5 Tomosipo: Flexible tomography in Python

“The difference between right and wrong often lies in less than five meters.”

“Het verschil tussen goed en fout ligt vaak in niet meer dan vijf meter.”

Johan Cruijff,
Vrij Nederland, 21 Dec 1974

Tomographic imaging enables the examination of the internal structure of an object. The object is typically placed between a source and detector, and its structure is reconstructed using projection images from a range of different positions. Collectively, the position information of the source, object, and detector determine the acquisition geometry.

Most common tomographic techniques rely on a selection of standard acquisition geometries, such as circular cone beam or single-axis parallel beam [31]. In recent years, several scientific and industrial applications have emerged whose needs are not met by the standard selection of paths. Such scientific applications include diffraction contrast tomography (DCT) [184] and X-ray scattering tensor tomography (XSTT) [93]. These techniques measure X-ray effects other than absorption, which necessarily give rise to more complex acquisition geometries. Complex geometries also arise in industrial applications like automotive and aerospace testing [52,

This chapter is based on:
A. A. Hendriksen, D. Schut, W. J. Palenstijn, N. Viganó, D. M. Kim Jisoo Pelt, T. van Leeuwen, and K. J. Batenburg. “Tomosipo: Fast, Flexible, and Convenient 3D Tomography for Complex Scanning Geometries in Python”. *Optics Express* (2021).

101], as objects may be too large to fit in conventional scanners. Instead, a robot arm moves the source and detector along an irregular path around the object.

Efficient reconstruction algorithms exist for many common acquisition geometries [31, 89]. Such filtered backprojection (FBP)-type algorithms are typically fast to compute [138], but require the source and detector to follow a regular path. Algorithms that permit flexible acquisition geometries, such as SIRT [60] and total variation minimization (TV-MIN) [168], typically follow an iterative reconstruction scheme. As iterative algorithms tend to be more computationally demanding than FBP-type algorithms, they benefit more from an efficient implementation.

Software packages for computing reconstructions can be roughly subdivided by their target audience. For application scientists in electron tomography [119] and synchrotron tomography [64, 130, 186, 188], software exists that provides pre-processing and reconstruction capabilities. For scientists developing new reconstruction algorithms, packages exist that integrate tomography in optimization methods [150, 193] and neural networks [175], or implement tomographic primitives on the graphics processing unit (GPU), such as the TIGRE and ASTRA Toolbox [1, 2, 20].

Existing tomography software is typically limited in its ability to represent, create, visualize, and reconstruct using complex acquisition geometries. Software for application scientists usually includes optimized reconstruction routines for a selection of acquisition geometries, but generally does not provide the flexibility to represent arbitrary acquisition geometries. Some software packages providing tomographic primitives, like the ASTRA Toolbox, can represent arbitrarily complex acquisition geometries, but do not provide effective tools to create them. In fact, the positions and orientations of the object and detector are usually computed using trigonometric formulas, requiring tedious and error-prone handwork [1]. In addition, limited facilities are included to visualize geometries, making it difficult to validate the computed geometry. Therefore, defining unconventional acquisition geometries requires extraordinary attentiveness. The lack of validation capabilities can also be problematic when processing data from advanced experiments, as it can be difficult or impossible to determine whether certain reconstruction artifacts are caused by an incorrect modeling of the acquisition geometry, or are due to other common sources of artifacts (e.g., sample motion, beam stability, etc). This may lead to sub-optimal reconstruction results and could prohibit further analysis of the data.

In this chapter, we introduce the `tomosipo` Python package[1], which is designed to alleviate the problems in defining complex acquisition geometries for tomography. Specifically, the package provides convenient primitives for the representation, creation, visualization, and reconstruction of complex acquisition geometries, as described below.

Representation and creation. Tomosipo allows the user to assemble increasingly complex acquisition geometries by composing geometric transforms and applying them to primitive acquisition geometries. Several standard geometric

[1]Tomósipo is pronounced with the stress on the second syllable.

transformations can be defined, such as rotation, translation, scaling, and reflection. Tomosipo's representation of the acquisition geometries is flexible. Therefore, the result of applying a geometric transform, e.g., rotation, to an acquisition geometry can be represented in tomosipo. In addition to flexible geometries, tomosipo provides convenience methods to create standard acquisition geometries, such as circular cone beam and single-axis parallel beam geometries.

Visualization. To aid in validation and communication, visualization of the resulting geometry is crucial. With tomosipo, the defined geometry can be viewed in a 3D environment or a Jupyter notebook [145], and saved to disk as a video or scalable vector graphic (SVG).

Reconstruction. Tomosipo provides a concise and efficient application programming interface (API) for computing reconstructions. Its design is similar to Matlab's Spot operators [21] and the computations are powered by the ASTRA Toolbox. In addition, tomosipo integrates with several packages for GPU computing, such as PyTorch [139] and CuPy, enabling the user to implement reconstruction algorithms without moving intermediate results to and from the GPU, yielding immediate speed benefits. These speed benefits are observed both in iterative and FBP-type reconstruction methods, as implemented in the separate `ts_algorithms` package[2].

This chapter provides an overview of the design of tomosipo and case studies of possible applications. First, the tomography problem is introduced in Section 5.1. In Section 5.2, key concepts of the package are described. In Section 5.3, these concepts are demonstrated on two simple absorption contrast tomography examples and two complex acquisition schemes exploiting X-ray diffraction and scattering. In Section 5.4, reconstructions are shown of experimental data using several algorithms. In Section 5.5, the use of tomosipo on the GPU is demonstrated and its speed is compared to existing reconstruction algorithms in the ASTRA Toolbox. We conclude with a discussion in Section 5.6.

5.1 Standard tomography problem

Common tomography setups expose a sample to a beam of high energy particles, e.g., photons, electrons, or neutrons, which are collected on a detector. Contrast in the measured projection images is generated by differences in attenuation, refraction index or scattering of the object (e.g. phase and diffraction, respectively), or the emission of secondary signals (e.g. X-ray fluorescence, Compton, Auger). Many of these problems can be modeled as a collection of line integrals through space where the ith measurement $y_i \in \mathbb{R}$ is obtained as a line integral

$$y_i = \int_{\mathbb{R}} x(\mathbf{s}_i + t\boldsymbol{\eta}_i)\ \mathrm{d}t \tag{5.1}$$

through a point $\mathbf{s}_i \in \mathbb{R}^3$ with direction $\boldsymbol{\eta}_i \in \mathbb{R}^3$. The canonical case is absorption contrast tomography, which we describe here.

[2]`https://github.com/ahendriksen/ts_algorithms`

In absorption contrast tomography, the reconstruction problem can be posed as a linear discrete inverse problem. Suppose measurements $\mathbf{y} \in \mathbb{R}^{N_\theta \times N_p^2}$ are acquired from N_θ positions using a square detector that is divided into N_p^2 pixels. Define the cubic reconstruction volume $\mathbf{x} \in \mathbb{R}^{N_v^3}$ on a voxel grid and let $\mathbf{A}$ denote the projection matrix such that $\mathbf{A}_{ij}$ describes the absorption by object voxel j of the ray to measurement i. The goal is to determine the value of $\mathbf{x}$ that gave rise to the measurement

$$\mathbf{A}\mathbf{x} = \mathbf{y}. \tag{5.2}$$

The computation of the linear operator $\mathbf{A}$ depends strongly on the geometry of the acquisition. This includes the direction of the rays, the position and orientation of the reconstruction volume, and the position and orientation of the detector.

5.2 Framework concepts

Three concepts are essential to the tomosipo package. These are geometries, geometric transformations, and the projection operator $\mathbf{A}$. Geometries represent the position of the source, sample, and detector at each time step. The sample's position and orientation is represented by a volume geometry, and the X-ray source and flat panel detector are represented by a projection geometry, which can model both point sources (cone beam geometry) and parallel box beams (parallel beam geometry). All geometries have two representations: a simple representation that defines a standard trajectory, and a flexible representation that permits arbitrary movement and orientation. Volume and projection geometries are discussed in Section 5.2.2.

Geometries can be manipulated using geometric transforms, as well as split and joined using subsampling and concatenation. In this way, complicated acquisition geometries can be assembled from simple geometries. This is described in Sections 5.2.3 and 5.2.5.

