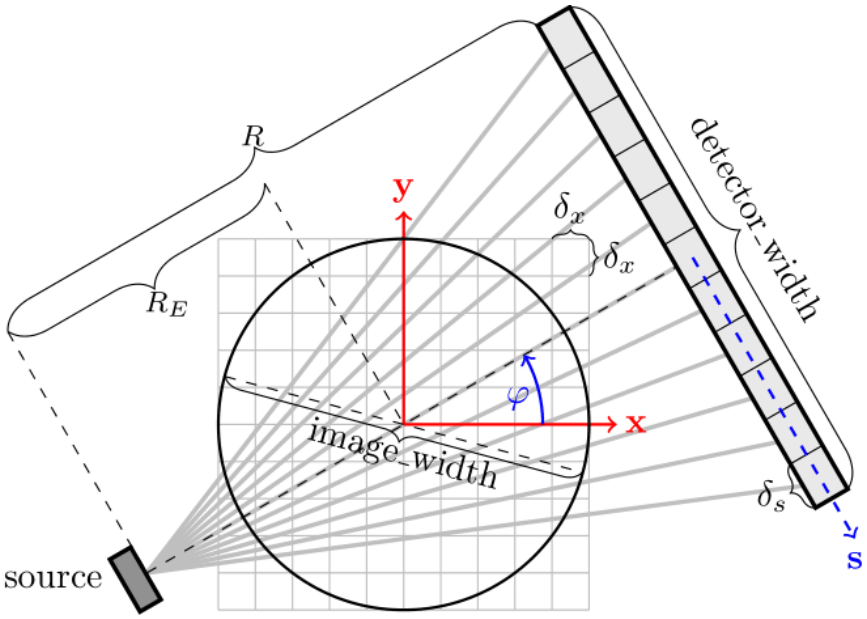
Moreover, the projection requires discretization parameters, i.e., the shape of the image to project and the number of detector pixels to map to. Note that these transforms are scaling-invariant in the sense that rescaling all *physical* quantities by the same factor creates operators which are rescaled versions of the original ones. On the other hand, changing the number of pixels of the image or the detector leaves the physical system unchanged and simply reflects a finer/coarser discretization.

The angular range for the parallel beam setting is $[0, \pi[$, while for the fanbeam setting, it is $[0, 2\pi[$. By default, it is assumed that the given angles completely partition the angular range. In case this is not desired, a limited-angle situation can be considered by adapting the **angles** and **angle_weights** parameters of *gratopy.ProjectionSettings*, impacting, for instance, the backprojection operator. Note also that the projections considered are rotation-invariant in the sense, that projection of a rotated image yields a sinogram which is translated in the angular dimension.

Note that the angles are measured (counterclockwise) from the positive x axis and reflect the projection-direction. The positive detector-direction is the clockwise rotation of the projection-direction by $\frac{\pi}{2}$.



Geometry of the parallel beam setting.



Geometry of the fanbeam setting.

The main functions of gratopy are *forwardprojection* and *backprojection*, which use a **projectionsetting** as the basis for computation and allow to project an image **img** onto a sinogram **sino** and to backproject **sino** onto **img**, respectively. Next, we describe how to use and interpret images and sinograms in gratopy.

2.1.2 Images in gratopy

An image **img** is represented in gratopy by a `pyopencl.array.Array` of dimensions (N_x, N_y) – or (N_x, N_y, N_z) for multiple slices – representing a rectangular grid of equidistant quadratic pixels of size $\delta_x = \text{image\_width}/\max\{N_x, N_y\}$, where the associated values correspond to the average mass inside the area covered by each pixel. The area covered by the pixels is called the image domain, and the image array can be associated with a piecewise constant function on the image domain. Usually, we think of the investigated object as being circular and contained in the rectangular image domain. More generally, **image_width** corresponds to the larger side length of a rectangular (N_x, N_y) grid of quadratic image pixels which allows considering *slim* objects. The image domain is, however, always a rectangle or square that is aligned with the *x* and *y* axis. When using an image together with **projectionsetting** – an instance of *gratopy.ProjectionSettings* – the values (N_x, N_y) have to coincide with the attribute **img_shape** of **projectionsetting**, we say they need to be **compatible**. The data type of this array must be `numpy.float32` or `numpy.float64`, i.e., single or double precision, and can have either *C* or *F* contiguity.

Note that in gratopy, the first and second axis of an image array corresponds to the *x* and *y* axis, respectively, as depicted in the figures above.

2.1.3 Sinograms in gratopy

Similarly, a sinogram **sino** is represented by a `pyopencl.array.Array` of the shape (N_s, N_a) or (N_s, N_a, N_z) for N_s being the number of detectors and N_a being the number of angles for which projections are considered. When used together with a **projectionsetting** of class *gratopy.ProjectionSettings*, these dimensions must be **compatible**, i.e., (N_s, N_a) has to coincide with the **sinogram_shape** attribute of **projectionsetting**. The width of the detector is given by the attribute **detector_width** of **projectionsetting** and the detector pixels are equidistantly partitioning the detector line with detector pixel width $\delta_s = \text{detector\_width}/N_s$. The angles, on the other hand, do not need to be equidistant or even partition the entire angular range; gratopy allows for rather general angle sets. The values associated with pixels in the sinogram again correspond to the average intensity values of a continuous sinogram counterpart and thus can be associated with a piecewise constant function. The data type of this array must be `numpy.float32` or `numpy.float64`, i.e., single or double precision, and can have either *C* or *F* contiguity.

2.1.4 Adjointness in gratopy

Gratopy allows a great variety of geometric setups for the forward projection and the backprojection. One particular feature is that forward projection and backprojection are adjoint operators, which is important, for instance, in the context of optimization algorithms. Here, adjointness is achieved with respect to natural scalar products in image and sinogram Hilbert space that we wish to clarify in the following. As described above, the discrete values in an image array are associated with values of piecewise constant functions inside square pixels (of area δ_x^2) in the image domain. For such piecewise constant functions, the classical L^2 scalar product is considered, which results in $\langle \text{img1}, \text{img2} \rangle = \delta_x^2 \sum_{x,y} \text{img1}_{x,y} \text{img2}_{x,y}$ for image arrays **img1** and **img2**. Similarly, the discrete values of the sinogram are associated with a piecewise constant function on the Cartesian product of an interval of length **detector_width** and the angular domain. Correspondingly, the natural inner product for the sinogram space is given by $\langle \text{sino1}, \text{sino2} \rangle = \delta_s \sum_{s,a} \Delta_a \text{sino1}_{s,a} \text{sino2}_{s,a}$, where Δ_a denotes the length of the angular range covered (in the sense of piecewise constant discretization) by the a-*th* angle (by default, all Δ_a are determined automatically based on the **angles** parameter, for more information on **angle_weights**, see *gratopy.ProjectionSettings*). Hence, the implementations of the forward and backprojection in gratopy are to be understood in this context, and in particular, the forward projection and backprojection operator are adjoint with respect to these scalar products, as can be observed in *tests.test_radon.test_adjointness()* and *tests.test_fanbeam.test_adjointness()*.

