Numer Algor (2016) 71:673–697
DOI 10.1007/s11075-015-0016-4

ORIGINAL PAPER

Easy implementation of advanced tomography algorithms using the ASTRA toolbox with Spot operators

Folkert Bleichrodt[1] · Tristan van Leeuwen[1] · Willem Jan Palenstijn[1,2] · Wim van Aarle[2] · Jan Sijbers[2] · K. Joost Batenburg[1,2,3]

Received: 11 December 2014 / Accepted: 4 June 2015 / Published online: 21 August 2015

Abstract Mathematical scripting languages are commonly used to develop new tomographic reconstruction algorithms. For large experimental datasets, high performance parallel (GPU) implementations are essential, requiring a re-implementation of the algorithm using a language that is closer to the computing hardware. In this paper, we introduce a new MATLAB interface to the ASTRA toolbox, a high performance toolbox for building tomographic reconstruction algorithms. By exposing the ASTRA linear tomography operators through a standard MATLAB matrix syntax, existing and new reconstruction algorithms implemented in MATLAB can now be applied directly to large experimental datasets. This is achieved by using the Spot toolbox, which wraps external code for linear operations into MATLAB objects that can be used as matrices. We provide a series of examples that demonstrate how this Spot operator can be used in combination with existing algorithms implemented in MATLAB and how it can be used for rapid development of new algorithms, resulting in direct applicability to large-scale experimental datasets.

Keywords Linear operators · Reconstruction algorithms · Software · Tomography

Mathematics Subject Classification (2010) 65F10 · 65F22 · 65F50

✉ Folkert Bleichrodt
F.Bleichrodt@cwi.nl

[1] Centrum Wiskunde & Informatica, Science Park 123, 1098 XG Amsterdam, The Netherlands

[2] iMinds - Vision Lab, University of Antwerp, Universiteitsplein 1, B-2610 Antwerp, Belgium

[3] Mathematisch Instituut, Leiden University, Niels Bohrweg 1, 2333 CA Leiden, The Netherlands

1 Introduction

Tomography is an imaging technique for reconstructing an object from projections. In medical imaging, projections can be obtained as X-ray images by CT-scanners. In the scientific community, many devices and setups are used for tomographic data acquisition, from electron microscopes to large synchrotron facilities [13, 25]. Hardware advances have pushed the ability to image on smaller scales and at large pixel densities. Projection images in the order of 4000 by 4000 pixels can now be obtained routinely [21]. At the same time, many innovative tomography applications are inherently limited in the number of projections that can be acquired, and their associated noise level.

In recent years, we have seen many advances in reconstruction algorithms for tomography that incorporate prior knowledge about the scanned object. Examples can be found in sparse reconstruction techniques and discrete tomography [4, 14, 16, 31, 33, 34]. The benefit of these methods is their ability to produce accurate reconstructions from limited projection data. To develop such algorithms, high-level mathematical scripting languages such as MATLAB are commonly used due to the complex mathematics involved. As a result, the initial implementations may not be suitable for dealing with large experimental datasets due to the inherent performance and memory limits of the scripting platform.

Tomography algorithms are usually constructed by combining two linear operators – *forward projection* and *backprojection* – with additional algorithmic steps. For large-scale datasets, the corresponding matrices are too large to store explicitly. For this reason, MATLAB implementations based on explicit matrix computations cannot be used in a straightforward manner. Efficient, parallel (GPU) implementations are used instead [2, 22]. Although software packages that exploit parallelism are widespread [10, 24, 29, 30, 36], it is not trivial to use these software implementations in combination with algorithms written in scripting languages.

The work presented in this paper is based on two popular toolboxes for MATLAB: the ASTRA toolbox [1, 27, 28] and the Spot toolbox [7]. The ASTRA toolbox is a MATLAB toolbox for tomographic reconstruction, based on high-performance GPU primitives. It supports multiple geometries (parallel beam, fan beam, cone beam) with highly flexible source/detector positioning. The Spot toolbox exposes external implementations of linear operations through a standard MATLAB matrix interface.

Our key contribution is the introduction of a Spot operator for ASTRA, which we have named `opTomo`. We will show how it can be used to easily develop complex tomography algorithms that are directly applicable to large datasets. Our examples show that the `opTomo` operator enables the use of a range of built-in and external MATLAB packages for tomography. Additionally, the Spot operator can be used to develop new algorithms without having to deal with complex implementation details. The code resembles pseudocode and is therefore easy to understand and maintain. Moreover, the code is highly generic, since it can still be used with explicit matrices.

We focus on the MATLAB interface of the ASTRA toolbox rather than the optional *Python* interface. MATLAB is commonly used by applied mathematicians who work on new reconstruction methods, linear solvers and sparse reconstruction. Therefore,

many MATLAB templates are available of such algorithms which can benefit from using our opTomo Spot operator.

The paper is structured in six sections. First we give a short introduction to tomography in Section 2. In Section 3 we describe the software elements that are used for implementing the opTomo operator. Several use cases of the opTomo operator are described in Section 4. To give an idea of the efficiency that is achieved by using the opTomo operator, performance benchmarks are discussed in Section 5. Finally, conclusions are drawn in Section 6.

2 Tomography

As an example of the scanning geometry we first introduce the common parallel beam geometry, illustrated in Fig. 1a. In this setup, the detector consists of an array of pixels that measure the radiation intensity along parallel lines. Projections are measured along a range of angles, rotating around the object. The object is subsequently reconstructed from these projections by a tomographic reconstruction algorithm.

Until recently, analytical reconstruction methods such as filtered backprojection (FBP) [35] were used almost exclusively due to their computational efficiency and accuracy, provided that sufficient data is available. Throughout this paper we will focus on *algebraic reconstruction methods* (see, *e.g.*, Chapter 7 of [35] and [3, 12, 15, 17]). These methods are based on a particular discretization of the data in *pixels* and involve algebraic equations for the values of the pixels. This approach offers more flexibility for enforcing constraints on the reconstructed image, in contrast to analytical methods. In algebraic reconstruction methods, the object is assumed to have a constant density in each pixel, which is represented in the image by the *grey value*. The contribution of an object pixel to a detector pixel measurement is proportional to its grey value. In many cases, a weight for the pixel is determined from the length

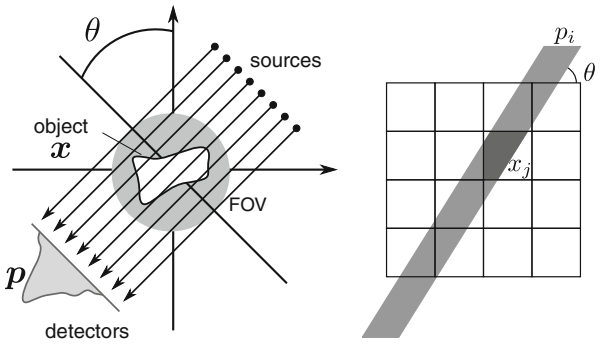


Fig. 1 Tomography with parallel beam geometry. The left image shows the geometry of a typical parallel beam scanner. The image on the right shows the corresponding discretization. The object is represented by an image and a projection is modeled as a linear combination of the pixel values

(line model) or area of intersection of the beam and the pixel (strip model), but there are many other options [11, 23, 26, 32]. The strip model is illustrated in Fig. 1b.

This linear relation between object pixels and a detector pixel measurement is expressed by the *ray sum* or *line projection*

$$p_i = \sum_{j=1}^{N} w_{ij} x_j, \tag{1}$$

where w_{ij} is the weight assigned to image pixel j and detector pixel i. The full set of equations leads to the following linear system:

$$Wx = p. \tag{2}$$

The object $x \in \mathbb{R}^N$ and the projection data $p \in \mathbb{R}^M$ are represented by vectors. The sparse matrix $W \in \mathbb{R}^{M \times N}$, referred to as *projection matrix* or *system matrix*, holds the weights of each pixel.