Together, a volume and projection geometry define the projection operator $\mathbf{A}$. In tomosipo, the computation using $\mathbf{A}$ is GPU-accelerated using the ASTRA Toolbox. Most tomosipo geometries have an ASTRA counterpart, except for the flexible volume geometry whose movement and orientation is compensated for by exploiting the flexibility of ASTRA's projection geometries. The creation of projection operators is discussed in the next section, and the integration with the ASTRA Toolbox and Python array libraries in Section 5.2.4. The main concepts of tomosipo and their relation to the ASTRA toolbox and the physical geometry are summarized in Figure 5.1.

5.2.1 Tomographic projection

In this section, we describe the creation and use of the projection operator $\mathbf{A}$. Tomosipo provides a convenient representation of the projection operator $\mathbf{A}$, offering

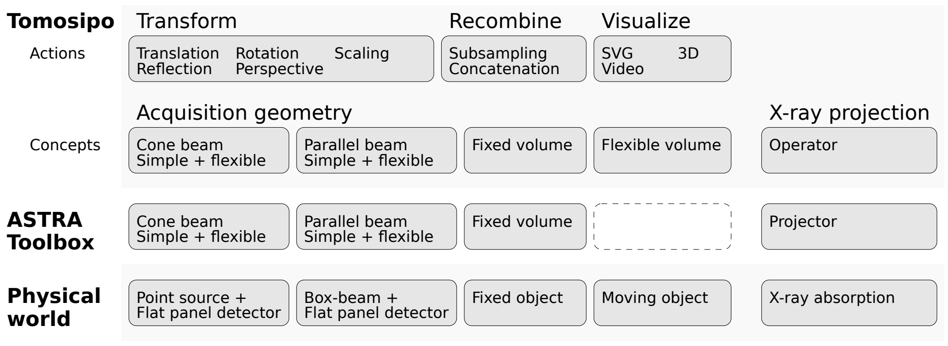


Figure 5.1: The relation between tomosipo, the ASTRA Toolbox, and the physical world. Tomosipo can be roughly divided in actions and concepts. The concepts describe the acquisition geometry and the X-ray projection and can be directly mapped onto ASTRA primitives, except for the flexible (moving) volume, which has no ASTRA counterpart. The actions provide the means to transform, recombine, and visualize tomosipo's geometry primitives.

an API that is similar to the opTomo Spot operator in the ASTRA Toolbox [21]. Given a volume geometry `vg` and projection geometry `pg`, the linear operator **A** from Equation (5.2) can be obtained as follows:

```python
import tomosipo as ts
vg = ts.volume([...])            # Argument details are described
pg = ts.parallel([...])          # in next section
A = ts.operator(vg, pg)
```

The operator `A` is a stand-alone object. It has `domain_shape` and `range_shape` properties that facilitate the creation of data of the right shape in its mathematical domain and range, i.e., image space and sinogram space.

```python
x = np.ones(A.domain_shape, dtype=np.float32)
```

It can be applied to data as follows:

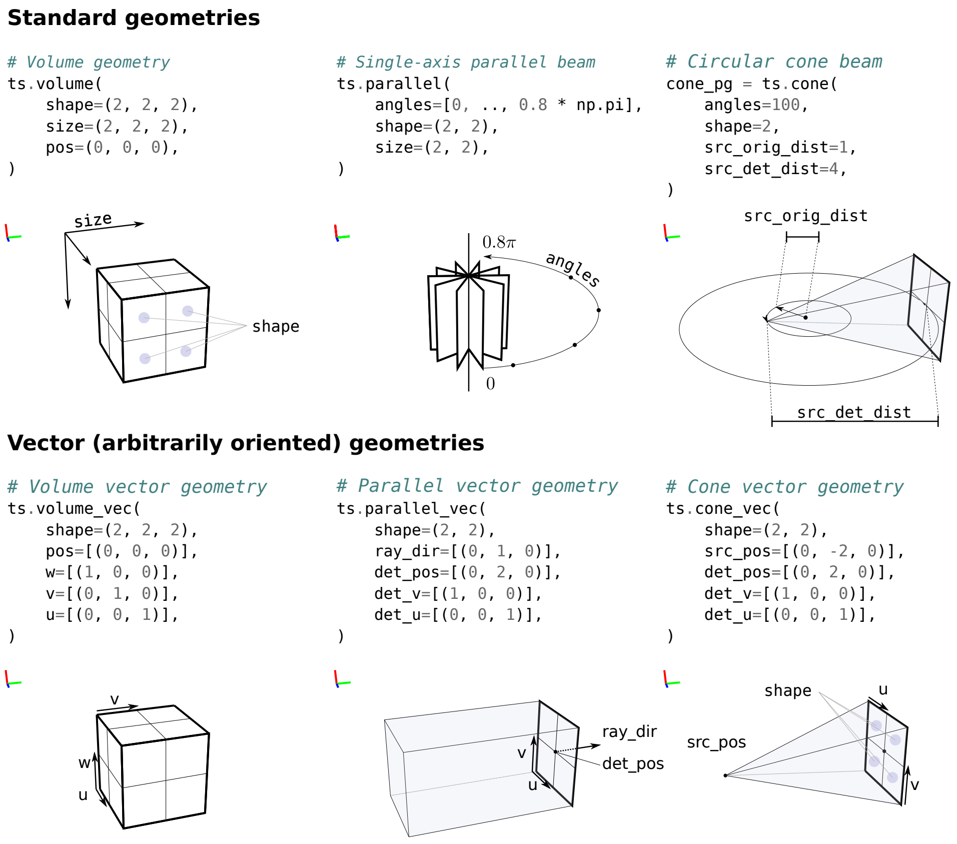
```python
y = A(x)
backprojection = A.T(y)
```

The computation is performed on the GPU, and is handled by the ASTRA Toolbox.

The operator `A` can be used to solve the inverse problem posed in Equation 5.2. In the code below, this is demonstrated by computing a simple Landweber iteration [105] with step size `eta`.

```python
x_rec = np.zeros(A.domain_shape, np.float32)
for i in range(num_iterations):
    x_rec = x_rec + eta * A.T(y - A(x_rec))
```

In the next section, we describe how to define the volume and projection geometries that are required to create a projection operator.



Standard geometries

```python
# Volume geometry
ts.volume(
    shape=(2, 2, 2),
    size=(2, 2, 2),
    pos=(0, 0, 0),
)
```

```python
# Single-axis parallel beam
ts.parallel(
    angles=[0, .., 0.8 * np.pi],
    shape=(2, 2),
    size=(2, 2),
)
```

```python
# Circular cone beam
cone_pg = ts.cone(
    angles=100,
    shape=2,
    src_orig_dist=1,
    src_det_dist=4,
)
```

Vector (arbitrarily oriented) geometries

```python
# Volume vector geometry
ts.volume_vec(
    shape=(2, 2, 2),
    pos=[(0, 0, 0)],
    w=[(1, 0, 0)],
    v=[(0, 1, 0)],
    u=[(0, 0, 1)],
)
```

```python
# Parallel vector geometry
ts.parallel_vec(
    shape=(2, 2),
    ray_dir=[(0, 1, 0)],
    det_pos=[(0, 2, 0)],
    det_v=[(1, 0, 0)],
    det_u=[(0, 0, 1)],
)
```

```python
# Cone vector geometry
ts.cone_vec(
    shape=(2, 2),
    src_pos=[(0, -2, 0)],
    det_pos=[(0, 2, 0)],
    det_v=[(1, 0, 0)],
    det_u=[(0, 0, 1)],
)
```

Figure 5.2: Creation of typical tomographic geometries. From left to right: a volume geometry, single-axis parallel beam geometry, and a circular cone beam geometry. Below, arbitrarily oriented vector geometries are shown. The parameters are specified using keyword-only arguments [179]. The `pos` parameter, for instance, determines the position of a volume, other parameters have accompanying labels in the diagrams.

5.2.2 Acquisition geometry primitives

Tomosipo provides three standard geometries: the fixed volume geometry, the single-axis parallel beam geometry, and the circular cone beam geometry. In addition, these geometries have a flexible counterpart that permits arbitrary orientation and movement. The flexible geometries are known as vector geometries, following the terminology of [1]. All geometry primitives are defined in the ASTRA Toolbox as well, except for the volume vector geometry that can represent an arbitrarily oriented moving reconstruction grid. The geometries are illustrated in Figure 5.2 with accompanying code.