Though this is, in a sense, the natural discretization and sense of adjointness, it might be of interest to consider adjointness in a different sense. In this respect, gratopy allows to alter the sinogram space by manually setting the angle weights $(\Delta_a)_a$ to desired values, which changes the weights in the backprojection, but always leads to an adjoint operator in the sense of the aforementioned scalar products.

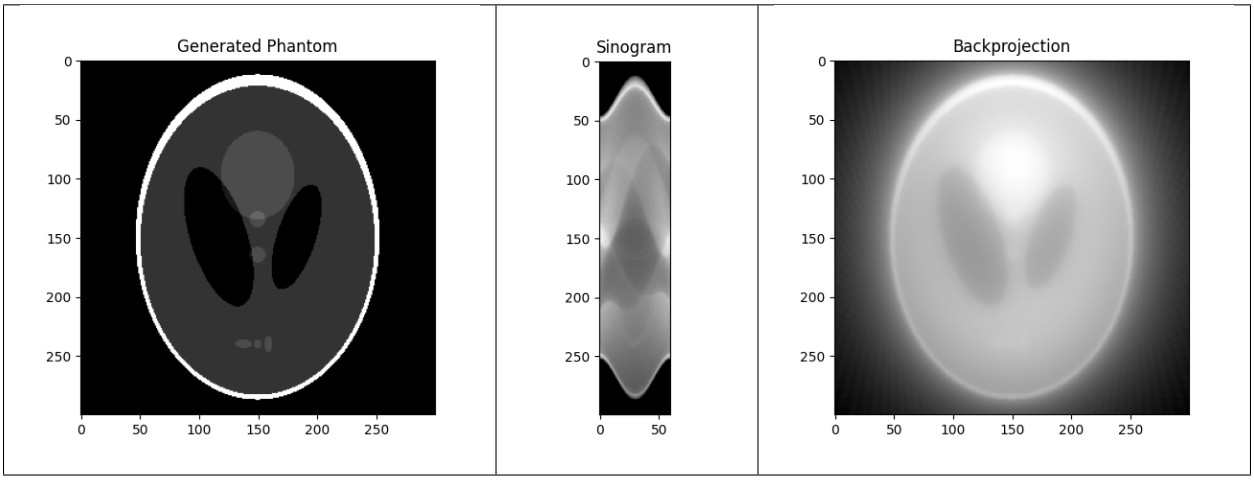
For example, all angles can be weighted equally with 1 in a sparse angle setting. When setting **angle_weights** $\Delta_a = \frac{\delta_x^2}{\delta_s}$, the operators are adjoint with respect to the standard scalar products $\langle \text{img1}, \text{img2} \rangle = \sum_{x,y} \text{img1}_{x,y} \text{img2}_{x,y}$ and $\langle \text{sino1}, \text{sino2} \rangle = \sum_{s,a} \text{sino1}_{s,a} \text{sino2}_{s,a}$.

2.2 First example: Radon transform

One can start in Python via the following simple code which computes the forward and backprojection of a phantom:

```
# initial import
import numpy as np
import pyopencl as cl
import matplotlib.pyplot as plt

import gratopy

# discretization parameters
number_angles = 60
number_detectors = 300
Nx = 300
# Alternatively to number_angles one could give as angle input
# angles = np.linspace(0, np.pi, number_angles+1)[:-1]

# create pyopencl context
ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)

# create phantom as test image (a pyopencl.array.Array of dimensions (Nx, Nx))
phantom = gratopy.phantom(queue,Nx)

# create suitable projectionsettings
PS = gratopy.ProjectionSettings(queue, gratopy.RADON, phantom.shape,
                                number_angles, number_detectors)

# compute forward projection and backprojection of created sinogram
# results are pyopencl arrays
sino = gratopy.forwardprojection(phantom, PS)
backproj = gratopy.backprojection(sino, PS)

# plot results
plt.figure()
plt.title("Generated Phantom")
plt.imshow(phantom.get(), cmap="gray")

plt.figure()
plt.title("Sinogram")
plt.imshow(sino.get(), cmap="gray")

plt.figure()
plt.title("Backprojection")
plt.imshow(backproj.get(), cmap="gray")
plt.show()
```

The following depicts the plots created by this example.



2.3 Second example: Fanbeam transform

As a second example, we consider a fanbeam geometry that has a detector that is 120 (cm) wide, the distance from the the source to the center of rotation is 100 (cm), while the distance from the source to the detector is 200 (cm). We do not choose the **image_width** but rather let gratopy automatically determine a suitable **image_width**. We visualize the defined geometry via the *gratopy.ProjectionSettings.show_geometry* method.

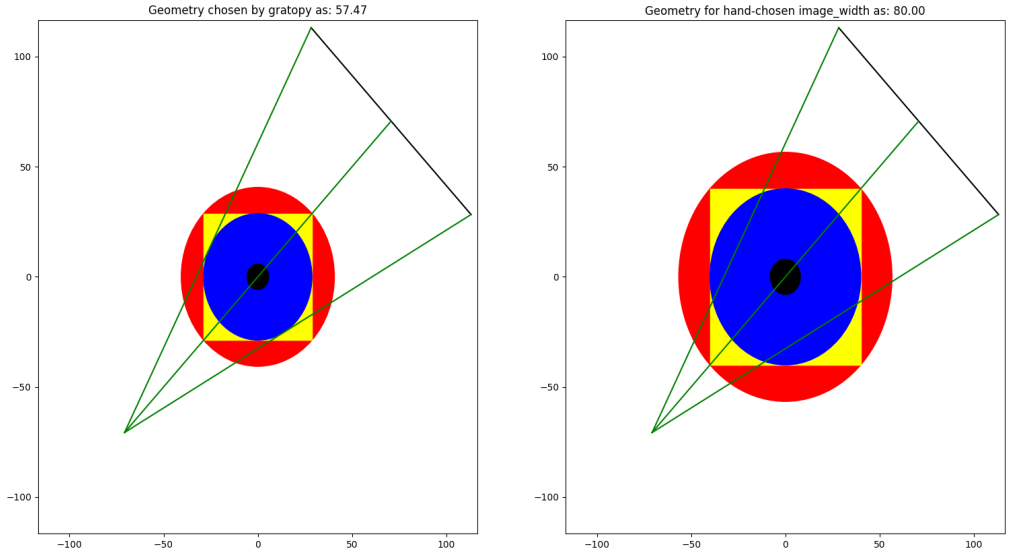
```
# initial import
import numpy as np
import pyopencl as cl
import matplotlib .pyplot as plt

import gratopy

# discretization parameters
number_angles = 60
number_detectors = 300
image_shape = (500, 500)

# create pyopencl context
ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)

# physical parameters
my_detector_width = 120
my_R = 200
my_RE = 100

# fanbeam setting with automatic image_width
PS1 = gratopy.ProjectionSettings(queue, gratopy.FANBEAM,
                                 img_shape=image_shape,
                                 angles=number_angles,
                                 n_detectors=number_detectors,
```