In many applications, the system in Eq. (2) is underdetermined and the projection matrix does not have full row rank. This results in a challenging ill-posed reconstruction problem. A basic approach for solving it is by minimizing the residual norm, which is referred to as *projection distance*:

$$\underset{x}{\text{minimize}} \; \|Wx - p\|_2^2. \tag{3}$$

If the system is underdetermined, there is no unique solution. Moreover, if noise is present in the projection data, the system of equations can be inconsistent. Regularization techniques should be used to alleviate both of these problems.

3 Software implementation

In this section we discuss the implementation of the `opTomo` operator and introduce the software tools that are used.

3.1 The ASTRA toolbox

The ASTRA toolbox is an open source software package for tomographic reconstruction and algorithm design [28]. The toolbox provides tools and building blocks for the development and implementation of tomographic reconstruction methods. Moreover, it provides many popular reconstruction algorithms, such as filtered backprojection (FBP) and several iterative reconstruction methods such as SIRT and CGLS [8, 15]. The toolbox has a MATLAB and Python interface, which give access to the *forward* and *backprojection* operations. These operations are based on the model in Eq. (2). The forward projection generates projections from an image vector, *i.e.*, this corresponds to multiplying an image vector by the projection matrix W. A backprojection corresponds to multiplication by W^T, the transpose of the projection operator. The ASTRA toolbox uses ray-tracing techniques to compute these matrix-vector products, such that matrices are not stored, but their elements are generated when needed. High memory usage is avoided in this way; only the reconstruction and the

projections should fit in memory (and possibly a few copies, depending on the reconstruction algorithm). The GPU can be used for fast computation of the forward and backprojection steps, allowing large datasets to be processed in reasonable time.

We will not go into detail about the inner workings of ASTRA's MATLAB interface, but rather briefly introduce the main ideas to call the GPU backend. In Listing 1, a utility function is shown for the forward projection algorithm on the GPU through MATLAB's *mex* interface. A few details need to be clarified here. The `data` MATLAB array represents (a slice of) the object. It can be a phantom image or a (partial) reconstruction, from which we want to compute projections. The computation of the projections depends on the particular scanner geometry being used. Therefore, the `proj_geom` structure contains details about the beam type (parallel, fan, cone), the angles at which projections need to be generated and the detector dimensions. The output consists of a data identifier (used internally in ASTRA) and a MATLAB array containing the projection data. Similarly, the volume geometry or `vol_geom` structure contains details about the dimension of the object, from which projections are computed. Because the ASTRA toolbox is very flexible in setting up geometries, many acquisition schemes can be modeled, including circular cone-beam, helical cone-beam, and laminography setups. More details about the geometries are given in Appendix B. For further details about the use of and possibilities of the ASTRA toolbox, we refer the reader to [28].

Listing 1 Utility function for forward projection

```
% forward projection
[id, sinogram] = astra_create_sino_cuda(data, proj_geom, ...
    vol_geom);
```

3.2 The Spot toolbox

For implementing algorithms based on a linear operation, it is very convenient to use matrix-vector notation, as used in MATLAB, because it is similar to the mathematics and results in clean and concise code. However, in many use cases it is not practical to form the matrix corresponding to the linear operation explicitly. As a solution, the Spot toolbox provides a MATLAB framework that wraps linear operations into MATLAB objects that act like matrices [7]. The toolbox introduces a new kind of data type (by using *classes*), called *Spot operators*.

A Spot operator for a linear operation A relies on (external) software implementations of the following operations:

$$y = Ax, \tag{4}$$

$$y = A^T x. \tag{5}$$

The matrix operations listed in Table 1 are overloaded for Spot operators and are based on these basic operations of Eqs. (4) and (5). Most MATLAB *functions* which would not directly support a Spot operator are overloaded as well. One example is the `sum` function. Applying any operation listed in Table 1 to a Spot operator (except for division), does not produce a matrix, but generates another Spot operator of a different type. If the Spot operator is applied to a vector, the result will be computed

Table 1 Operations overloaded by Spot

Matrix operations			
ctranspose	A'	plus	A+B
divide	A\B	subsasgn	A(s1,s2,..,sn) = B
horzcat	[A B]	subsref	A(s1,s2,..,sn)
minus	A−B	transpose	A.'
mldivide	A\B	uminus	−A
mpower	A^i	uplus	+A
mrdivide	A/B	vertcat	[A; B]
mtimes	A*B		

based on the implementation of Eqs. (4) and (5). All other operations can be derived from these. For example, if we want to extract rows or columns from a matrix, we use the parentheses syntax and subscript sets s_1 and s_2 to indicate which rows and columns we want to extract.

Since the matrix elements of a Spot operator A are never stored explicitly, some operations can be slower than expected. For example, if a vector is multiplied by the first row of A (using MATLAB notation), `y =A(1,:)*x`, Spot uses the implementation of Eq. (4) and effectively computes:

```
1    y = A*x;
2    y = y(1);
```

which takes more time if the first row of A could have been formed explicitly. For the same reasons, operations that work element-wise are not implemented for Spot operators, such as element-wise multiplication: `B = A.*A`, or using functions such as `norm(A)`. However, in many typical use cases, such operations can be avoided or if possible can be implemented at a lower level (*e.g.*, in the ASTRA toolbox).

By using the Spot toolbox, MATLAB code that uses matrices can now also be used with Spot operators that are linked to fast, external implementations of operations Eqs. (4) and (5). So, without modification, the same MATLAB code can be used with different implementations of the linear operations.

3.3 The ASTRA Spot operator

For the tomography Spot operator, which we refer to as `opTomo`, we use the forward and backprojection operations from the ASTRA toolbox as implementations of Eqs. (4) and (5), based on the model in Eq. (2).

In code Listing 2 the construction of an `opTomo` object is shown, requiring three arguments:

1. The linear model used to generate W
2. The projection geometry
3. The volume geometry

These arguments are used to set up the forward and backprojection algorithms of ASTRA and to allocate data structures. With the creation of this new Spot operator, we can directly use the matrix operations listed in Table 1. In line 5 of Listing 2 we compute a forward projection of a sample image. We will now explain the internals of these operations in greater detail. The matrix multiplication operation `mtimes` is overloaded for `opSpot` (the superclass from which all Spot operators are derived). Therefore, the image is passed through the `mtimes` function to the `multiply` function of `opTomo`. We implemented the `multiply` function which: reshapes the vector to an image, passes the data to the ASTRA toolbox and calls the forward projection algorithm. Similarly, the backprojection is called if we use the transpose of the `opTomo` operator (also through the `multiply` function). This typical code flow is illustrated in Fig. 2 and shows how the components (MATLAB, Spot, `opTomo`, ASTRA) are connected.