In contrast to the standard projection geometries, whose movement is parameterized by the rotation angle, vector geometries move arbitrarily in time. We therefore refer to the state of the acquisition geometry at a specific time as a time step. Furthermore, geometries have a `num_steps` property that describes in how many time steps their movement is discretized.

Standard geometries

Volume geometry. A volume geometry describes the position and size of an axis-aligned voxel grid on which the object is reconstructed. A volume geometry can be created with `size`, `pos`, and `shape` parameters, which define its physical size, center position, and the number of voxels in each direction. By default, the volume is centered on the origin, and if the size is not specified, it is set to equal the shape, causing the voxel size to equal 1. Other parametrizations, such as in terms of the volume's extents, are described in the documentation.

Single-axis parallel beam. In the parallel beam geometry, X-rays run along parallel lines and are collected on a flat panel detector that rotates around a single axis on the origin. It can be created with `size`, `shape`, and `angles` parameters, which define the detector's physical size, the number of pixels in each dimension, and the rotation angles. If an integer argument is provided for `angles`, equispaced rotation angles in the interval $[0, \pi)$ are used. Otherwise, a provided array is interpreted as containing the rotation angles in radians.

Circular cone beam. Like the parallel beam geometry, the flat panel detector of a cone beam geometry rotates around an axis located on the origin and the `angles`, `shape`, and `size` parameters behave similarly. In contrast to the parallel beam geometry, the rays in a cone beam geometry are emitted from a point source, and the source-to-origin distance and source-to-detector distances can be specified using the `src_orig_dist` and `src_det_dist` parameters. Also, when `angles` is provided as an integer, a rotation is performed along a full arc $[0, 2\pi)$ as opposed to $[0, \pi)$.

Flexible vector geometries

Any geometry `g` can be converted to a vector geometry by calling `g.to_vec()`. Vector geometries can also be created directly as described below.

Volume vector geometry. In contrast to a volume geometry, which is static, a volume vector geometry may move over time and the reconstruction grid may be arbitrarily oriented. It can be created by providing the shape of the voxel grid and 3 vectors describing the local frame of reference of the grid at each point in time. In practice, a vector volume geometry is easier to obtain by applying a geometric transformation to a standard volume geometry.

The ASTRA Toolbox, tomosipo's computational back end, does not support non-axis-aligned volume geometries. Internally, tomosipo aligns the volume to the origin and moves the projection geometry with it. The transformed geometries are handed to ASTRA, causing the projection operation to be performed in the frame of reference of the object.

Arbitrarily oriented parallel beam. In a parallel vector geometry, the detector can be arbitrarily oriented and positioned. In addition, the direction of the incoming rays can be adjusted to a direction that is not necessarily orthogonal to the detector plane. It can be created by specifying a fixed detector shape and varying ray directions, detector positions, and detector orientations at each time step. The orientation is determined by parameters `det_u` and `det_v` that specify the vector from detector pixel $(0, 0)$ to $(0, 1)$ and $(0, 0)$ to $(1, 0)$, respectively. An example is the dual-axis parallel beam geometry, which is common in electron tomography [128].

Arbitrarily oriented cone beam. In a cone vector geometry, the detector can be arbitrarily oriented and the source can be placed in an arbitrary location. For instance, this geometry can represent a helical cone beam acquisition, as we show in Section 5.3.1. It can be created like the parallel vector geometry: instead of a ray direction, however, a source position must be provided for each time step. In the next section, we describe in more detail how vector geometries can be obtained as transformations of simple geometries.

5.2.3 Geometric transforms

Tomosipo defines geometric transforms that can rotate, translate, scale, and reflect the previously introduced geometries. In addition, the package provides a perspective transform to switch between different frames of reference. The transforms are stand-alone objects instead of functions that act on geometries directly. We first discuss the internal representation of the transforms and then we introduce the built-in functions to create transforms.

Representation. Internally, homogeneous coordinates [154] are used so that a 4×4 matrix $\mathbf{M}$ describes a single time step of a transformation. An orientation vector $\mathbf{v} = (v_1, v_2, v_3)$ is represented in homogeneous coordinates by $\mathbf{v} = (v_1, v_2, v_3, 0)$, whereas a position $\mathbf{p} = (p_1, p_2, p_3)$ is represented by $\mathbf{p} = (p_1, p_2, p_3, 1)$. This way, application of a geometric transform — notably translation — to points and vectors can be performed by matrix multiplication. That is, in homogeneous coordinates, the transformed vector equals $\mathbf{Mv}$ and the transformed point equals $\mathbf{Mp}$. In code, a vector and point in Euclidean coordinates are transformed as follows

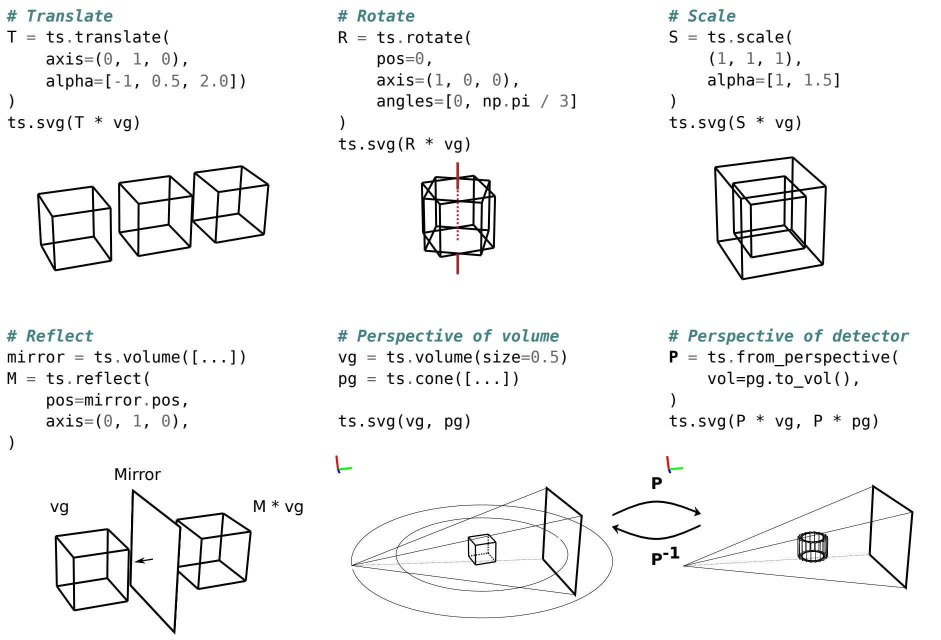
```
transformed_v = T.transform_vec(v)
transformed_p = T.transform_point(p)
```

Application of a transform to a geometry is expressed in code as

```
transformed_vg = T * vg
```

In the internal representation, the composition of two transforms is also computed by matrix multiplication. The matrix representation of the composition $T = T_1 \circ T_2$ of two transforms T_1, T_2 represented by matrices $\mathbf{M}_1, \mathbf{M}_2$ is equal to the matrix product of the matrices, i.e., $\mathbf{M} = \mathbf{M}_1\mathbf{M}_2$. In code, this is expressed as

```
T = T1 * T2
```



```
# Translate
T = ts.translate(
    axis=(0, 1, 0),
    alpha=[-1, 0.5, 2.0])
)
ts.svg(T * vg)
```

```
# Rotate
R = ts.rotate(
    pos=0,
    axis=(1, 0, 0),
    angles=[0, np.pi / 3]
)
ts.svg(R * vg)
```

```
# Scale
S = ts.scale(
    (1, 1, 1),
    alpha=[1, 1.5]
)
ts.svg(S * vg)
```

```
# Reflect
mirror = ts.volume([...])
M = ts.reflect(
    pos=mirror.pos,
    axis=(0, 1, 0),
)
```

```
# Perspective of volume
vg = ts.volume(size=0.5)
pg = ts.cone([...])

ts.svg(vg, pg)
```

```
# Perspective of detector
P = ts.from_perspective(
    vol=pg.to_vol(),
)
ts.svg(P * vg, P * pg)
```

Figure 5.3: Overview of geometric transforms in tomosipo. From left to right, translation, rotation, scaling, and reflection. In the two panes in the bottom right, a typical cone beam acquisition is shown from two perspectives: a static volume with the source and detector rotating around it and a static source and detector with a volume rotating in between. A perspective transform $\boldsymbol{P}$ allows switching between the two frames of reference. The vector illustrations are created using the `ts.svg()` function.