(continues on next page)

(continued from previous page)

```
                                    detector_width=my_detector_width,
                                    R=my_R, RE=my_RE)


print("image_width chosen by gratopy: {:.2f}".format((PS1.image_width)))

# fanbeam setting with set image_width
my_image_width = 80.0
PS2 = gratopy.ProjectionSettings(queue, gratopy.FANBEAM,
                                    img_shape=image_shape,
                                    angles=number_angles,
                                    n_detectors=number_detectors,
                                    detector_width=my_detector_width,
                                    R=my_R, RE=my_RE,
                                    image_width=my_image_width)

# plot geometries associated to these projectionsettings
fig, (axes1, axes2) = plt.subplots(1,2)
PS1.show_geometry(np.pi/4, figure=fig, axes=axes1, show=False)
PS2.show_geometry(np.pi/4, figure=fig, axes=axes2, show=False)
axes1.set_title("Geometry chosen by gratopy as: {:.2f}".format((PS1.image_width)))
axes2.set_title("Geometry for manually-chosen image_width as: {:.2f}"
                .format((my_image_width)))
plt.show()
```

Once the geometry has been defined via the **projectionsetting**, forward and backprojections can be used just like for the Radon transform in the first example. Note that the automatism of gratopy chooses **image_width** = 57.46 (cm). When looking at the corresponding plot via *gratopy.ProjectionSettings.show_geometry*, the **image_width** is such that the entirety of an object inside the blue circle (with diameter 57.46) is exactly captured by each projection, and thus, the area represented by the image corresponds to the yellow rectangle and blue circle which is the smallest rectangle to capture the entire object. On the other hand, the outer red circle illustrates the diameter of the smallest circular object entirely containing the image.



Plot produced by *gratopy.ProjectionSettings.show_geometry* for the fanbeam setting with automatic and manually chosen **image_width**, both for projection from 45°.

Further examples can be found in the source files of the *Test examples*.

CHAPTER

THREE

TEST EXAMPLES

The following documents a number of tests covering essentially all functions and features contained in the package. These functions serve the double purpose of showing that the package is indeed working as desired (via `pytest` or `nosetests`, see *Installation*), and illustrating to users how to set various parameters of the gratopy toolbox and what their effect are (cf. the source code for the tests).

The tests are also able to produce plots of the results. To turn on plotting, the environment variable `GRATOPY_TEST_PLOT` needs to be set, e.g. the command

```
GRATOPY_TEST_PLOT=true pytest
```

can be issued in the `gratopy` directory.

3.1 Radon transform

`tests.test_radon.`**`test_projection`**`()`

Basic projection test. Simply computes forward and backprojection of the Radon transform for two test images in order to visually confirm the correctness of the method. This projection is repeated 10 times to estimate the required time per execution.

`tests.test_radon.`**`test_types_contiguity`**`()`

Types and contiguity test. Runs forward and backprojections for parallel beam geometry for different precision and contiguity settings, checking that they all lead to the same results.

`tests.test_radon.`**`test_weighting`**`()`

Mass preservation test. Check whether the total mass of an image (square with side length 4/3 and pixel values, i.e. density, 1) is correctly transported into the total mass of a projection, i.e., the scaling is adequate.

`tests.test_radon.`**`test_adjointness`**`()`

Adjointness test. Creates random images and sinograms to check whether forward and backprojection are indeed adjoint to one another (by comparing the corresponding dual pairings). This comparison is carried out for 100 experiments to affirm adjointness with some certainty.

`tests.test_radon.`**`test_nonquadratic`**`()`

Non-quadratic image test. Tests and illustrates the projection operator for non-quadratic images.

`tests.test_radon.`**`test_limited_angles`**`()`

Limited angle test. Tests and illustrates how to set the angles in case of limited angle situation, in particular showing artifacts resulting from the incorrect use for the limited angle setting (leading to undesired angle_weights). This can be achieved through the format of the **angles** parameter or by setting the **angle_weights** directly as shown in the test.

tests.test_radon.**test_angle_input_variants**()

Angle parameter input test. Illustrates all possibilities to specify projection angles, checks the resulting **angles** and **angle_weights** as well as tests the possibility to set the **angle_weights** manually.

tests.test_radon.**test_midpoint_shift**()

Shifted midpoint test. Tests and illustrates how the sinogram changes if the midpoint of an images is shifted away from the center of rotation.

tests.test_radon.**test_create_sparse_matrix**()

Tests the *create_sparse_matrix* method to create a sparse matrix associated with the transform, and tests it by applying forward and backprojection via matrix multiplication.

3.2 Fanbeam transform

tests.test_fanbeam.**test_projection**()

Basic projection test. Computes the forward and backprojection of the fanbeam transform for two test images to visually confirm the correctness of the method. This projection is repeated 10 times to estimate the required time per execution.

tests.test_fanbeam.**test_types_contiguity**()

Types and contiguity test. Types and contiguity test. Runs forward and backprojections for fanbeam geometry for different precision and contiguity settings, checking that they all lead to the same results.

tests.test_fanbeam.**test_weighting**()

Mass preservation test. Checks whether the total mass of an image is correctly transported into the total mass of a projection. Due to the fan geometry, the width of a projected object on the detector is wider than the original object was, as the width of the fan grows linearly with the distance it travels. Consequently, also the total mass on the detector is roughly the multiplication of the total mass in the object by the ratio **R** to **RE**. This estimate is verified numerically.

tests.test_fanbeam.**test_adjointness**()

Adjointness test. Creates random images and sinograms to check whether forward and backprojection are indeed adjoint to one another (by comparing the corresponding dual pairings). This comparison is carried out for 100 experiments to affirm adjointness with some certainty.

tests.test_fanbeam.**test_nonquadratic**()

Non-quadratic image test. Tests and illustrates the projection operator for non-quadratic images.

tests.test_fanbeam.**test_limited_angles**()

Limited angle test. Tests and illustrates how to set the angles in case of limited angle situation, in particular showing artifacts resulting from the incorrect use for the limited angle setting (leading to undesired angle_weights). This can be achieved through the format of the **angles** parameter or by setting the **angle_weights** directly as shown in the test.

tests.test_fanbeam.**test_midpoint_shift**()

Shifted midpoint test. Tests and illustrates how the sinogram changes if the midpoint of an images is shifted away from the center of rotation.

tests.test_fanbeam.**test_geometric_orientation**()

Geometric orientation test. Considers projections with parallel and fanbeam geometry for very simple images in different shifted geometries to illustrate how the geometry of the projection work and that they indeed behave analogously for parallel and fanbeam setting. Note that the axes of the images shown by `matplotlib.pyplot.imshow()` are always rotated by 90 degrees compared to the standard (x, y)-axes.