Listing 2 `opTomo` operator

```
1  % Create a tomography Spot operator 'opTomo'
2  W = opTomo('cuda', proj_geom, vol_geom);
3
4  % can be used to create projection data as a vector
5  p = W*im(:);
6
7  % reconstruction using a Krylov subspace method
8  x = lsqr(W,p);
```

Through the `opTomo` operator, we expose the forward and backprojection operations of ASTRA to MATLAB. By choosing the first argument in line 1 of Listing 2 we can choose the model used for the forward and backprojection (to generate W in Eq. (2)). For the CPU projectors, the models `'linear'` [23], `'line'` [32] and

MATLAB
```
p = W*x;
```
↓
Spot
```
% overloaded in superclass
opSpot.mtimes(W,x)
```
↓
opTomo operator
```
% multiply function contains ASTRA code
% input argument two: 1 – no transpose, 2 – transpose
opTomo.multiply(x,1)
```
↓
ASTRA
```
% ASTRA code for forward projection
x = reshape(x, vsize);
% store data in C++
astra_mex_data3d('store', vol_id, x)
% run forward projection
astra_mex_algorithm('iterate', cfg_fp);
% obtain Matlab array
p = astra_mex_data3d('get', sino_id);
```

Fig. 2 Typical code flow of the `opTomo` operator

'strip'[38] are available. If we pass the option 'cuda', the fast GPU projector is used, which is based on the Joseph interpolation kernel [23] for the forward projection and uses a pixel-driven method with linear interpolation for the backprojection [20]. Note that the backprojector in ASTRA is not exactly equivalent to the transpose of the matrix corresponding to the forward projector. This design was chosen to greatly improve performance of the backprojector [27]. As a result, the corresponding matrices of these operators are not fully consistent with Eqs. (4) and (5). However, it was shown that an unmatched forward and backprojector can even improve convergence rates of reconstruction algorithms [19, 37].

Listing 2 illustrates that using the opTomo operator we can compute projection data by using intuitive syntax similar to Eq. (2). The code in line 8 reconstructs the ground truth image from its projections. By using opTomo, the code in Listing 2 is short and stays close to the mathematics, is generic and easy to follow for someone not familiar with the toolbox.

4 Case studies

In the previous sections we have discussed the motivation and implementation details of the opTomo Spot operator. In this section we will demonstrate that the opTomo operator in combination with ASTRA enables the application of MATLAB scripts, ranging from simple scripts to large external packages, to large experimental datasets.

4.1 Custom SIRT implementation

Our first use case is an implementation of SIRT [15] using opTomo. This example demonstrates the simplicity of implementing existing or new algorithms based on their pseudocode. Although SIRT is already implemented in ASTRA, using the code of the current example can have benefits. For example, if we want to use Tikhonov regularization with SIRT we can simply do this by concatenating Spot operators. To see this, note that Tikhonov regularization is based on Eq. (3) with an additional penalty term $\lambda\|x\|_2^2$ on the Euclidean norm of the solution. We can rewrite this problem as:

$$\underset{x}{\text{minimize}} \left\| \begin{pmatrix} W \\ \sqrt{\lambda} I \end{pmatrix} x - \begin{pmatrix} p \\ 0 \end{pmatrix} \right\|_2^2. \tag{6}$$

The corresponding concatenated matrices and right-hand side can now be used as input for SIRT to enable Tikhonov regularization.

The pseudocode of SIRT is listed in Algorithm 1. Note that SIRT converges to a weighted least squares solution [18],

$$x^* = \arg\min_x \|Wx - p\|_R^2 \tag{7}$$

where the norm $\|u\|_R^2 = u^T Ru$, is scaled by inverse row sums.

We use the notation $\mathsf{diag}(x)$ to represent a diagonal matrix with x on its diagonal. SIRT iteratively refines an image vector x by adding a weighted backprojection of the

Algorithm 1 SIRT

Input: Projection data p, projection operator W and initial guess x^0.
Output: Reconstruction x.
Compute inverse column sums:
$c_j = 1/\sum_{i=1}^{M} w_{ij}$ for $j = 1, \ldots, N$

Compute inverse row sums:
$r_i = 1/\sum_{j=1}^{N} w_{ij}$ for $i = 1, \ldots, M$

Let $C = \mathsf{diag}(c)$ and $R = \mathsf{diag}(r)$

for $k = 0, 1, \ldots$ **do**
$u^k = p - Wx^k$
$x^{k+1} = x^k + CW^T Ru^k$
end for

residual projection data. Such an algorithm is presented very compactly in matrix-vector products. As a result, the MATLAB code for SIRT, shown in Listing 3, is almost identical to the pseudocode.

Listing 3 The SIRT algorithm using `opTomo`

```
% To use Tikhonov regularization:
% V = [W;lambda * opEye(size(W,2))];
% p = [p; zeros(size(W,2),1)];
% and use V below instead of W

% determine scaling matrices
r = 1./sum(W,2);
c = 1./sum(W,1);

c(c==Inf) = 0;
r(r==Inf) = 0;

% set up diagonal Spot 'matrices'
C = opDiag(c);
R = opDiag(r);

for i = 1:maxit
    % compute residual
    u = p - W*x;

    % update current solution
    x = x + C*W'*R*u;
end
```

4.2 Cone beam reconstruction

In practice, most tomographic scanners use a point X-ray source that emits a cone-shaped beam, in contrast to a parallel beam. This *cone beam* geometry is also

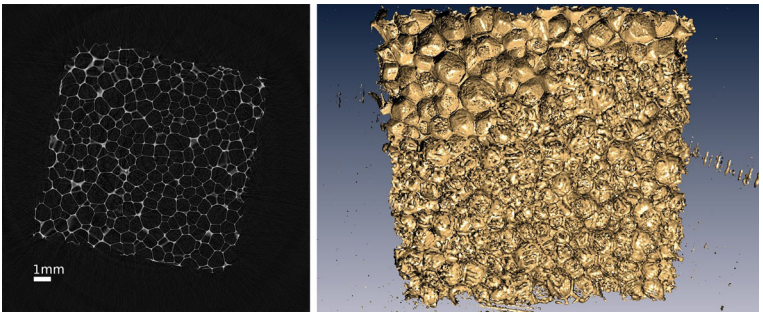


Fig. 3 Cone beam dataset acquired using a Skyscan 1172. The left image shows one slice of the reconstruction of size $1000 \times 1000 \times 524$. On the right an isosurface is rendered in 3D, showing the structure of the pores

supported by the ASTRA toolbox. A detailed description of the geometric parameters is given in Appendix B.

Listing 2 can directly be applied if the `proj_geom` structure has been set up for a cone beam geometry, as the geometry is not hard-coded in the algorithm. To demonstrate that the code is not restricted to small test problems, we have applied it to a large dataset. The dataset consists of projections from a metal foam and are of size 1000×524 taken at 511 angles. The reconstruction grid was $1000 \times 1000 \times 524$. We used LSQR for the reconstruction and stopped the computation after 100 iterations. The central slice and isosurface of the reconstruction are shown in Fig. 3.

For the computation we used a workstation with an NVIDIA Tesla C2070 GPU. The details of the hardware are given in Section 5.1. The computation took 1 hour and 51 minutes, which is about 67 seconds per LSQR iteration. For this dataset, a forward projection takes about 44 seconds and the backprojection takes about 21 seconds.

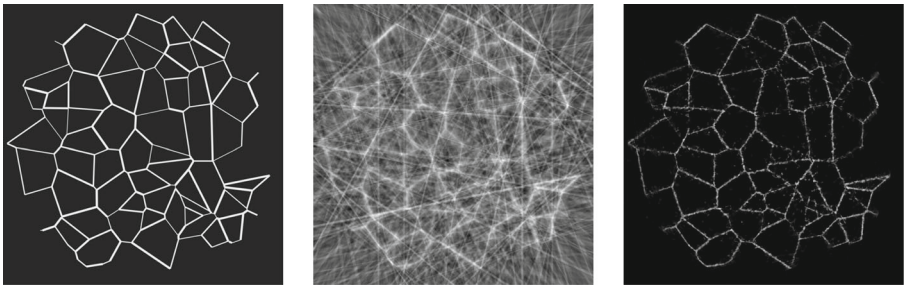
4.3 Sparse image reconstruction

Ideas and methods used in compressive sensing are now commonly applied in tomography. If the object is sparse in a suitable basis, it can often be reconstructed accurately by using ℓ_1-regularization. Algorithms using ℓ_1-regularization are more elaborate to implement compared to linear least squares solvers.