Composition of transforms is demonstrated in Section 5.3.1, where a helical cone beam geometry is created.

Rigid and scaling transforms. Tomosipo provides functions to create a translation, rotation, scaling, or reflection transform. These are illustrated in Figure 5.3. A transform may change over time, i.e., at each time step it can define a different geometric transformation. The functions that create the transforms are designed to facilitate defining transforms that vary over time.

A translation transform is parameterized by an `axis` and an array `alpha`. The displacement vector at time step `i` is defined by `alpha[i] * axis`.

A rotation transform is created using the axis angle representation. The `axis`, `pos`, and `angles` parameters describe the orientation and location of the rotation axis, as well as the angle of rotation. Each of these parameters may be provided as an array to define the rotation at multiple time steps. The angles are expressed in radians and the direction of rotation is right-handed by default.

A scaling transform describes a scaling operation centered on a position. The scaling is not necessarily isotropic: some directions can be scaled more than others. An `alpha` parameter can be used to modulate the scaling at each time step.

A reflection transform describes a reflection in a plane that is parameterized by a position `pos` and a normal vector `axis`. Both can be specified as an array, defining a reflection in a moving plane at several time steps.

Perspective. The `ts.from_perspective` function creates a perspective transform. This function takes a volume and returns the transform that moves the volume back to the origin and rotates it back into a single axis-aligned orientation. All projection geometries have a `to_vol()` method that describes the frame of reference of the detector at each time step. This makes it easy to create a transform that converts to the detector's frame of reference. In the case of a circular cone beam trajectory, for example, the source and detector rotate around the volume, from the volume's perspective. From the perspective of the detector, on the other hand, the volume rotates. This change in perspective is illustrated in the last two panes of Figure 5.3. Both perspectives yield the same projection operator **A**.

5.2.4 Interoperability and GPU-acceleration

In this section, we discuss tomosipo's interoperability with NumPy arrays [65] and GPU-accelerated Python packages. In addition, we discuss the performance benefits of using GPU-accelerated arrays and also some trade-offs in favor of CPU arrays.

Projection operations are calculated using the ASTRA Toolbox. We have extended the ASTRA Toolbox API to enable direct operation on NumPy arrays. Before any ASTRA operation, the input arrays are automatically linked to the ASTRA runtime, and unlinked afterwards. This represents a substantial ergonomic improvement over the existing API. Apart from NumPy arrays, array types from other Python packages can also be linked. Out of the box, tomosipo interoperates with PyTorch and CuPy [136, 139]. More integrations can be added though an API, which can be used in the future to add interoperability with a variety of array libraries through the currently developing Python array API standard[3].

Integration with GPU array libraries can enable substantial performance improvements. In the snippet below, a NumPy array and a PyTorch array are forward projected. The NumPy array is located in RAM attached to the CPU, and the PyTorch array is located on the GPU.

```
y_numpy = A(np.ones(A.domain_shape, dtype=[...]))
y_torch = A(torch.ones(A.domain_shape, device="cuda"))
```

On line 1, the data is first moved to the GPU, the forward projection is calculated, and the data is moved back to the CPU. On line 2, no data movement takes place: the forward projection is calculated on the GPU. In iterative algorithms, where the forward and backprojection are repeatedly executed substeps of the algorithm, the latency imposed by CPU-GPU communication can dominate the computation time, as we demonstrate in Section 5.5. Note that PyTorch arrays can also be created on the CPU. In that case, the computation of the forward projection goes through exactly the same steps as a NumPy array would.

[3]`https://data-apis.org/array-api/latest/purpose_and_scope.html`

There are cases where it is beneficial to keep data on CPU. When data is too big to fit in GPU memory, the ASTRA Toolbox automatically splits data residing on CPU and performs the computation on the GPU in a streaming fashion. In this case, the user does not have to split up the data manually. When multiple GPUs are present on the system, they can be used automatically. In the code below, the ASTRA Toolbox is instructed to use four GPUs on line 1. The computation of the forward projection on line 2 is distributed over the four GPUs.

```
astra.set_gpu_index([0, 1, 2, 3])
y_numpy = A(np.ones(A.domain_shape, dtype=[...]))
```

5.2.5 Splitting and joining geometries

The ability to split and join geometries in tomosipo's API allows users to customize their design easily. Tomosipo allows subsampling a geometry to obtain a subgeometry. In addition, it allows joining the time steps of sequences of geometries into a single geometry. First, we demonstrate subsampling of projection geometries. Subsampling of volume geometries and geometric transforms works similarly and is described in the documentation. A projection geometry can be subsampled to obtain a geometry describing a subset of the detector surface. In the code below, the detector surface is cropped, removing a border of 100 pixels from each side. On the next line, the detector surface is subsampled, selecting every other row and column of pixels. Subsampling induces a slight shift in the detector's center, which is taken into account and described in detail in the documentation.

```
pg_cropped = pg[:, 100:-100, 100:-100]
pg_subsampled = pg[:, ::2, ::2]
```

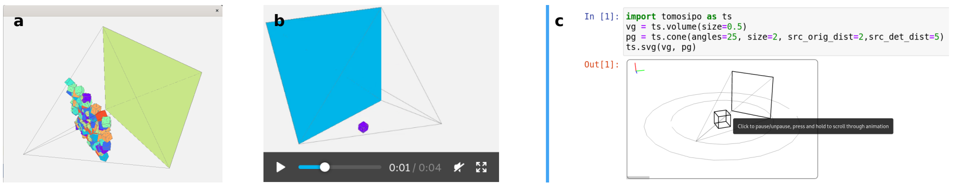
Subsampling the angular dimension is also possible. In this dimension, subsampling supports both slicing as well as Boolean masks [65]. In the code below, the angular direction is subsampled, obtaining a geometry that contains every other projection angle. In the line below, angles are selected when a `condition` array equals `True`.

```
pg_even_angles = pg[::2]
pg_boolean = pg[condition == True]
```

In Section 5.3.3, Boolean masking is demonstrated in the case study of X-ray diffraction tomography, where diffraction occurs in a subset of projection angles.

In addition to indexing, tomosipo also includes functionality to concatenate geometries and transforms. The concatenation of multiple projection geometries combines their time steps into a single geometry. In the code below, two projection geometries are combined. In the next line, a rotation `R` is repeatedly composed with different translations `T1, T2, T3`.

```
pg_combined = ts.concatenate([pg1, pg2])
TR_combined = ts.concatenate([T1 * R, T2 * R, T3 * R])
```


a
b
0:01 / 0:04
c
Click to pause/unpause, press and hold to scroll through animation

```
In [1]: import tomosipo as ts
        vg = ts.volume(size=0.5)
        pg = ts.cone(angles=25, size=2, src_orig_dist=2,src_det_dist=5)
        ts.svg(vg, pg)
Out[1]:
```

```
anim = animate(*vg, pg)
anim.window()
```

```
anim = animate(vg, pg)
anim.save("video.mp4")
```

```
svg = ts.svg(vg, pg)
svg.save("fig.svg")
```

Figure 5.4: Visualization options in tomosipo: (**a**) interactive 3D environment, (**b**) video, (**c**) interactive animation in a Jupyter Notebook. A single-particle Cryo-EM setup [17] is shown in panes (**a**) and (**b**), and a circular cone beam acquisition is shown in pane (**c**). Code snippets demonstrate how visualizations are created.

The concatenation of transforms is demonstrated in the case study of X-ray scattering tensor tomography in Section 5.3.4, where it is used to define a repeated rotation at several tilt angles.

5.2.6 Visualization

Tomosipo provides extensive support for visualizing geometries. Animations can be saved as a video or as a scalable vector graphic (SVG). In addition, geometries can be investigated in a 3D-accelerated environment on the desktop, allowing the user to zoom, pan, and rotate the view. Finally, an interactive SVG animation can be shown in an online Jupyter notebook, allowing for quick inspection of intermediate results. These options are illustrated in Figure 5.4. All other illustrations in this chapter have been generated using tomosipo. They were saved in the SVG format and extended using Inkscape.