tests.test_fanbeam.**test_range_check_walnut**()

The walnut data set from [HHKKNS2015] is considered for testing the implementation. This test observes that with suitable parameters, the data is well-explained by the model defined by gratopy's operators. In particular, one can observe that there is a slight imperfection in the data set as the detector is not perfectly centered. Indeed, the total mass of the upper detector-half theoretically needs to coincide with the lower detector-half's total mass (up to numerical precision), but these values differ significantly. Moreover, this test serves to verify the validity of the conjugate gradients (CG) method. It is well-known that the CG algorithm approximates the minimal-norm least squares solution to the data, and in particular, the forward projection of this solution corresponds to the projection of data onto the range of the operator. As depicted in the plots of the residual data shown by this test, the walnut projection data admit, after **detector_shift** correction, only slight intensity variations as systematic error.

tests.test_fanbeam.**test_landweber**()

Landweber reconstruction test. Performs the Landweber iteration to compute a reconstruction from a sinogram contained in the walnut data set of [HHKKNS2015], testing the implementation.

tests.test_fanbeam.**test_conjugate_gradients**()

Conjugate gradients reconstruction test. Performs the conjugate gradients iteration to compute a reconstruction from a sinogram contained in the walnut data set of [HHKKNS2015], testing the implementation.

tests.test_fanbeam.**test_total_variation**()

Total variation reconstruction test. Performs the toolbox's total-variation-based approach to compute a reconstruction from a sinogram contained in the walnut data set of [HHKKNS2015], testing the implementation.

tests.test_fanbeam.**test_create_sparse_matrix**()

Tests the *create_sparse_matrix* method to create a sparse matrix associated with the transform, and tests it by applying forward and backprojection via matrix multiplication.

CHAPTER

FOUR

FUNCTION REFERENCE

4.1 Definition of geometry

A cornerstone in applying projection methods is to define for which geometry the projection has to be computed. Thus, the first step in using gratopy is always creating an instance of *gratopy.ProjectionSettings* defining the geometry, and thus internally precomputing relevant quantities.

class gratopy.**ProjectionSettings**(*queue*, *geometry*, *img_shape*, *angles*, *n_detectors=None*, *angle_weights=None*, *detector_width=2.0*, *image_width=None*, *R=None*, *RE=None*, *detector_shift=0.0*, *midpoint_shift=[0.0, 0.0]*, *reverse_detector=False*)

Creates and stores all relevant information concerning the projection geometry. Serves as a parameter for virtually all gratopy's functions.

Parameters

- **queue** (pyopencl.CommandQueue) – The OpenCL command queue with which the computations are associated.
- **geometry** (int) – Determines whether parallel beam (gratopy.RADON) or fanbeam geometry (gratopy.FANBEAM) is considered.
- **img_shape** (tuple (N_x, N_y)) – The number of pixels of the image in x- and y-direction respectively, i.e., the image dimension. It is assumed that by default, the center of rotation is in the middle of the grid of quadratic pixels. The midpoint can, however, be shifted, see the **midpoint_shift** parameter.
- **angles** (int, list[float] / numpy.ndarray, list[tuple(int/list[float]/numpy.ndarray, float, float)]) – Determines which angles are considered for the projection. An integer is interpreted as the number N_a of uniformly distributed angles in the angular range $[0, \pi[$, $[0, 2\pi[$ for Radon and fanbeam transform, respectively, where for negative integers the same angles according to its modulus but with reversed order are generated. Alternatively, the angles can be given explicitly as a list or numpy.ndarray. These two options also imply a full angle setting (as opposed to limited angle setting).

  A limited angle setting can be specified in two ways. First, a list of angular range sections can be passed as input. An angular range section is a tuple with either an integer or a list/array of angles (first element) together with a pair specifying the lower and upper bound of the angular range interval (second and third element), i.e., of type tuple(int, float, float) or tuple(list[float], float, float). If the first element is an integer, the angular interval will be uniformly partitioned into the modulus number of angles (note that the first and last angles are not the lower/upper bounds to ensure uniform angle weights) again in increasing or decreasing order, depending on the sign. Otherwise, a list or array

specifying the individual angles is expected. In particular, multiple angular sections can be specified, by passing a list of angular range sections.

Alternatively, one can use a list of angles and set **angle_weigths** (see below) manually to suitable values by passing a scalar, a list or an array.

- **n_detectors** (int, default None) – The number N_s of (equi-spaced) detector pixels considered. When None, N_s will be chosen as $\sqrt{N_x^2 + N_y^2}$.
- **angle_weights** (None, float, list[float] or numpy.ndarray, default None) – The weights $(\Delta_a)_a$ associated with the angles, which influences the weighting of the rays for the backprojection. See *Adjointness in gratopy* for a more detailed description. If None (by default), the weights are computed automatically based on the **angles** parameter. In the full angle setting, this automatism considers a partition of the half circle for parallel beam and the full circle for fanbeam geometry based on the given angles and sets the angle weight to the average of the distances from of an angle to its two neighbors (in the sense of a circle). Similarly, in the limited angle case, each angle section is partitioned by the angles associated with this section and the weights are chosen taking additionally the boundary of the section into account. In case of a scalar input, this scalar will be used as the (constant) angle weight for all angles. Further, all angle weights can directly be set by passing an input of type list[float] or numpy.ndarray of suitable length.
- **detector_width** (float, default 2.0) – Physical length of the detector. For standard Radon transformation this can usually remain fixed at the default value (together with image_width).
- **image_width** (float, default None) – Physical size of the image indicated by the length of the longer side of the rectangular image domain. For parallel beam geometry, when None, **image_width** is chosen as 2.0. For fanbeam geometry, when None, **image_width** is chosen such that the projections exactly capture the image domain. To illustrate, choosing **image_width** = **detector_width** results in the standard Radon transform with each projection touching the entire object, while **img_width** = 2 **detector_width** results in each projection capturing only half of the image.
- **R** (float, **must be set for fanbeam geometry**) – Physical (orthogonal) distance from source to detector line. Has no impact for parallel beam geometry.
- **RE** (float, **must be set for fanbeam geometry**) – Physical distance from source to origin (center of rotation). Has no impact for parallel beam geometry.
- **detector_shift** (list[float], default 0.0) – Physical shift of all detector pixels along the detector line. Defaults to the application of no shift, i.e., the detector pixels span the range [-**detector_width**/2, **detector_width**/2].
- **midpoint_shift** (list, default [0.0, 0.0]) – Two-dimensional vector representing the shift of the image away from center of rotation. Defaults to the application of no shift, i.e., the center of rotation is also the center of the image.
- **reverse_detector** (bool, default False) – When True, the detector direction is flipped in case of fanbeam geometry, i.e., the positive and negative detector positions are swapped. This parameter has no effect for parallel geometry. When activated together with swapping the sign of the angles, this has the same effect for projection as mirroring the image.