One approach to compute a sparse solution of the algebraic reconstruction problem in Eq. (3) is basis pursuit denoising, formulating the sparse reconstruction problem as minimizing the ℓ_1-norm of the image under consistency conditions:

$$\underset{x}{\text{minimize}} \ \|x\|_1 \text{ subject to } \|Wx - p\|_2 \leq \sigma. \tag{8}$$

The ℓ_1-norm promotes solutions with few nonzeros. The parameter σ is an estimate of the noise level.



(a) phantom (b) LSQR solution (c) SPGL1 solution

Fig. 4 A foam phantom presented as a sparse image. An algebraic reconstruction method LSQR is compared to the method SPGL1. The SPGL1 method exploits the sparsity prior of the solution

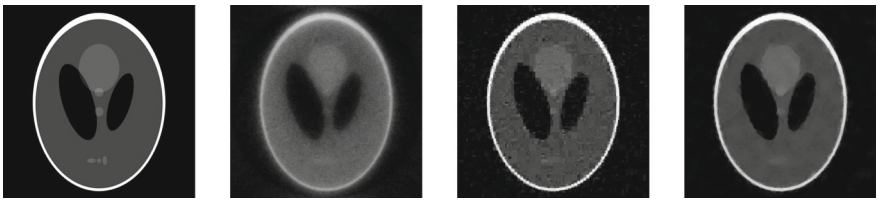
The basis pursuit denoising approach is implemented in the MATLAB package SPGL1, written by Friedlander and van den Berg [6]. This solver is based on matrix-vector products and is therefore suitable for Spot operators without any modifications to the code. In the following experiments we will use SPGL1 for sparse image reconstruction.

In Fig. 4a a foam phantom is shown that has around 7 % of nonzero pixels. This dataset is therefore very suitable for sparse image reconstruction. The ground truth image has dimensions 8192×8192. In total 25 projections were generated using Eq. (2) with a total of 512 detector elements per projection angle. The projection data was reconstructed at an image size of 512×512. In Fig. 4b a least squares solution computed with LSQR is shown. The high angular separation of the projections and the sparse character of the ground truth results in many streak artifacts. Resolving the edges of this reconstruction, *e.g.*, by segmentation, will be difficult. In Fig. 4c a reconstruction using SPGL1 and the `opTomo` operator is shown. This result shows that including sparsity priors during the reconstruction drastically improves the quality of the reconstruction. Moreover, it is easier to resolve the edges of the foam, by segmentation.

For computations we used a workstation with an NVIDIA GTX 570 GPU, the details of the hardware are given in Section 5.1. LSQR was set to stop after 100 iterations or if a relative residual of 0.01 was achieved. In total 11 iterations were needed and the total runtime was 162 ms. The SPGL1 routine was set to a fixed number of 100 iterations, which took 3.5 s. Both the forward- and backprojection took about 4 ms.

4.4 Sparse wavelet reconstruction

The approach of the previous section is not limited to objects that are sparse in a pixel basis, but can also be used with other sparsity priors. For example, images that have few edges and large homogeneous regions with a constant grey value (such as the Shepp–Logan phantom in Fig. 5a), have a sparse representation in a Haar wavelet basis. In this case, we need to minimize the ℓ_1-norm of the wavelet coefficients. Note



(a) phantom (b) LSQR solution (c) wavelet based SPGL1 solution (d) TV-min solution

Fig. 5 Comparison of several reconstruction algorithms. LSQR is an algebraic reconstruction method. The SPGL1 solution exploits sparsity of the ground truth in a Haar wavelet basis. The TV-min solution exploits sparsity of the gradient image

that an image x can be decomposed in its wavelet coefficients y, by using a linear transformation:

$$y = Bx. \tag{9}$$

The matrix B is formed from the basis vectors corresponding to the discrete wavelet decomposition. Because B is unitary for the Haar wavelet, the image can be formed from its wavelet decomposition by multiplying the coefficient vector y with its transpose B^T from the left. The system matrix is adjusted to incorporate the wavelet coefficients:

$$\underset{y}{\text{minimize}} \ \|y\|_1 \text{ subject to } \|WB^T y - p\|_2 \leq \sigma. \tag{10}$$

Note that this approach is the same as the basis pursuit denoising problem in Eq. (8), which is solved with SPGL1. The linear operator involved is now a combination of the wavelet operator and tomography operator. The corresponding Spot operators can be combined in a straightforward manner, as shown in Listing 4. This results in very compact code that is easy to understand from a mathematical perspective.

Listing 4 Combined Spot operator

```
% Projection operator
W = opTomo('cuda', proj_geom, vol_geom);
% 2D wavelet operator
B = opWavelet2(n, n, 'Haar', [], levels);
sigma = 200;
y_spgl1 = spgl1(W*B', sinogram(:), [], sigma);
```

We applied the algorithm on a dataset based on the Shepp–Logan phantom of size 4096×4096. A total of 100 projections were generated with 4096 detector elements per angle. The data was perturbed by applying a moderate amount of Poisson noise to the projection data. The reconstruction size was also 4096×4096.

The results are shown in Fig. 5. Note that we abort LSQR (Fig. 5b) after three iterations, to prevent overfitting (such that it has a similar ℓ_2-norm as the SPGL1 solution). For brevity, we do not provide quantitative details on the noise generation, but the noise level can be observed quite clearly in the LSQR reconstruction. The resulting reconstruction contains substantial noise and details have been blurred. The

wavelet based SPGL1 solution in Fig. 5c is an improvement. Due to the shape of the Haar wavelet, most of the noise is in the detail coefficients that are likely to be suppressed. Although the Haar wavelet is causing block-like artifacts, the edges are better pronounced compared to the least squares solution. For comparison, we also show the results of applying the Chambolle–Pock algorithm for Total Variation (TV) minimization, which will be introduced in the next section. For the Shepp-Logan phantom, TV-minimization appears to be a more suitable prior as shown in Fig. 5d. Both the wavelet and total variation minimization method suppress high gradients and therefore produce reconstructions with less noise, compared to LSQR.

For these computations we used a workstation with a GTX 570 GPU. LSQR took 3.8 s, SPGL1 138 s, Chambolle–Pock 679 s, with a total number of iterations of 3, 30 and 350 respectively. For this dataset the forward- and backprojections took 338 ms and 278 ms.

4.5 TV-minimization using the Chambolle–Pock algorithm

For images consisting of large homogeneous regions, Total Variation based priors are commonly used. Formally, the (anisotropic) total variation of an image is defined as:

$$\mathrm{TV}_{\ell_1}(x) = \sum_{i=1}^{m}\sum_{j=2}^{n} |x_{(i-1)n+j} - x_{(i-1)n+j-1}| + \sum_{i=2}^{m}\sum_{j=1}^{n} |x_{(i-1)n+j} - x_{(i-2)n+j}| \quad (11)$$

We assume that the image vector corresponds to an $m \times n$ image, stored row-wise.

In practice, TV-minimization is often applied in a generic minimization approach where the data consistency term is mixed with the TV-norm:

$$\underset{x\in\mathbb{R}_+^N}{\text{minimize}} \ \frac{1}{2}\|Wx - p\|_2^2 + \lambda \mathrm{TV}_{\ell_1}(x). \quad (12)$$

Note that $\mathbb{R}_+^N$ denotes the set of nonnegative real numbers.

One method for solving this convex minimization problem was presented by Chambolle and Pock [9]. Their approach is to use a primal-dual formulation of the problem. The algorithm they propose is short and has few parameters.