5.3 Case studies

In this section, the concepts developed in the previous section are put into practice. We describe two simple examples and two complex acquisition schemes that are in use at synchrotron tomography beamlines. The first example demonstrates how geometric transforms can be composed to create a helical cone beam geometry. The second example models single-axis parallel beam tomography with a non-standard center of rotation in the frame of reference of the laboratory. In the first case study, we describe X-ray diffraction contrast tomography, which demonstrates the use of reflection and subsampling using a Boolean mask. In the second case study, we describe X-ray scattering tensor tomography, which demonstrates the use of concatenation. The case studies demonstrate that X-ray diffraction and scattering can be modeled using tomosipo's projection operators.

```
t = np.linspace(-1, 1, 100)    # Time t = -1.0, -.98, ..., 1
s = 2 * np.pi * t              # Angle
radius = 2                     # Radius of helix
h = 1.0                        # Vertical "speed"

vg = ts.volume()
pg = ts.cone(src_orig_dist=radius, src_det_dist=2 * radius)

R = ts.rotate(pos=0, axis=(1, 0, 0), angles=s)
T = ts.translate(axis=(1, 0, 0), alpha = h * s / (2 * np.pi))
H = T * R

ts.svg(vg, H * pg.to_vec())
```

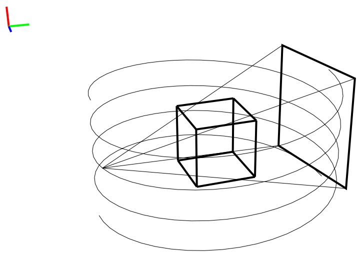


Figure 5.5: A helical cone beam geometry can be obtained as a composition of translation and rotation. The volume and cone beam geometries are defined to be non-moving. At each time step, the helical transform H applies a rotation R and then a translation T to the cone beam geometry.

5.3.1 Basic example: Helical cone beam geometry

As a demonstration of the composition of two primitive transforms, we define the helical cone beam geometry [88]. Here, the source and detector follow a helical path around the object. Using the notation of [88], we describe the helical geometry as a composition of translation and rotation in Figure 5.5. First, a static volume and a static cone beam geometry are defined. Next, a rotation R and translation T are defined, which rotate around and translate along the z-axis. The helical transform H is defined such that it applies the rotation R_i and then a translation T_i at time step i. When it is applied to the cone beam geometry, the resulting trajectory of the source and detector is helical. We note that the helical trajectory could have been obtained as a translation of a non-static cone beam geometry, effectively hiding the rotation in the cone beam geometry.

5.3.2 Basic example: Parallel beam in the lab frame

Acquisition using the single-axis parallel beam geometry is common at synchrotron beam lines. The detector is often positioned at a fixed location and the sample is mounted on a movable rotation stage. Typically, it is assumed that the center of rotation and the center of the detector coincide. In many cases in practice, however, it is difficult to achieve this with the described setup. Therefore, the offset between the center of rotation and the center of the detector has to be taken into account in order to achieve an accurate reconstruction. This is possible in tomosipo by positioning the detector, volume, and rotation axis independently from each other.

In Figure 5.6, the acquisition geometry is defined in the frame of reference of the laboratory. First, a static detector is translated from the origin to its final position by a transform `T`. Next, a static volume geometry is created at the initial position of the sample. A rotation is defined with a specific position of the rotation axis. Finally, the rotation is applied to the static volume, obtaining a rotating volume whose center rotates around the rotation axis.

There is an advantage to this formulation. In existing tomography packages, the position of the volume is commonly chosen to coincide with the rotation axis. However, this causes the reconstructed images to be translated when a different

```python
# Static detector at custom position
T = ts.translate(det_pos)
static_pg = ts.parallel(angles=1, shape=det_shape)
pg = T * pg_static.to_vec()

# Static volume at custom position
vg_static = ts.volume(pos=vol_pos, shape=vol_shape)

# Rotate the volume
R = ts.rotate(pos=rot_axis_pos, axis=z_axis, angles=angles)
vg = R * vg_static.to_vec()

A = ts.operator(vg, pg)
```

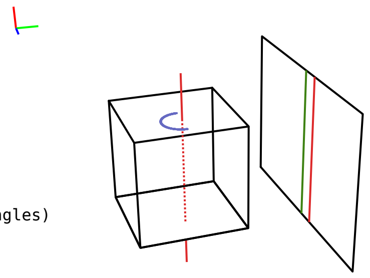


Figure 5.6: A single-axis parallel beam acquisition with a custom center of rotation. An object, whose changing position during rotation is indicated in blue, is rotated around a non-standard axis of rotation (in red). The center of the detector is indicated in green.

center of rotation is provided. This can cause problems when the determination of the correct center of rotation is based on the reconstructed images. In contrast, the proposed formulation opens up the possibility of determining the correct center of rotation by maximizing the auto-correlation in the reconstruction at several values of the center of rotation.

5.3.3 Complex case study: Diffraction contrast tomography

X-ray diffraction contrast tomography (DCT) [184] is an imaging technique used to investigate the internal structure of poly-crystalline materials. The crystal lattice is divided into grains, homogeneous regions where the lattice has a similar orientation. The orientation, size, shape, and arrangement of individual grains strongly influence macroscopic properties of the material. Therefore, mapping the orientation of grains is important to characterize a poly-crystalline material [124]. Here, we take as an example the three-dimensional DCT acquisition geometry as described in [184] to demonstrate specific features of tomosipo.

The goal of DCT is to reconstruct a vector field representing the intra-granular orientation of the crystal lattice. This is achieved by discretizing the orientation space on a regular grid that can be represented by unit vectors $\hat{\mathbf{o}}_1, \ldots, \hat{\mathbf{o}}_{N_o}$. For each orientation $\hat{\mathbf{o}}_k$, a scalar field, i.e., a volume, is reconstructed that represents the diffraction "intensity" at that orientation. A variational reconstruction algorithm ensures that neighboring voxels have similar orientations. A crystal lattice reflects an incoming X-ray beam at specific incidence directions, characterized by the so-called Bragg angles. When the diffraction geometry of the material under investigation is known beforehand, the intra-granular orientation of the crystal lattice can be recovered from those projection images at which Bragg diffraction is expected to occur.

As shown in Figure 5.7, the acquisition uses a monochromatic parallel box beam and the diffracted signal of the sample is measured on a flat-panel detector. As the sample is rotated, the reflection of the incoming beam in a voxel with local orientation $\hat{\mathbf{o}}$ forms a figure of eight on the detector. Bragg diffraction only

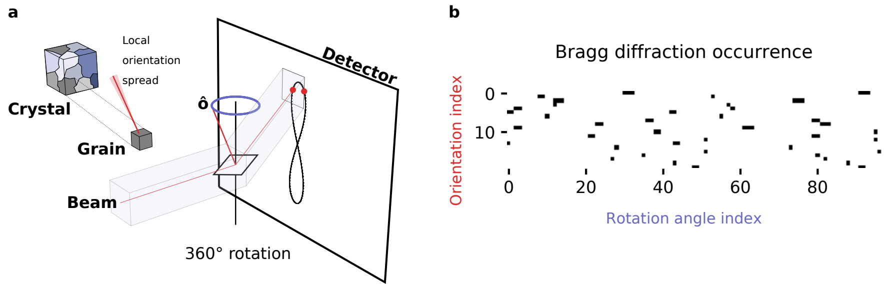


Figure 5.7: (**a**) In X-ray diffraction contrast tomography, a crystal sample is illuminated by a monochromatic X-ray box beam. The crystal sample is divided into grains, which have a minimal spread in local orientation. As the sample is rotated, the incoming beam is diffracted when its incident angle with the local lattice plane equals the Bragg angle. The intersection points of the reflected beam with the detector form a figure of eight, on which Bragg diffraction occurs only twice (marked in red) as the sample is rotated. (**b**) For a random sample of 20 orientations, the occurrence of Bragg diffraction at a rotation angle is indicated in black. Here, diffraction occurs in just 3.3% of the orientation-rotation combinations.

occurs in the instances where the beam and local lattice are in Bragg condition. Occurrence of the Bragg condition is relatively rare, as shown in Figure 5.7 (**b**). Both the reflection and its intermittent nature can be modeled in tomosipo.

Bragg diffraction (reflection). The diffraction, i.e., reflection, of an incident X-ray beam can be represented in tomosipo. As shown in Figure 5.7, a parallel bundle of rays remains parallel after it has been reflected. Therefore, the measurement of a diffracted parallel beam can be modeled using a standard parallel-beam geometry with altered ray direction.