These input parameters create attributes of the same name in an instance of *ProjectionSettings*, though the corresponding values might be slightly restructured by internal processes. Further useful attributes are listed below. It is advised not to set these attributes directly but rather to choose suitable input parameters for the initialization.

Variables

- **is_parallel** (bool) – True if the geometry is for parallel beams, False otherwise.

- **is_fan** (bool) – True if the geometry is for fanbeam geometry, False otherwise.
- **angles** (numpy.ndarray) – Angles from which projections are considered.
- **n_angles** (int) – Number of all angles N_a.
- **sinogram_shape** (tuple (N_s, N_a)) – Represents the number of considered detectors (**n_detectors**) and angles (**n_angles**).
- **delta_x** (float) – Physical width and height δ_x of the image pixels.
- **delta_s** (float) – Physical width δ_s of a detector pixel.
- **delta_ratio** (float) – Ratio δ_s/δ_x, i.e. the detector pixel width relative to unit image pixels.
- **angle_weights** (numpy.ndarray) – Represents the angular discretization width for each angle which are used to weight the projections, see parameter angle_weights above. When none was given as input, the angle_weights chosen by the automatism will be written to this variable.
- **prg** (gratopy.Program) – OpenCL program containing the gratopy OpenCL kernels. For the corresponding code, see *gratopy.create_code*
- **struct** (dict see *radon_struct()* and *fanbeam_struct()* returns) – Data used in the projection operator. Contains in particular dictionaries of numpy.ndarray associated to precision single and double with the angular information necessary for computations.

create_sparse_matrix(*dtype=dtype('float32')*, *order='F'*)

Creates a sparse matrix representation of the associated forward projection.

Parameters

- **dtype** (numpy.dtype, default numpy.float32) – Precision to compute the sparse representation in.
- **order** (str, default F) – Contiguity of the image and sinogram array to the transform, can be F or C.

Returns

Sparse matrix corresponding to the forward projection.

Return type

scipy.sparse.coo_matrix

Note that for high resolution projection operators, this may require infeasibly much time and memory.

show_geometry(*angle*, *figure=None*, *axes=None*, *show=True*)

Visualize the geometry associated with the projection settings. This can be useful in checking that indeed, the correct input for the desired geometry was given.

Parameters

- **angle** (float) – The angle for which the projection is considered.
- **figure** (matplotlib.figure.Figure, default None) – Figure in which to plot. If neither **figure** nor **axes** are given, a new figure (figure(0)) will be created.
- **axes** (matplotlib.axes.Axes, default None) – Axes to plot into. If None, a new axes inside the figure is created.
- **show** (bool, default True) – Determines whether the resulting plot is immediately shown (True). If False, matplotlib.pyplot.show() can be used at a later point to show the figure.

Returns
Figure and axes in which the geometry visualization is plotted.

Return type
tuple(matplotlib.figure.Figure, matplotlib.axes.Axes)

4.2 Transforms

The functions *forwardprojection()* and *backprojection()* perform the projection operations based on the geometry defined in **projectionsetting**. The images **img** and the sinograms **sino** need to be interpreted and behave as described in Getting started.

gratopy.**forwardprojection**(*img*, *projectionsetting*, *sino=None*, *wait_for=[]*)

Performs the forward projection (either for the Radon or the fanbeam transform) of a given image using the given projection settings.

Parameters

- **img** (pyopencl.array.Array with *compatible* dimensions) – The image to be transformed.
- **projectionsetting** (*gratopy.ProjectionSettings*) – The geometry settings for which the forward transform is computed.
- **sino** (pyopencl.array.Array with *compatible* dimensions, default None) – The array in which the result of transformation is saved. If None (per default) is given, a new array will be created and returned.
- **wait_for** (list[pyopencl.Event], default []) – The events to wait for before performing the computation in order to avoid, e.g., race conditions, see pyopencl.Event. This program will always wait for img.events and sino.events (so you need not add them to wait_for).

Returns
The sinogram associated with the projection of the image. If the **sino** is not None, the same pyopencl array is returned with the values in its data overwritten.

Return type
pyopencl.array.Array

The forward projection can be performed for single or double precision arrays. The dtype (precision) of **img** and **sino** (if given) have to coincide and the output will be of the same precision. It respects any combination of *C* and *F* contiguous arrays where output will be of the same contiguity as img if no sino is given. The OpenCL events associated with the transform will be added to the output's events. In case the output array is created, it will use the allocator of **img**. If the image and sinogram have a third dimension (z-direction) the operator is applied slicewise.

gratopy.**backprojection**(*sino*, *projectionsetting*, *img=None*, *wait_for=[]*)

Performs the backprojection (either for the Radon or the fanbeam transform) of a given sinogram using the given projection settings.

Parameters

- **sino** (pyopencl.array.Array with *compatible* dimensions) – Sinogram to be backprojected.
- **projectionsetting** (*gratopy.ProjectionSettings*) – The geometry settings for which the forward transform is computed.

- **img** (pyopencl.array.Array with *compatible* dimensions, default None) – The array in which the result of backprojection is saved. If None is given, a new array will be created and returned.
- **wait_for** (list[pyopencl.Event], default []) – The events to wait for before performing the computation in order to avoid, e.g., race conditions, see pyopencl.Event. This program will always wait for img.events and sino.events (so you need not add them to wait_for).

Returns
The image associated with the backprojected sinogram, coinciding with the **img** if not None, with the values in its data overwritten.

Return type
pyopencl.array.Array

The backprojection can be performed for single or double precision arrays. The dtype (precision) of **img** and **sino** have to coincide. If no **img** is given, the output precision coincides with **sino**'s. The operation respects any combination of *C* and *F* contiguous arrays, where if **img** is None, the result's contiguity coincides with **sino**'s. The OpenCL events associated with the transform will be added to the output's events. In case the output array is created, it will use the allocator of **sino**. If the sinogram and image have a third dimension (z-direction), the operator is applied slicewise.

4.3 Solvers

Based on these forward and backward operators, one can implement a variety of reconstruction algorithms, where the toolbox's focus is on iterative methods (as those in particular are dependent on efficient implementation). The following constitute a few easy-to-use examples which also serve as illustration on how gratopy can be included in custom pyopencl implementations.

gratopy.**landweber**(*sino*, *projectionsetting*, *number_iterations=100*, *w=1*)

Performs a Landweber iteration [L1951] to approximate a solution to the image reconstruction problem associated with a projection and sinogram. This method is also known as SIRT.

Parameters

- **sino** (pyopencl.array.Array) – Sinogram data to reconstruct from.
- **projectionsetting** (*gratopy.ProjectionSettings*) – The geometry settings for which the projection is considered.
- **number_iterations** (int, default 100) – Number of iteration steps to be performed.
- **w** (float, default 1) – Relaxation parameter weighted by the norm of the projection operator (w<1 guarantees convergence).

Returns
Reconstruction from given sinogram gained via Landweber iteration.