Sidky et al. discussed and elaborated on the Chambolle–Pock TV-minimization method applied to the tomography problem [33]. They present the essential part of the algorithm in four lines of pseudocode. Compared to other TV-minimization algorithms, for example based on FISTA [5], the implementation in a MATLAB environment is straightforward. Because it only uses matrix-vector operations, Spot operators can be used.

In addition to the projection operator, a discrete TV operator based on Eq. (11) is needed, which computes horizontal and vertical differences of an image. The TV operator can either be constructed from a diagonal band matrix, with 1 and -1 on the (sub)diagonals, or it can be formulated as an image processing step using a convolution. In this case, the vertical differences are computed using a convolution of the filter $[-1, 1]$ and the image. In the horizontal difference operator, this filter is simply transposed. To implement the Chambolle–Pock TV-minimization algorithm,

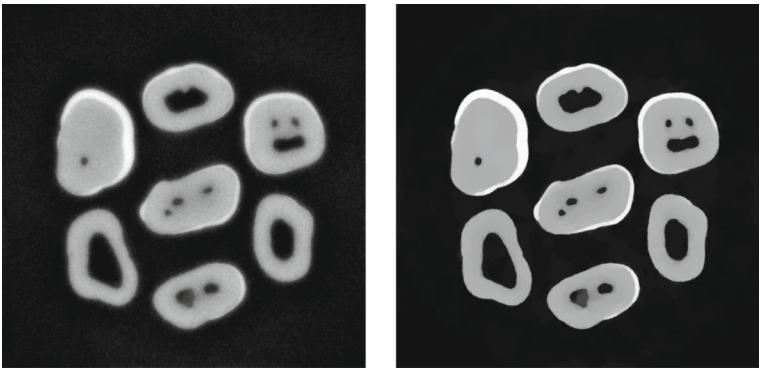
we chose to construct a TV Spot operator `opTV` based on the image convolution, because it is fast.

We applied the algorithm to a dataset from the ESRF synchrotron facility. This dataset was recorded at beamline ID19, which is dedicated for high-resolution diffraction topography. The monochromatic source's energy level was 60 keV. In total 1500 projection images were measured from seven teeth, each image having a resolution of 2048×290. For this dataset, which has been obtained using a high beam intensity, a standard FBP reconstruction can provide very accurate results. To make the reconstruction problem more challenging, Poisson noise was applied to the experimental projection data to simulate a dataset with a low signal-to-noise ratio. TV-minimization is not only an effective technique to apply on limited angle datasets, but should enhance the quality of reconstructions for noisy datasets as well.

For this example, we focus on the reconstruction of a single slice and run the Chambolle–Pock algorithm for 200 iterations. Even for a single slice, the Chambolle–Pock algorithm would require at least 23 GB memory if matrices were formed explicitly (in single precision). Therefore, on most workstations this MATLAB code would not have been applicable to this dataset without using the ASTRA toolbox. And without using the `opTomo` and `opTV` operators, the implementation would have required substantially more effort. The reconstruction was run on a workstation with GTX 570 GPU (see Section 5.1 for details of the hardware) and took 248 seconds. The forward and backprojections took 213 ms and 245 ms respectively. The TV Spot operator `opTV` does not employ the GPU, but is performed in MATLAB. Multiplication by this operator takes approximately 80 ms.

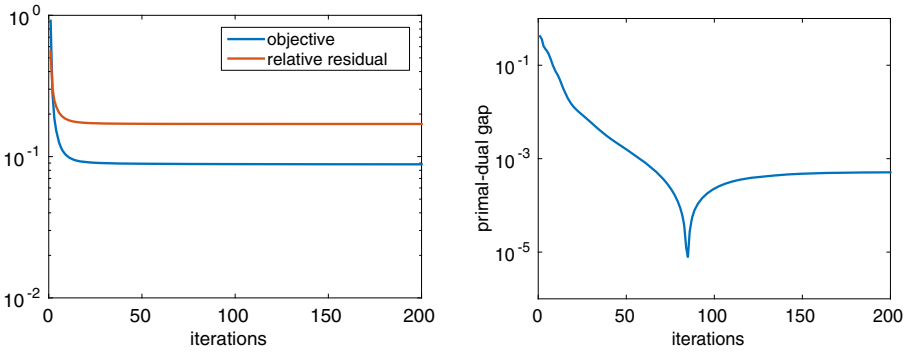
The reconstructions are shown in Fig. 6. The sparse gradient prior results in a much sharper reconstruction that has more homogeneous areas of constant grey values. Moreover, the background is completely black, due to the nonnegativity constraint specified in Eq. (12) and supported by the Chambolle–Pock algorithm. Also the noise does not affect the reconstruction as much as it does in a least squares solution.

To assess the convergence of the Chambolle–Pock algorithm we have plotted the relative residual and the objective function from Eq. (12) in Fig. 7a. In Fig. 7b,



(a) LSQR solution (b) TV-min solution

Fig. 6 Reconstructed slice for the ESRF teeth dataset



(a) convergence of relative residual and objective (b) absolute value of the primal-dual gap

Fig. 7 Convergence results of the Chambolle–Pock algorithm applied to the experimental teeth dataset from ESRF. The primal-dual gap starts out negative and around 80 iterations becomes positive which explains the dip in the absolute value of the primal-dual gap (on a logarithmic scale)

the (conditional) primal-dual gap is shown [33]. Note that the relative residual does not converge to zero, due to the noise in the projection data. The primal-dual gap is initially negative, but becomes positive after about 80 iterations and converges to 5×10^4.

This example illustrates that using the Spot operator, the pseudocode given in the paper from Sidky *et al.* [33], can directly benefit from the fast GPU ASTRA back end, which enables application of Chambolle–Pock to a real experimental dataset.

5 Performance benchmarks

In this section we show benchmarks of the GPU and CPU code of the ASTRA toolbox in combination with the Spot operator. Since the memory use of explicit matrices is the key limitation in standard MATLAB code, we compare memory usage of the major components of the forward projection operator, such as copies of volume data and projection data, with that of explicit MATLAB matrices. Finally, we measure the computational overhead that is introduced by the Spot toolbox, when using the `opTomo` operator

5.1 The forward and backprojection operations

The code that we run as a benchmark is line 5 of Listing 2. All the benchmarks were timed using the `timeit` function of MATLAB, which takes care of "warming up" the CPU and GPU. The wall clock time is averaged over the total number of runs, which is chosen automatically by `timeit`. The machine we used for benchmarking was a Linux workstation with an Intel Core i7-2600K@ 3.4 GHz CPU with 16 GB of system RAM and a NVIDIA GTX 570 GPU. We compared the results with a TESLA C2070 GPU in a server machine with Intel Core i7-3930K@ 3.2 GHz and 64 GB of system RAM. We report results only for the Linux version of the software.

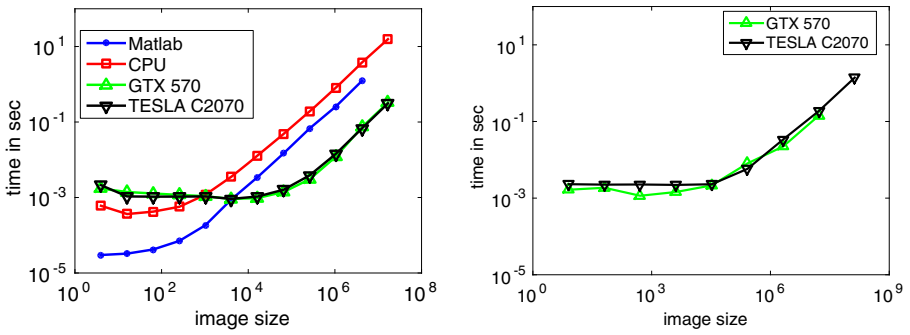
For a similar workstation running the Windows operating system (also supported by the ASTRA toolbox), similar results were observed. A pre-release of the ASTRA toolbox version 1.6 was used.