The code below models the reflection of the incoming beam by a rotating crystal lattice. The orientation of the lattice is represented by a plane normal vector. First, the plane normal of the crystal lattice is rotated. Then, a reflection M is created in the rotating plane normal. The position of the reflection is arbitrary, as it is used to transform the direction of the beam and not its location. Finally, a static parallel beam geometry is modified such that the ray direction corresponds to that of the beam reflected in the rotating plane normal. The position and orientation of the detector remain static.

```python
# Rotation of the rotation stage
R = ts.rotate(pos=0, axis=(1, 0, 0), angles=rot_angles)

def diffracted_pg(pg_static, plane_normal, R):
    rotated_plane_normal = R.transform_vec(plane_normal)
    M = ts.reflect(pos=0, axis=rotated_plane_normal)
    return ts.parallel_vec(
        shape=pg_static.det_shape,
        ray_dir=M.transform_vec(pg_static.ray_dir),
        det_pos=pg_static.det_pos,
        det_v=pg_static.det_v,
        det_u=pg_static.det_u,
    )
```

Bragg condition (Boolean masking). The occurrence of Bragg diffraction can be considered as a Boolean mask, an example of which is shown in Figure 5.7 (**b**). It is computed in the code below. First, the plane normal is rotated. Then, the Bragg condition is determined at each rotation angle.

```
bragg_mask = np.empty((num_orientations, num_angles), dtype=bool)

for i in range(num_orientations):
    rotated_normal = R.transform_vec(plane_normals[i])
    for j in range(num_angles):
        bragg_mask[i, j] = in_bragg_condition(
            rotated_normal[j], incoming_ray_dir, bragg_angle
        )
```

The created Boolean mask is used to select a subset of each projection geometry. For each orientation, the code below creates an operator that computes the forward projection only at rotation angles where Bragg diffraction occurs.

```
vg = R * ts.volume(shape=100).to_vec()
diffracted_pgs = [
    diffracted_pg(pg_static, normal, R) for normal in plane_normals
]
# Compute an operator per orientation
operators = [
    ts.operator(vg[bragg_mask[i]], diffracted_pgs[i][bragg_mask[i]])
    for i in range(num_orientations)
]
```

Multi-orientation tomography (sums of masked operators). The forward projection computes the diffraction pattern of $\mathbf{x} \in \mathbb{R}^{N_o \times N_v^3}$, representing all discretized plane orientations at N_v^3 locations, onto $\mathbf{y} \in \mathbb{R}^{N_\theta \times N_p^2}$, representing the N_p^2 pixel intensities at N_θ rotation angles. The operation is a linear combination of the masked operators defined above.

```
x = np.zeros((num_orientations, *vg.shape), dtype=np.float32)

def fp(x):
    y = np.zeros((N_p, N_angles, N_p), dtype=np.float32)
    for x_oriented, A, mask in zip(x, operators, bragg_mask):
        y[:, mask] += A(x_oriented)
    return y

y = fp(x)
```

In the interest of space, the backprojection operation is omitted. The full code listing can be found in the Supplemental materials of [74]. Implementing the variational reconstruction algorithm described in [184] is outside of the scope of this manuscript. We have shown how the DCT geometry can be succinctly expressed using tomosipo's rotation and reflection transformations. In addition, we have used subsampling with a Boolean mask to limit the forward projection to the few instances where Bragg diffraction occurs.

5.3.4 Complex case study: X-ray scattering tensor tomography

X-ray scattering tensor tomography (XSTT) is an imaging technique used to investigate materials with micro- and nano-scale structures over an orders of magnitude larger volumetric field of view, compared to conventional tomographic modalities [110, 117]. Here, we take the XSTT acquisition geometry that is described in [93] as an example to demonstrate specific features of tomosipo.

The goal of XSTT is to reconstruct a vector field representing the directional scattering intensity of a sample. This is achieved by reconstructing $N_{\hat{\mathbf{s}}} \geq 6$ scalar fields that represent the squared scattering coefficient along unit vector $\hat{\mathbf{s}}_1, \ldots, \hat{\mathbf{s}}_{N_{\hat{\mathbf{s}}}}$ at each voxel. After reconstruction, the directional scattering intensities are fine-tuned using per-voxel PCA (principal component analysis) [187]. XSTT has various biological and industrial applications [93]. As an example, the recovered local directional scattering intensities can be used to predict macroscopic properties of fibrous materials. These properties depend on the local fiber arrangement. Fibers scatter X-rays the least along their primary fiber orientation. Therefore, the local fiber orientation can be recovered from the shortest principal axis (smallest scattering magnitude) of the fitted scattering ellipsoid. The possibility to investigate these local structures over large enough volumes is valuable for the research and development of new materials.

We describe the acquisition process to obtain one of the scalar fields $\mathbf{x}_{\hat{\mathbf{s}}}$, representing the squared scattering coefficient along a unit vector $\hat{\mathbf{s}}$. First, we discuss the forward model at a single orientation of the sample, i.e., without any rotation or tilting. Let $\mathbf{x}_{\hat{\mathbf{s}}} \in \mathbb{R}^{N_v^3}$ represent the sample's squared scattering coefficient along a vector $\hat{\mathbf{s}}$. The sample is illuminated by a monochromatic parallel X-ray beam. Before they are measured on a detector, the X-rays travel through a panel that is etched with a periodic array of multi-circular gratings [93], generating a reference pattern. The panel is placed at a fixed propagation distance from the detector to maximize the visibility of the patterns. Different types of gratings require different

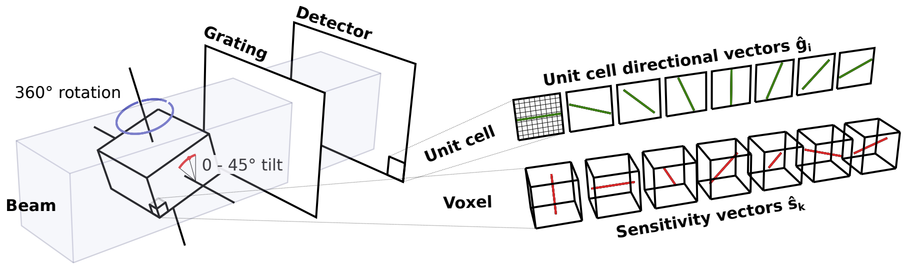

360° rotation
Beam
Grating
Detector
0 - 45° tilt
Unit cell
Voxel
Unit cell directional vectors $\hat{\mathbf{g}}_i$
Sensitivity vectors $\hat{\mathbf{s}}_k$

Figure 5.8: In X-ray scattering tensor tomography, a sample is illuminated by a box beam. The sample is repeatedly rotated at several tilt angles. The scattered signal passes through an array of gratings before being measured on a detector. The detector pixels are grouped into 9×9 pixel unit cells. In each unit cell, a directional intensity is measured along vectors $\hat{\mathbf{g}}_i$. In each voxel, scattering coefficients for multiple scattering sensitivity vectors $\hat{\mathbf{s}}_k$ are reconstructed.

acquisition geometries. The acquisition discussed in this case study is specifically geared to circular gratings.

With the use of circular gratings, the pixels of the detector are grouped into 9×9 pixel unit cells. In each unit cell, a 2D directional intensity is measured along multiple unit vectors $\hat{\mathbf{g}}_i$. The measured intensity $\mathbf{y}_i$ along the vector $\hat{\mathbf{g}}_i$ on the detector for a single beam direction $\hat{\mathbf{b}}$ is computed by scaling the forward projection with the scalar $\nu_{\hat{\mathbf{b}},\hat{\mathbf{s}},\hat{\mathbf{g}}_i}$ [117], defined by

$$\left(\left|\hat{\mathbf{b}} \times \hat{\mathbf{s}}\right| \langle \hat{\mathbf{s}}, \hat{\mathbf{g}}_i \rangle\right)^2 \mathbf{A}_{\hat{\mathbf{b}}}\, \mathbf{x}_{\hat{\mathbf{s}}} = \nu_{\hat{\mathbf{b}},\hat{\mathbf{s}},\hat{\mathbf{g}}_i} \mathbf{A}_{\hat{\mathbf{b}}}\, \mathbf{x}_{\hat{\mathbf{s}}} = \mathbf{y}_i. \tag{5.3}$$

The scaling is the same for each unit cell on the detector and varies as the sample (and thus $\hat{\mathbf{s}}$) is rotated.