Return type
pyopencl.array.Array

gratopy.**conjugate_gradients**(*sino*, *projectionsetting*, *number_iterations=20*, *epsilon=0.0*, *x0=None*)

Performs a conjugate gradients iteration [HS1952] to approximate a solution to the image reconstruction problem associated with a projection and sinogram.

Parameters

- **sino** (pyopencl.array.Array) – Sinogram data to invert.

- **projectionsetting** (*gratopy.ProjectionSettings*) – The geometry settings for which the projection is considered.
- **number_iterations** (float, default 20) – Maximal number of iteration steps to be performed.
- **x0** (pyopencl.array.Array, default None) – Initial guess for iteration (defaults to zeros if None).
- **epsilon** (float, default 0.00) – Tolerance parameter, the iteration stops if relative residual<epsilon.

Returns
Reconstruction gained via conjugate gradients iteration.

Return type
pyopencl.array.Array

gratopy.**total_variation**(*sino*, *projectionsetting*, *mu*, *number_iterations=1000*, *slice_thickness=1.0*, *stepsize_weighting=10.0*)

Performs a primal-dual algorithm [CP2011] to solve a total-variation regularized reconstruction problem associated with a given projection operator and sinogram. This corresponds to the approximate solution of $\min_u \frac{\mu}{2}\|\mathcal{P}u - f\|_{L^2}^2 + \mathrm{TV}(u)$ for $\mathcal{P}$ the projection operator, f the sinogram and μ a positive regularization parameter (i.e., an $L^2 - \mathrm{TV}$ reconstruction approach).

Parameters

- **sino** (pyopencl.array.Array) – Sinogram data to invert.
- **projectionsetting** (*gratopy.ProjectionSettings*) – The geometry settings for which the projection is considered.
- **mu** – Regularization parameter, the smaller the stronger the applied regularization.
- **number_iterations** (float, default 1000) – Number of iterations to be performed.
- **slice_thickness** (float, default 1.0, i.e., isotropic voxels) – When 3-dimensional data sets are considered, regularization is also applied across slices. This parameter represents the ratio of the slice thickness to the length of one pixel within a slice. The choice **slice_thickness** =0 results in no coupling across slices.
- **stepsize_weighting** (float, default 10.0) – Allows to weight the primal-dual algorithm's step sizes σ (stepsize for dual update) and τ (stepsize for primal update) (with $\sigma\tau\|\mathcal{P}\|^2 \leq 1$) by multiplication and division, respectively, with the given value.

Returns
Reconstruction gained via primal-dual iteration for the total-variation regularized reconstruction problem.

Return type
pyopencl.array.Array

gratopy.**normest**(*projectionsetting*, *number_iterations=50*, *dtype='float32'*, *allocator=None*)

Estimate the spectral norm of the projection operator via power iteration, i.e., the operator norm with respect to the norms discussed in *section concerning adjointness*. Useful for iterative methods that require such an estimate, e.g., *landweber()* or *total_variation()*.

Parameters

- **projectionsetting** (*gratopy.ProjectionSettings*) – The geometry settings for which the projection is considered.
- **number_iterations** (int, default 50) – The number of iterations to terminate after.

- **dtype** (numpy.dtype, default numpy.float32) – Precision for which to apply the projection operator (which is not supposed to impact the estimate significantly).

Returns
An estimate of the spectral norm for the projection operator.

Return type
float

gratopy.**weight_sinogram**(*sino*, *projectionsetting*, *sino_out=None*, *divide=False*, *wait_for=[]*)

Performs an angular rescaling of a given sinogram via multiplication (or division) with the projection's angle weights (size of projections in angle dimension, see attributes of *ProjectionSettings*) to the respective projections. This can be useful, e.g., for computing norms or dual pairings in the appropriate Hilbert space.

Parameters

- **sino** (pyopencl.array.Array) – The sinogram whose rescaling is computed. This array itself remains unchanged unless the same array is given as **sino_out**.
- **projectionsetting** (*gratopy.ProjectionSettings*) – The geometry settings for which the rescaling is computed.
- **sino_out** (pyopencl.array.Array default None) – The array in which the result of rescaling is saved. If None (per default) is given, a new array will be created and returned. When giving the same array as **sino**, the values in **sino** will be overwritten.
- **divide** (bool, default False) – Determines whether the sinogram is divided (instead of multiplied) by the angular weights. If True, a division is performed, otherwise, the weights are multiplied.
- **wait_for** (list[pyopencl.Event], default []) – The events to wait for before performing the computation in order to avoid, e.g., race conditions, see pyopencl.Event. This program will always wait for img.events and sino.events (so you need not add them to wait_for).

Returns
The weighted sinogram. If **sino_out** is not None, it is returned with the values in its data overwritten. In particular, giving the same array for sino and sino_out will overwrite this array.

Return type
pyopencl.array.Array

4.4 Data generation

For convenient testing, a phantom generator is included which creates a modified two-dimensional phantom of arbitrary size.

gratopy.**phantom**(*queue*, *N*, *modified=True*, *E=None*, *ret_E=False*, *dtype='double'*, *allocator=None*)

Generate an OpenCL Shepp-Logan phantom of size (N, N).

Parameters

- **queue** (pyopencl.CommandQueue) – The OpenCL command queue.
- **N** (int or array_like) – Matrix size, (N, N) or (M, N).
- **modified** (bool) – Use original gray-scale values as given in [SL1974]. Most implementations use modified values for better contrast (for example, see[2] and[3]).

[2] https://sigpy.readthedocs.io/en/latest/_modules/sigpy/sim.html#shepp_logan
[3] http://www.mathworks.com/matlabcentral/fileexchange/9416-3d-shepp-logan-phantom

- **E** (array_like or None) – $e \times 6$ numeric matrix defining e ellipses. The six columns of **E** are:
  - Gray value of the ellipse (in [0, 1])
  - Length of the horizontal semi-axis of the ellipse
  - Length of the vertical semi-axis of the ellipse
  - x-coordinate of the center of the ellipse (in [-1, 1])
  - y-coordinate of the center of the ellipse (in [-1, 1])
  - Angle between the horizontal semi-axis of the ellipse and the x-axis of the image (in rad)
- **ret_E** (bool) – Return the matrix **E** used to generate the phantom.
- **dtype** (str or numpy.dtype) – The pyopencl data type in which the phantom is created.
- **allocator** (An implementation of pyopencl.tools.AllocatorInterface or None) – The pyopencl allocator used for memory allocation.

Returns
Phantom/parameter pair (**ph** [, **E**]).

Variables

- **ph** (pyopencl.array.Array) – The Shepp-Logan phantom.
- **E** (array_like, optional) – The ellipse parameters used to generate **ph**.

This much abused phantom is due to [SL1974]. The tabulated values in the paper are reproduced in the Wikipedia entry[1]. The original values do not produce great contrast, so modified values are used by default (see Table B.1 in [TS1996] or implementations[Page 23, 2] and[Page 23, 3]).