First we explicitly form the system matrix as a sparse MATLAB matrix. This can be done using a utility function in ASTRA. Then we also time the code for two `opTomo` operators: one of type `'cuda'`, and one of type `'linear'`. Both of these generate the matrix elements on the fly based on the slice-interpolation kernel [23]. The CUDA version uses the GPU, while the other uses the CPU code of the ASTRA toolbox. The Shepp–Logan phantom was used in both 2D and 3D cases, with sizes $n \times n$ and $n \times n \times n$ respectively. A slice of the phantom is shown in Fig. 5a. For each experiment, a square (or line in 2D) detector was used that matches the width and height of the phantom in combination with a parallel beam geometry. The number of angles was fixed at 100.

In Fig. 8, the timings of the 2D forward projection are shown for different sizes of the phantom. We were able to use explicit matrices in MATLAB up to phantom sizes of 2048×2048, above which the explicit system matrix no longer fits in memory. For these data, 9 GB of memory was required. The ASTRA CPU code is somewhat slower than the use of explicit matrices, which is expected due to the need to generate matrix elements and overhead from the Spot operator. Also note that the GPU code outperforms the CPU code for image sizes larger than 32×32.

In the 3D case it is not practical to form matrices and we omit the use of explicit MATLAB matrices. Instead we compare two different GPUs, the GTX 570 and TESLA C2070. The results show that both cards are performing similarly, but the TESLA card can reconstruct a larger volume of $512 \times 512 \times 512$, since it has more memory (5.4 GB compared to 1.3 GB).

The estimated memory use of the forward projection is listed in Table 2 (in terms of data elements). It has been determined as follows: for the forward projection, the volume and its projections need to be stored. For the 2D case, this is n^2 for the volume and kn for the projections, where k is the number of angles. Similarly, the 3D volume consists of n^3 voxels and projections are kn^2. For the MATLAB code, additionally



(a) 2D (b) 3D

Fig. 8 Performance of the forward projection operator. The image size represents the total number of pixels/voxels in the volume

Table 2 Memory use of the forward projection operation

	MATLAB	ASTRA CPU	ASTRA GPU
2D	$(3k+1)n^2 + kn$	$n^2 + kn$	$2n^2 + kn$
3D	$(3k+1)n^3 + kn^2$	$n^3 + kn^2$	$2n^3 + kn^2$

the matrix should be stored, which has the same number of rows as detector measurements and the same number of columns as volume pixels/voxels. If we assume that at most $3n$ voxels have a nonzero contribution to a detector measurement, this adds $3n$ times the number of detector elements to the storage requirements. In Fig. 9 we have plotted the memory use corresponding to Table 2 for single precision floats. Note that the GPU code uses twice the storage for the input of the forward projection algorithm (which is the volume itself). This is because GPU textures are used to speed up data access.

We also benchmarked the backprojection operation of the ASTRA toolbox using the Spot operator. The results are shown in Fig. 10 and they are comparable to the timings of the forward projection. The memory use of the backprojection operation is the same as the forward projection, except that the GPU code needs to store the input twice and output once. For the backprojection, the input corresponds to the projection data, which is usually somewhat smaller than the volume.

Nowadays, CPU memory is limited to several 10s of gigabytes in a regular workstation, or up to hundreds of gigabytes in a high-end server. For example, 32 GB RAM limits the image size to approximately $300 \times 300 \times 300$ for explicit matrices. Whereas GPU RAM is limited to 12 GB of RAM. However, since the memory requirements of the GPU code are lower, the maximum image size is now $900 \times 900 \times 900$, for this amount of RAM. We see a clear improvement in the maximum size of datasets that can be handled by a single GPU.

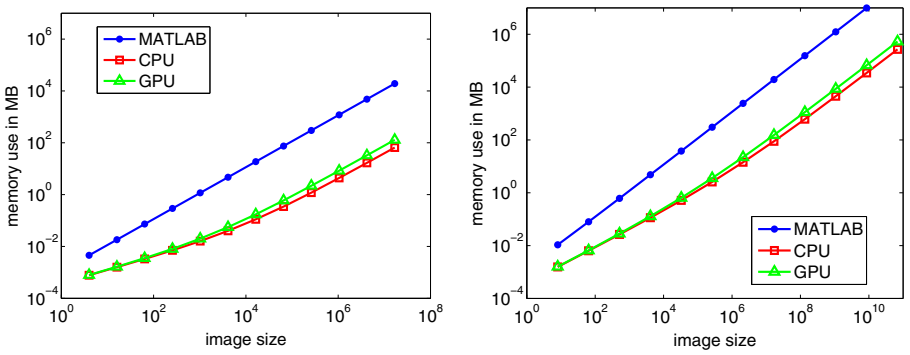
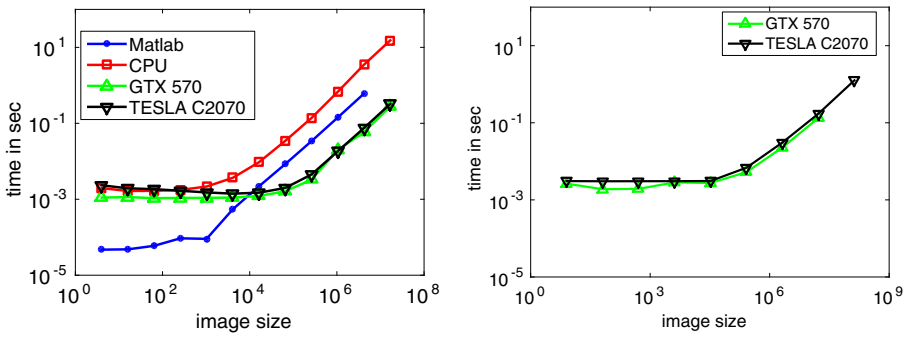


Fig. 9 Memory use of the forward projector. The image size represents the total number of pixels/voxels in the volume



(a) 2D (b) 3D

Fig. 10 Performance of the backprojection operator. The image size represents the total number of pixels/voxels in the volume

5.2 Overhead of the Spot toolbox

The use of the Spot operator `opTomo` results in computational overhead due to the Spot toolbox, compared to using the ASTRA toolbox directly. In order to quantify this overhead, we benchmark the LSQR algorithm as provided by MATLAB using three different ways to call it:

1. using `opTomo`
2. a version using ASTRA's MATLAB interface optimized for code length, referred to as *ASTRA util*,
3. a version using ASTRA's MATLAB interface optimized for speed, referred to as *ASTRA optim*.

LSQR allows the use of a *function handle* instead of a matrix. The function handle, `afun`, is specified, such that `afun(x,'notransp')` returns `A*x` and `afun(x,'transp')` returns `A'*x`. The second variant of LSQR (ASTRA util) uses a function handle of the ASTRA utility functions for the forward and backprojection, as show in Listing 5. This can be done with a few lines of code and is a straightforward implementation which is most likely used in practice. However, these functions allocate memory and set up the algorithm whenever they are called, which is every LSQR iteration.

In the third variant (ASTRA optim) we use an implementation at a lower level that preallocates memory and sets up the algorithm, the same as is done in the operator `opTomo` internally. However, this optimized code requires substantially more lines of code. Because the ASTRA code in this version is the same as used in `opTomo`, it allows us to see the exact overhead caused by the Spot toolbox.

In Table 3 and Table 4 the computation times of the three variants of LSQR using 20 iterations are shown for various sizes of the volume/detector. The same settings are used as in the previous paragraph. The code was run on the workstation with the GTX 570 GPU.