Rotation and tilt (concatenation). When using circular gratings, the sample must be measured with multiple tilted rotation axes [93, 109]. In the XSTT acquisition described in [93], the sample stage is rotated, while the stage is tilted in steps. At each step, the stage makes a full rotation, as illustrated in Figure 5.8. Each step can be represented in tomosipo by composing a single tilt operation with a full rotation. In the code below, the full motion is computed by concatenating each step.

```python
tilt = ts.rotate(pos=0, axis=(0, 0, 1), angles=tilt_angles)
rotate = ts.rotate(pos=0, axis=(1, 0, 0), angles=rotation_angles)
# For each tilt angle, perform a full rotation:
TR = ts.concatenate([tilt_single * rotate for tilt_single in tilt])
```

At each tilt and rotation angle, the scaling $\nu_{\hat{\mathbf{b}},\hat{\mathbf{s}},\hat{\mathbf{g}}_i}$ from Equation 5.3 is calculated as follows:

```python
def calculate_nu(b, s, g, TR):
    nu = np.zeros(TR.num_steps)
    for j, s_rot in enumerate(TR.transform_vec(s)):
        nu[j] = (norm(np.cross(b, s_rot)) * np.dot(s_rot, g)) ** 2
    return nu
```

Because the calculation is performed in the lab frame, the vector $\hat{\mathbf{s}}$ is rotated rather than the beam direction $\hat{\mathbf{b}}$ or sensitivity vector $\hat{\mathbf{g}}$,

Scaled linear combinations. After $\nu \in \mathbb{R}^{N_{\hat{\mathbf{s}}} \times N_{\hat{\mathbf{g}}} \times N_{\text{tilt}} N_{\text{rot}}}$ is calculated for all values of $\hat{\mathbf{s}}_1, \ldots, \hat{\mathbf{s}}_{N_{\hat{\mathbf{s}}}}$, all $\hat{\mathbf{g}}_i$, and all tilts and rotations, then the full projection can be calculated. Here, the measurement along $\hat{\mathbf{g}}_i$ is the sum of the contributions of the $N_{\hat{\mathbf{s}}}$ scalar fields representing the scattering coefficients of the sample, as calculated below. In the interest of space, the backprojection operation is omitted. The full code listing can be found in the Supplemental materials of [74].

```python
def fp(x, nu):
    y = torch.zeros(num_g, *A.range_shape, device=x.device)
    for k in range(num_s):
        for i in range(num_g):
            y[i] += nu[k, i][None, :, None] * A(x[k])

    return y
```

Data size. The reconstruction problem considered in [93] fits in memory on modern GPUs. The measured data consists of 46 tilt angles, 50 rotation angles, and 100×144 unit cells. Measuring along 8 $\hat{\mathbf{g}}_i$ vectors, the total number of measured unit cells equals $46 \times 50 \times 100 \times 144 \times 8 \approx 256 \times 10^6$, which requires approximately 1 GB when stored in 32 bit precision. The reconstruction volume consists of $44 \times 71 \times 71$ voxels, repeated for each of $N_{\hat{\mathbf{s}}} = 7$ scattering directions. In total, it requires roughly 6 MB to store in 32 bit precision. The size of the scaling matrix ν is negligible in comparison. Modern data center GPUs range in memory size from 16 GB to 80 GB. Therefore, it is possible to run an iterative SIRT reconstruction of the full problem on GPU. Benchmarks comparing the performance on GPU versus CPU are provided in Section 5.5.

5.4 Experimental data

In this section, we show reconstructions of experimental data acquired using the standard circular cone beam and single-axis parallel beam trajectories, as well as a reconstruction of an X-ray scattering tensor tomography dataset. The reconstructions have been computed using the algorithms implemented in the separate ts_algorithms package.

Circular cone beam. A laboratory micro-CT dataset of a bell pepper was acquired at the FleX-ray laboratory[42] at the CWI, Amsterdam, The Netherlands. A polychromatic microfocus X-ray point source with tube voltage and power of 90 kV and 49.5 W was used. The data consisted of 3600 projection images of 1512×1912 pixels, acquired over a 360° rotation. A reconstruction was computed on a grid of $1512 \times 1912 \times 1912$ voxels using FDK, a backprojection-type algorithm [53]. An axial slice of the reconstruction is shown in Figure 5.9 (**a**).

Single axis parallel beam. A 3D micro-tomography dataset of a fuel cell from the publicly available TomoBank [44] was used. This dataset (#81) was acquired at the TOMCAT beamline at the Swiss Light Source (SLS) at the Paul Scherrer Institut (PSI), Villigen, Switzerland [28]. The first 3600 projection images of 1100×1440 pixels were used to compute a reconstruction on an axial slice of 1400×1400 pixels. The reconstructions were computed using FBP (Ram-Lak filter), SIRT (200 iterations), and TV-MIN (500 iterations with $\lambda = 2 \times 10^{-7}$), as shown in Figure 5.9 ($\mathbf{b} - \mathbf{d}$).

X-ray scattering tensor tomography. The same validation sample was used as in a previous publication [93], which was also acquired at the TOMCAT beamline. It consisted of a $4 \times 4 \times 4\,\text{mm}^3$ plastic box containing three orthogonally oriented bundles of carbon fiber with a 12 µm diameter. The pixel and resulting unit cell size was $11 \times 11\,\mu\text{m}^2$ and $99 \times 99\,\mu\text{m}^2$, generating the dataset size described at the end of Section 5.3.4. An illustration of the validation sample and its reconstruction using tomosipo is shown in Figure 5.10. The reconstructions show the orientation of the fibers after post-processing using PCA and a similar thresholding strategy as in [93]. Thresholding causes noise in the background to be suppressed, as the X-ray scattering induced by plastic container is known to be negligible.

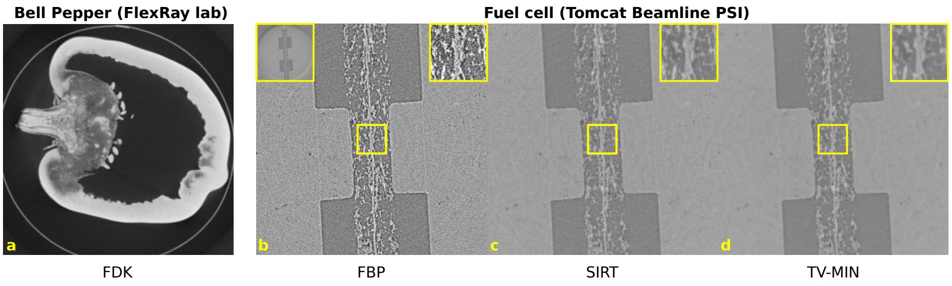


Figure 5.9: Reconstructions of experimental data acquired using laboratory micro-CT (**a**) and synchrotron micro-tomography (**b** − **d**). The yellow insets in the top-right corner show a magnified region of interest. The yellow inset in the top-left of pane (**b**) displays a full view, showing field-of-view artifacts due to the truncated projection images.

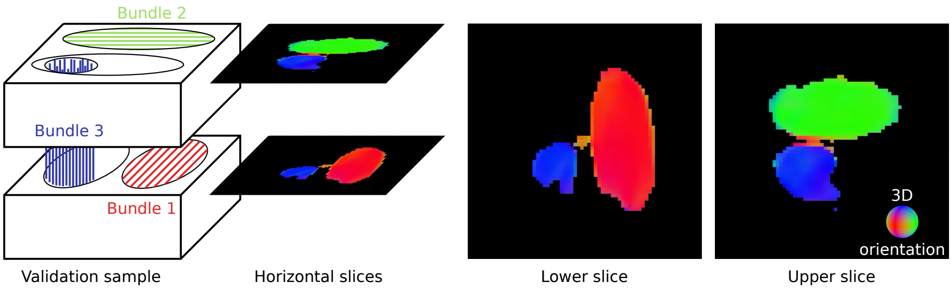


Figure 5.10: X-ray scattering tensor tomography reconstruction of a validation sample. The sample contains three orthogonally oriented carbon fiber bundles. A reconstruction of the orientation map is shown in two axial slices.