4.5 Internal functions

The following contains the documentation for a set of internal functions which could be of interest for developers. Note that these might be subject to change in the future.

gratopy.**radon**(*sino*, *img*, *projectionsetting*, *wait_for=[]*)

Performs the Radon transform of a given image using the given **projectionsetting**.

Parameters

- **sino** (pyopencl.array.Array) – The array in which the resulting sinogram is written.
- **img** (pyopencl.array.Array) – The image to transform.
- **projectionsetting** (*gratopy.ProjectionSettings*) – The geometry settings for which the Radon transform is performed.
- **wait_for** (list[pyopencl.Event], default []) – The events to wait for before performing the computation in order to avoid, e.g., race conditions, see pyopencl.Event. This program will always wait for img.events and sino.events (so you need not add them to wait_for).

Returns
Event associated with the computation of the Radon transform (which is also added to the events of **sino**).

Return type
pyopencl.Event

[1] https://en.wikipedia.org/wiki/Shepp%E2%80%93Logan_phantom

gratopy.**radon_ad**(*img*, *sino*, *projectionsetting*, *wait_for=[]*)

Performs the Radon backprojection of a given sinogram using the given **projectionsetting**.

Parameters

- **img** (pyopencl.array.Array) – The array in which the resulting backprojection is written.
- **sino** (pyopencl.array.Array) – The sinogram to transform.
- **projectionsetting** (*gratopy.ProjectionSettings*) – The geometry settings for which the Radon backprojection is performed.
- **wait_for** (list[pyopencl.Event], default []) – The events to wait for before performing the computation in order to avoid, e.g., race conditions, see pyopencl.Event. This program will always wait for img.events and sino.events (so you need not add them to wait_for).

Returns

Event associated with the computation of the Radon backprojection (which is also added to the events of **img**).

Return type

pyopencl.Event

gratopy.**radon_struct**(*queue*, *img_shape*, *angles*, *angle_weights*, *n_detectors=None*, *detector_width=2.0*, *image_width=2.0*, *midpoint_shift=[0, 0]*, *detector_shift=0.0*)

Creates the structure storing geometry information required for the Radon transform and its adjoint.

Parameters

- **queue** (pyopencl.CommandQueue) – OpenCL command queue in which context the computations are to be performed.
- **img_shape** (tuple (N_x, N_y)) – The number of pixels of the image in x- and y-direction respectively, i.e., the image size. It is assumed that by default, the center of rotation is in the middle of the grid of quadratic pixels. The midpoint can, however, be shifted, see the **midpoint_shift** parameter.
- **angles** (numpy.ndarray) – Determines which angles are considered for the projection.
- **angle_weights** (numpy.ndarray) – The weights associated to the angles, e.g., how much of the angular range is covered by this angle. This impacts the weighting of rays for the backprojection.
- **n_detectors** (int, default None) – The number N_s of considered (equi-spaced) detectors. If None, N_s will be chosen as $\sqrt{N_x^2 + N_y^2}$.
- **detector_width** (float, default 2.0) – Physical length of the detector line.
- **image_width** (float, default 2.0) – Physical size of the image indicated by the length of the longer side of the rectangular image domain. Choosing **image_width** = **detector_width** results in the standard Radon transform with each projection touching the entire object, while **img_width** = 2 **detector_width** results in each projection capturing only half of the image.
- **midpoint_shift** (list[float], default [0.0, 0.0]) – Two-dimensional vector representing the shift of the image away from center of rotation. Defaults to the application of no shift.
- **detector_shift** (list[float], default 0.0) – Physical shift of the detector along the detector line in detector pixel offsets. Defaults to the application of no shift, i.e., the detector reaches from [- **detector_width**/2, **detector_width**/2].

Returns
Struct dictionary with the following variables as entries, where the keys are strings of the same names:

Return type
dict

Variables

- **ofs_dict** (dict{numpy.dtype: numpy.ndarray}) – Dictionary containing the relevant angular information as numpy.ndarray for the data types numpy.float32 and numpy.float64. The arrays have dimension $(8, N_a)$ with columns:

0	weighted cosine
1	weighted sine
2	detector offset
3	inverse of cosine/sine
4	angular weight
5	flipped

  The remaining columns are unused. The value **flipped** indicates whether the x and y axis are flipped (1) or not (0), which is done for reasons of numerical stability. The 4th entry contains the inverse of sine if the axes are flipped and the inverse of cosine otherwise.

- **shape** (tuple) – Tuple of integers (N_x, N_y) representing the size of the image.
- **sinogram_shape** (tuple) – Tuple of integers (N_s, N_a) representing the size of the sinogram.
- **geo_dict** (dict{numpy.dtype: numpy.ndarray}) – Dictionary mapping the allowed data types to an array containing the values $[\delta_x, \delta_s, N_x, N_y, N_s, N_a]$.
- **angles_diff_buf** – Dictionary containing the same values as in **ofs_dict** [4] representing the weights associated with the angles (i.e., the length of sinogram pixels in the angular direction).

gratopy.**fanbeam**(*sino*, *img*, *projectionsetting*, *wait_for=[]*)

Performs the fanbeam transform of a given image using the given **projectionsetting**.

Parameters

- **sino** (pyopencl.array.Array) – The array in which the resulting sinogram is written.
- **img** (pyopencl.array.Array) – The image to transform.
- **projectionsetting** (*gratopy.ProjectionSettings*) – The geometry settings for which the fanbeam transform is performed.
- **wait_for** (list[pyopencl.Event], default []) – The events to wait for before performing the computation in order to avoid, e.g., race conditions, see pyopencl.Event. This program will always wait for img.events and sino.events (so you need not add them to wait_for).

Returns
Event associated with the computation of the fanbeam transform (which is also added to the events of **sino**).

Return type
pyopencl.Event

gratopy.**fanbeam_ad**(*img*, *sino*, *projectionsetting*, *wait_for=[]*)

Performs the fanbeam backprojection of a given sinogram using the given **projectionsetting**.

Parameters

- **img** (pyopencl.array.Array) – The array in which the resulting backprojection is written.
- **sino** (pyopencl.array.Array) – The sinogram to transform.
- **projectionsetting** (*gratopy.ProjectionSettings*) – The geometry settings for which the fanbeam backprojection is performed.
- **wait_for** (list[pyopencl.Event], default []) – The events to wait for before performing the computation in order to avoid, e.g., race conditions, see pyopencl.Event. This program will always wait for img.events and sino.events (so you need not add them to wait_for).

Returns

Event associated with the computation of the fanbeam backprojection (which is also added to the events of **img**).

Return type

pyopencl.Event

gratopy.**fanbeam_struct**(*queue*, *img_shape*, *angles*, *detector_width*, *source_detector_dist*, *source_origin_dist*, *angle_weights*, *n_detectors=None*, *detector_shift=0.0*, *image_width=None*, *midpoint_shift=[0, 0]*, *reverse_detector=False*)

Creates the structure storing geometry information required for the fanbeam transform and its adjoint.