Table 3 2D: LSQR runtimes in seconds

n	ASTRA optim	ASTRA util	opTomo
32	0.020	0.039	0.040
64	0.022	0.042	0.044
128	0.036	0.057	0.060
256	0.085	0.112	0.111
512	0.278	0.308	0.306
1024	1.137	1.140	1.178
2048	4.885	5.030	5.040

In the 2D case, shown in Table 3, ASTRA util and opTomo have comparable run times. If we look at the overhead caused by Spot by comparing ASTRA optim and opTomo we see that for small n the overhead is large. For $n = 32$, LSQR using opTomo is twice as slow compared to LSQR using function handle ASTRA optim. However, the total overhead is only 20 ms. If the amount of work is increased the relative overhead becomes drastically smaller. For a data size of $n = 2048$, the overhead of the opTomo operator compared to ASTRA optim is 155 ms. Relative to the total runtime, this overhead is 3 %.

In 3D, see Table 4, the amount of work compared to the overhead becomes large. In this case, the ASTRA utility functions are slower than opTomo due to overhead from memory allocation and initializing the forward and backprojection algorithms. The overhead of opTomo compared to ASTRA optim is still in the order of 20 ms. Therefore, the relative overhead with respect to total runtime is almost negligible in 3D. For $n = 256$, the relative overhead is 0.2 %.

In Listings 5 and 6 we compare the amount of code that is needed for ASTRA util and opTomo. We omit the code used for ASTRA optim, since it requires about 50 lines of code. From these code snippets it is clear that the opTomo operator can hide a lot of code and interface details that were necessary for implementing ASTRA util. Moreover, the optimizations used in ASTRA optim are also part of the opTomo Spot operator.

Table 4 3D: LSQR runtimes in seconds

n	ASTRA optim	ASTRA util	opTomo
32	0.138	0.176	0.158
64	0.408	0.444	0.437
128	2.379	2.519	2.434
256	15.615	16.506	15.643

Listing 5 LSQR ASTRA util

```
% set up data size (100 projection angles)
vsize = [n,n,n];
psize = [n,100,n];
% set up function handle
f = @(x, type) Afun2(x, type, vsize, psize, proj_geom, ...
    vol_geom);
% create forward projections
p = f(im, 'notransp');
% solve with lsqr
x = lsqr(f, p);

..

% used for function handle f
function y = Afun2(x, type, vsize, psize, proj_geom, vol_geom)
  if strcmp(type, 'notransp')
    % vector to volume
    x = reshape(x, vsize);
    % fp
    [y_id,y] = astra_create_sino3d_cuda(x, proj_geom, vol_geom);
    astra_mex_data3d('delete', y_id);
  else
    % vector to projections
    x = reshape(x, psize);
    % bp
    [y_id, y] = astra_create_backprojection3d_cuda(x, ...
        proj_geom, vol_geom);
    astra_mex_data3d('delete', y_id);
  end
  y = y(:);
end
```

Listing 6 LSQR `opTomo`

```
% create Spot operator
W = opTomo('cuda', proj_geom, vol_geom);
% create forward projections
p = W*im(:);
% solve with lsqr
x = lsqr(W,p);
```

6 Discussion and Conclusions

Advances in hardware for tomographic projection acquisition have led to an increase in data sizes. At the same time, advances in computational methods for limited data reconstruction have resulted in a broad range of algorithms that are powerful, yet highly computationally demanding. As a result, reconstruction software has to be implemented for parallel computation architectures such as computer clusters and graphics processing units (GPUs) to handle large-scale datasets efficiently.

Many novel reconstruction algorithms are prototyped in a high-level scripting language such as MATLAB. The syntax for these languages can be similar to

mathematical notation, which makes prototyping easier. Due to the nature of high-level languages, these implementations are often not suitable to apply on large-scale datasets. For tomographic datasets, the system matrix corresponding to the linear model of the forward projection $Wx = p$, is too large to store explicitly for even moderately sized datasets. The `opTomo` operator is able to bridge the gap between the flexibility of MATLAB scripts on one hand, and the fast and scalable GPU back end of the ASTRA toolbox. The `opTomo` operator allows using matrix syntax to call the fast GPU projection and backprojection implementations of the ASTRA toolbox. By overloading many common matrix operations, the Spot toolbox delivers an effective framework for linear operators.

We remark that the `opTomo` operator exposes ASTRA's highly efficient implementations of the forward and backprojection operations, but when more detailed access to the matrix is required, the user may find that certain operations are either computationally inefficient or simply not implemented. This includes operations that work element-wise on the matrix, such as element-wise multiplication. Therefore, MATLAB scripts that rely on such operations, for example to compute column or row norms, are currently not supported. Usually, it is possible to work around these limitations and as the ASTRA toolbox is continuously evolving, efforts are currently ongoing to extend the range of operations for which the `opTomo` operator provides a high level of efficiency.

In our benchmarks we have seen that a very small overhead is paid by using the Spot operator for ASTRA. As the data sizes increase the relative overhead becomes negligible and should not pose any problems.

Our software is freely available under an open source license (GPL), enabling easy implementation of novel advanced reconstruction algorithms in materials science, biomedical imaging, and other fields.

Acknowledgments This research was supported by the Netherlands Organisation for Scientific Research (NWO), programme 639.072.005. We would like to thank Paul Tafforeau of the European Synchrotron Radiation Facility (ESRF), Grenoble, France, for providing the dataset of the teeth, recorded at beamline ID19. Networking support was provided by the EXTREMA COST Action MP1207.

Appendix A: Availability of source code and data

The ASTRA toolbox and `opTomo` operator can be downloaded as open source software [1]. The Spot toolbox is available separately [7]. The MATLAB code of all the examples in this paper and the corresponding datasets can be obtained from https://github.com/astra-toolbox/astra-toolbox/.

Appendix B: Geometries in the ASTRA toolbox

In this section we describe the use of volume and projection geometries for the ASTRA toolbox. Since the 2D geometries can be embedded into a 3D geometry, we will not discuss these separately.

B.1 Volume geometry

The volume geometry describes the dimensions of the reconstruction area in terms of voxels. The reconstruction volume is always centered around the origin, and its voxels are cubes of unit size. This defines the Cartesian coordinate system in which the rest of the geometry is specified. To set up a reconstruction volume we can use the following MATLAB code:

```
vol_geom = astra_create_vol_geom(vx, vy, vz);
```

which results in a structure that contains the size of the reconstruction volume.

B.2 3D parallel beam

The parallel beam geometry for the 3D case is similar to the 2D case, except that the detector is two dimensional.

In the ASTRA toolbox this geometry can be specified in one of two ways:

```
proj_geom = astra_create_proj_geom('parallel3d', ...
    det_spacing_x, det_spacing_y, det_row_count, ...
    det_col_count, angles);
proj_geom_vec = astra_create_proj_geom('parallel3d_vec', ...
    det_row_count, det_col_count, vectors);
```

In the first case, a circular path (rotating around the z-axis) of the source and detector is used. The parameters `det_spacing_x` and `det_spacing_y` specify the distance between two adjacent detector pixels. The `det_row_count` and `det_col_count` determine the number of rows and columns of the detector. The `angle` array contains all angles in radians at which projections are measured.

In the second case, besides the number of rows and columns of the detector, an array `vectors` is passed. This array has K rows, one for each angle. A row contains the following parameters in order:

- `rayX`, `rayY`, `rayZ`. This vector gives the direction of the rays.
- `dx`, `dy`, `dz`. These are the x, y and z coordinates of the center of the detector.
- `ux`, `uy`, `uz`. The vector from detector pixel $(0, 0)$ to $(0, 1)$.
- `vx`, `vy`, `vz`. The vector from detector pixel $(0, 0)$ to $(1, 0)$.