5.5 Benchmarks

In this section, we give a demonstration of the computational speed of tomosipo. First, we compare an implementation of SIRT in tomosipo to the built-in implementation in the ASTRA Toolbox. Using the tomosipo implementation, we also investigate the impact of storing intermediate data on the CPU rather than on the GPU. This comparison is run on the examples from Section 5.3 with data sizes that fit on a single GPU. We exclude DCT, as its reconstruction algorithm is out of the scope of this manuscript. We also benchmark a non-iterative algorithm on a circular cone beam dataset that does not fit on the GPU. Here, we compare the speed of the built-in FDK implementation of the ASTRA Toolbox to the FDK implementation in ts_algorithms, tomosipo's accompanying reconstruction algorithms package.

We describe the algorithms, data size, and benchmark methodology. The SIRT reconstructions were computed in 50 iterations. The implementation in tomosipo used PyTorch and the ASTRA implementation used the `SIRT3D_CUDA` algorithm. The FDK benchmark compared the FDK implementation provided

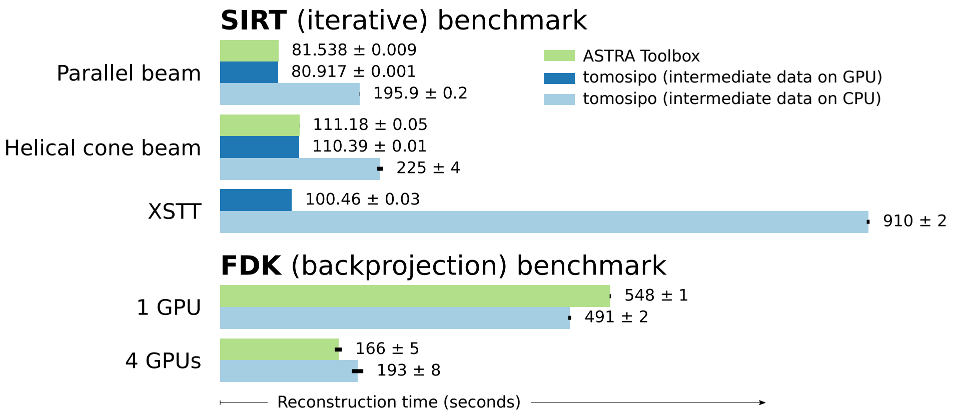


Figure 5.11: Comparison of reconstruction times using SIRT on a GPU-sized problem and using FDK on a lab-CT-sized problem. The SIRT implementations are compared on a parallel, helical and X-ray scattering tensor tomography (XSTT) acquisition geometry. The XSTT reconstruction cannot be implemented using the built-in ASTRA SIRT API. Because the FDK dataset is too large, intermediate data cannot be stored on the GPU, and the ASTRA implementation is compared to a tomosipo implementation that performs the filtering step on the CPU.

by ts_algorithms to ASTRA's built-in `accumulate_FDK` implementation. The dataset of the parallel beam and helical cone beam cases consisted of 768×768 pixel projection images acquired over 512 angles and was reconstructed on a 512^3 voxel volume. The sizes of the XSTT and circular cone beam dataset are described in Sections 5.3.4 and 5.4, respectively. The benchmarks were conducted on a dual-socket system containing 8-core Intel Xeon Silver 4110 CPUs at 2.10 GHz (Intel, Santa Clara, CA, USA) with 192 GB of RAM and four Nvidia GeForce GTX 1080 Ti GPUs (Nvidia, Santa Clara, CA, USA). Each benchmark was run once without measurement to minimize startup and caching effects. The mean and standard deviation of three trials are reported.

SIRT on GPU-sized problems. The results of the SIRT benchmark are shown in Figure 5.11. The ASTRA Toolbox and the Tomosipo implementation with intermediate data on the GPU are close in performance. They are are $2-9\times$ faster than the tomosipo implementation with intermediate data located on CPU memory. This indicates that CPU-GPU communication latency is non-negligible and that reconstruction algorithms benefit from being completely computed on the GPU. We note that in all three implementations the forward and backprojection are computed on the GPU using the ASTRA Toolbox. The native ASTRA SIRT implementation does not have an option to store intermediate data on CPU memory.

FDK on a lab-CT-sized problem. The FDK dataset is too big to fit in GPU memory. In tomosipo's implementation, the filtering step is performed on the CPU and the computation of the backprojection on chunks of projection data is distributed over multiple GPUs. The FDK implementation in the ASTRA Toolbox, on the other hand, first distributes chunks of projection data over available GPUs and performs the filtering and backprojection in a single step on each GPU.

Figure 5.11 shows the results of the FDK benchmark using one and four GPUs. Using one GPU, tomosipo's implementation of FDK is faster than ASTRA's. This can be attributed to fast filtering on the CPU, which is implemented using the Fast Fourier Transform provided by PyTorch and is approximately as fast as filtering on a single GPU. Using four GPUs, the run times of both implementations are reduced, but the ASTRA implementation comes out ahead. When four GPUs are available, the ASTRA implementation distributes the computation of the filter step over four GPUs, whereas the tomosipo implementation still computes the filtering step on the same amount of CPU cores.

The results show that a naive implementation of an iterative algorithm in tomosipo is not necessarily slower than a native implementation in the ASTRA Toolbox. In addition, the results illustrate the substantial negative impact that CPU-GPU communication has on reconstruction speed. Finally, the FDK results illustrate the benefits of interoperability with fast array libraries, but highlight the need for effective APIs to address multi-device streaming computation.

5.6 Discussion and conclusion

In short, tomosipo provides the expressive power to quickly and naturally define complex geometries, thereby unlocking the flexibility provided by the ASTRA Toolbox. We have demonstrated the ease of making common adjustments to an acquisition geometry, such as changing the center of rotation. In addition, the design and implementation of more complex geometries, such as the demonstrated X-ray diffraction and scattering setups, is made considerably easier by using tomosipo, especially compared to entering the formulas for all directional vectors manually. Reconstructions of real-world data from synchrotron and laboratory micro-CT sources are shown, computed using several common reconstruction algorithms. Finally, bechmarks demonstrate that the package enables the user to write fully GPU-accelerated reconstruction algorithms in Python whose speed is on par with native implementations. Because of tomosipo's interoperability with GPU-accelerated Python array libraries, intermediate results can remain on the GPU, avoiding the latency imposed by CPU-GPU communication.

The tomosipo package follows best practices. It has a comprehensive unit test suite, it is installable through the Anaconda package manager, it follows semantic versioning, it is developed in the open on GitHub, and it has extensive documentation.

Future developments are expected to go hand in hand with improvements in the ASTRA Toolbox. This includes support for curved detectors and more fine grained control of streams on the GPU, allowing for concurrency through pipelining. In addition, we intend to extend the interoperability of tomosipo's projection operator to more optimization packages. We note that the integration of tomosipo's projection operator in deep learning-based reconstruction methods using PyTorch is possible and is described in the documentation.

Compared to existing tomographic software packages, two features set tomosipo apart. First, the facilities to manipulate geometries significantly simplify defining complex acquisition geometries such as those in the described case studies. Although other packages including the ASTRA Toolbox and the Core Imaging Library (CIL) [82] can represent these acquisition geometries, they do not provide tools to define them. Specifically, the geometric transforms, subsampling, concatenation, and visualization features are not provided by the ASTRA Toolbox. Second, the extensible integration of tomosipo with GPU-accelerated Python array libraries provides two advantages. It enables the user to write custom reconstruction algorithms in Python that are comparable in computational efficiency to a native implementation. In addition, it enables integrating tomographic operators in deep learning-based reconstruction methods. This is technically possible using the ASTRA Toolbox, but the APIs that it exposes are designed to be wrapped by a user-friendly library, such as tomosipo.

We stress that tomosipo aims to be a building block in a larger system. Therefore, other software packages may be preferable for many purposes. Facilities for loading of various file formats, preprocessing of tomographic data, or post-processing of reconstructed images are present in TomoPy, Savu, and CIL [64, 82, 188]. Packages such as PyLops, CIL, and JUDI [82, 150, 193] provide building blocks and built-in optimization algorithms that enable rapid prototyping of variational reconstruction methods, among others. The reconstruction algorithms show-cased in this manuscript, on the other hand, are implemented in a separate package [70]. An advantage of the focused scope of tomosipo, is that it has only two required dependencies (NumPy and the ASTRA Toolbox), making it easy to install on various platforms, but contains several integrations with third-party packages, making it easy integrate into an existing system.

In summary, tomosipo provides scientists with an excellent tool to model and visualize complex tomographic acquisition geometries while maintaining and extending the fast reconstruction capabilities of the ASTRA Toolbox.