Parameters

- **queue** (pyopencl.CommandQueue) – OpenCL command queue in which context the computations are to be performed.
- **img_shape** (tuple (N_x, N_y)) – The number of pixels of the image in x- and y-direction respectively, i.e., the image size. It is assumed that by default, the center of rotation is in the middle of the grid of quadratic pixels. The midpoint can, however, be shifted, see the **midpoint_shift** parameter.
- **angles** (numpy.ndarray) – Determines which angles are considered for the projection.
- **detector_width** (float, default 2.0) – Physical length of the detector line.
- **source_detector_dist** (float) – Physical (orthogonal) distance **R** from the source to the detector line.
- **source_origin_dist** (float) – Physical distance **RE** from the source to the origin (center of rotation).
- **angle_weights** (numpy.ndarray) – The weights associated to the angles, e.g., how much of the angular range is covered by this angle. This impacts the weighting of rays for the backprojection.
- **n_detectors** (int or None, default None) – The number N_s of considered (equi-spaced) detectors. If None, N_s will be chosen as $\sqrt{N_x^2 + N_y^2}$.
- **detector_shift** (list[float], default 0.0) – Physical shift of the detector along the detector line in detector pixel offsets. Defaults to the application of no shift, i.e., the detector reaches from [- **detector_width**/2, **detector_width**/2].
- **image_width** (float, default None) – Physical size of the image indicated by the length of the longer side of the rectangular image domain. If None, **image_width** is chosen to capture just all rays.

- **midpoint_shift** (list[float], default [0.0, 0.0]) – Two-dimensional vector representing the shift of the image away from center of rotation. Defaults to the application of no shift.
- **reverse_detector** (bool, default False) – When True, the detector direction is flipped.

Returns
Struct dictionary with the following variables as entries, where the keys are strings of the same names:

Return type
dict

Variables

- **img_shape** (tuple) – Tuple of integers (N_x, N_y) representing the size of the image.
- **sinogram_shape** (tuple) – Tuple of integers (N_s, N_a) representing the size of the sinogram.
- **ofs_dict** (dict{numpy.dtype: numpy.ndarray}) – Dictionary containing the relevant angular information as numpy.ndarray for the data types numpy.float32 and numpy.float64. The arrays have dimension $(8, N_a)$ with columns:

0 1	vector of length δ_s pointing in positive detector direction
2 3	vector connecting source and center of rotation
4 5	vector connection the origin and its projection onto the detector line
6	angular weight

  The remaining column is unused.
- **sdpd_dict** (dict{numpy.dtype: numpy.ndarray}) – Dictionary mapping numpy.float32 and numpy.float64 to a numpy.ndarray representing the values $\sqrt{(s^2 + R^2)}$ for the weighting in the fanbeam transform (weighted by **delta_ratio**, i.e., δ_s/δ_x).
- **image_width** – Physical size of the image. Equal to the input parameter if given, or to the determined image size if **image_width** is None (see parameter **image_width**).
- **geo_dict** (dict{numpy.dtype: numpy.ndarray}) – Dictionary mapping the allowed data types to an array containing the values [source detector distance/δ_x, source origin distance/δ_x, width of a detector_pixel relative to width of image_pixels i.e. δ_s/δ_x, image midpoint x-coordinate (in pixels), image midpoint y-coordinate (in pixels), detector line midpoint (in detector-pixels), N_x, N_y, N_s, N_a, width of an image pixel (δ_x)].
- **angles_diff** (dict{numpy.dtype: numpy.ndarray}) – Dictionary containing the same values as in **ofs_dict** [6] representing the weights associated with the angles (i.e., the length of sinogram pixels in the angular direction).

gratopy.**create_code**()

Reads and creates CL code containing all OpenCL kernels of the gratopy toolbox.

Returns
The toolbox's CL code.

Return type
str

CHAPTER

FIVE

ACKNOWLEDGEMENTS

5.1 Authors, publications and funding

5.1.1 Authors

- **Kristian Bredies**, University of Graz, kristian.bredies@uni-graz.at
- **Richard Huber**, University of Graz, richard.huber@uni-graz.at

5.1.2 Publications

If you find the toolbox useful, please cite the following associated publications.

- Kristian Bredies and Richard Huber. (2021). *Convergence analysis of pixel-driven Radon and fanbeam transforms.* SIAM Journal on Numerical Analysis 59(3), 1399–1432. https://doi.org/10.1137/20M1326635.
- Kristian Bredies and Richard Huber. (2021). *Gratopy 0.1* [Software]. Zenodo. https://doi.org/10.5281/zenodo.5221442

5.1.3 Funding

The development of this software was supported by the following projects:

- **Regularization Graphs for Variational Imaging**, funded by the Austrian Science Fund (FWF), grant P-29192,
- **International Research Training Group IGDK 1754 Optimization and Numerical Analysis for Partial Differential Equations with Nonsmooth Structures**, funded by the German Research Council (DFG) and the Austrian Science Fund (FWF), grant W-1244.

5.2 Used data sets and code

The walnut data set included in this toolbox is licensed under CC BY 4.0 and available on Zenodo:

- Keijo Hämäläinen, Lauri Harhanen, Aki Kallonen, Antti Kujanpää, Esa Niemi and Samuli Siltanen. (2015). *Tomographic X-ray data of a walnut* (Version 1.0.0) [Data set]. Zenodo. http://doi.org/10.5281/zenodo.1254206

The phantom creation code is based on Phantominator, copyright by its contributors and licensed under GPLv3. See https://github.com/mckib2/phantominator.

5.3 License

GNU GENERAL PUBLIC LICENSE Version 3

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

BIBLIOGRAPHY

[HHKKNS2015] Keijo Hämäläinen and Lauri Harhanen and Aki Kallonen and Antti Kujanpää and Esa Niemi and Samuli Siltanen. "Tomographic X-ray data of a walnut". https://arxiv.org/abs/1502.04064

[L1951] Landweber, L. "An iteration formula for Fredholm integral equations of the first kind." Amer. J. Math. 73, 615–624 (1951). https://doi.org/10.2307/2372313

[HS1952] Hestenes, M. R., Stiefel, E. "Methods of Conjugate Gradients for Solving Linear Systems." Journal of Research of the National Bureau of Standards, 49:409–436 (1952). https://doi.org/10.6028/jres.049.044

[CP2011] Chambolle, A., Pock, T. "A First-Order Primal-Dual Algorithm for Convex Problems with Applications to Imaging." J Math Imaging Vis 40, 120–145 (2011). https://doi.org/10.1007/s10851-010-0251-1

[SL1974] Shepp, Lawrence A., and Benjamin F. Logan. "The Fourier reconstruction of a head section." IEEE Transactions on nuclear science 21.3 (1974): 21-43.

[TS1996] Toft, Peter Aundal, and John Aasted Sørensen. "The Radon transform-theory and implementation." (1996).

PYTHON MODULE INDEX

INDEX

W