In Fig. 11, we show an example of the geometric parameters. In this example, a projection is taken at an angle of 0°. Note that this specifies the acquisition of one projection image and therefore, the parameters for all other angles should be passed as well. Using the vectors d and ray, many projection acquisition schemes can be parametrized.

Note that it is not necessary for the detector and volume pixel to have the same size, although this setting is commonly used for reconstructions.

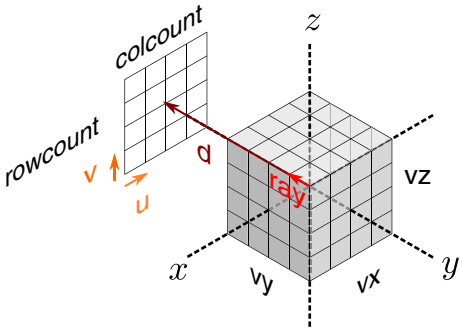


Fig. 11 Illustration of the parameters for the volume and 3D parallel beam projection geometries

B.3 Cone beam

Setting up a cone beam geometry is very similar to setting up a 3D parallel beam geometry. In this case, the rays are not parallel, but they originate from a single point: the ray source. Therefore, instead of indicating the ray direction, the center of the ray source is passed.

```
1    proj_geom = astra_create_proj_geom('cone', det_spacing_x, ...
         det_spacing_y, det_row_count, det_col_count, angles, ...
         source_origin, origin_det);
2    proj_geom = astra_create_proj_geom('cone_vec', ...
         det_row_count, det_col_count, vectors);
```

References

1. ASTRA - Tomographic Reconstruction toolbox. http://sourceforge.net/p/astra-toolbox/. [Online; accessed 1-March-2015] (2015)
2. Agulleiro, J.I., Fernandez, J.J.: Fast tomographic reconstruction on multicore computers. Bioinformatics **27**(4), 582–583 (2011)
3. Andersen, A.H., Kak, A.C.: Simultaneous algebraic reconstruction technique (SART): a superior implementation of the ART algorithm. Ultrason. Imaging **6**(1), 81–94 (1984)
4. Batenburg, K.J., Sijbers, J.: DART: A practical reconstruction algorithm for discrete tomography. IEEE Trans. Image Process **20**(9), 2542–2553 (2011)
5. Beck, A., Teboulle, M.: A fast iterative shrinkage-thresholding algorithm for linear inverse problems. SIAM. J. Imaging Sci. **2**(1), 183–202 (2009)
6. van den Berg, E., Friedlander, M.P.: Probing the Pareto frontier for basis pursuit solutions. SIAM J. Sci. Comput. **31**(2), 890–912 (2008)
7. van den Berg, E., Friedlander, M.P.: Spot - a linear-operator toolbox., http://www.cs.ubc.ca/labs/scl/spot/ (2014). [Online; accessed 1-July-2014]
8. Björck, Å., Elfving, T.: Accelerated projection methods for computing pseudoinverse solutions of systems of linear equations. BIT **19**(2), 145–163 (1979)
9. Chambolle, A., Pock, T.: A first-order primal-dual algorithm for convex problems with applications to imaging. J. Math. Imaging Vision **40**(1), 120–145 (2011)
10. Chilingaryan, S., Mirone, A., Hammersley, A., Ferrero, C., Helfen, L., Kopmann, A., dos Santos Rolo, T., Vagovic, P.: A GPU-based architecture for real-time data assessment at synchrotron experiments. IEEE Trans. Nucl. Sci. **58**(4), 1447–1455 (2011)

11. De Man, B., Basu, S.: Distance-driven projection and backprojection in three dimensions. Phys. Med. Biol. **49**(11), 2463–2475 (2004)
12. Dempster, A.P., Laird, N.M., Rubin, D.B.: Maximum likelihood from incomplete data via the EM algorithm. J. R. Stat. Soc. B Methodol., 1–38 (1977)
13. Donoghue, P.C.J., Bengtson, S., Dong, X., Gostling, N.J., Huldtgren, T., Cunningham, J.A., Yin, C., Yue, Z., Peng, F., Stampanoni, M.: Synchrotron X-ray tomographic microscopy of fossil embryos. Nature **442**(7103), 680–683 (2006)
14. Figueiredo, M.A.T., Nowak, R.D., Wright, S.J.: Gradient projection for sparse reconstruction: Application to compressed sensing and other inverse problems. IEEE J. Sel. Topics Signal Process **1**(4), 586–597 (2007)
15. Gilbert, P.: Iterative methods for the three-dimensional reconstruction of an object from projections. J. Theoret. Biol. **36**(1), 105–117 (1972)
16. Goldstein, T., Osher, S.: The split Bregman method for L1-regularized problems. SIAM J. Imaging Sci. **2**(2), 323–343 (2009)
17. Gordon, R., Bender, R., Herman, G.T.: Algebraic reconstruction techniques (ART) for three-dimensional electron microscopy and X-ray photography. J. Theoret. Biol. **29**(3), 471–481 (1970)
18. Gregor, J., Benson, T.: Computational analysis and improvement of SIRT. IEEE Trans. Med. Imag. **27**(7), 918–924 (2008)
19. Guedouar, R., Zarrad, B.: A comparative study between matched and mis-matched projection/back projection pairs used with ASIRT reconstruction method. Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers. Detectors and Associated Equipment **619**(1), 225–229 (2010)
20. Guedouar, R., Zarrad, B.: A new reprojection method based on a comparison of popular reprojection models. Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers. Detectors and Associated Equipment **619**(1), 270–275 (2010)
21. Hu, Q., Ley, M.T., Davis, J., Hanan, J.C., Frazier, R., Zhang, Y.: 3D chemical segmentation of fly ash particles with X-ray computed tomography and electron probe microanalysis. Fuel **116**, 229–236 (2014)
22. Jang, B., Kaeli, D., Do, S., Pien, H.: Multi GPU implementation of iterative tomographic reconstruction algorithms. In: Biomedical Imaging: From Nano to Macro, 2009. ISBI'09. IEEE International Symposium on, pp. 185–188. IEEE (2009)
23. Joseph, P.: An improved algorithm for reprojecting rays through pixel images. IEEE Trans. Med. Imag. **1**(3), 192–196 (1982)
24. Kremer, J.R., Mastronarde, D.N., McIntosh, J.R.: Computer visualization of three-dimensional image data using IMOD. J. Struct. Biol. **116**(1), 71–76 (1996)
25. Kübel, C., Voigt, A., Schoenmakers, R., Otten, M., Su, D., Lee, T., Carlsson, A., Bradley, J.: Recent advances in electron tomography: TEM and HAADF-STEM tomography for materials science and semiconductor applications. Microsc. Microanal. **11**(5), 378–400 (2005)
26. Lewitt, R.M.: Alternatives to voxels for image representation in iterative reconstruction algorithms. Phys. Med. Biol. **37**(3), 705–716 (1992)
27. Palenstijn, W.J., Batenburg, K.J., Sijbers, J.: Performance improvements for iterative electron tomography reconstruction using graphics processing units (GPUs). J. Struct. Biol. **176**(2), 250–253 (2011)
28. Palenstijn, W.J., Batenburg, K.J., Sijbers, J.: The ASTRA tomography toolbox. In: 13th International Conference on Computational and Mathematical Methods in Science and Engineering, CMMSE (2013)
29. Pedemonte, S., Bousse, A., Erlandsson, K., Modat, M., Arridge, S., Hutton, B.F., Ourselin, S.: GPU accelerated rotation-based emission tomography reconstruction. In: Nuclear Science Symposium Conference Record (NSS/MIC), pp. 2657–2661. IEEE (2010)
30. Rivers: M.L.: tomoRecon: High-speed tomography reconstruction on workstations using multi-threading. In: Proc. SPIE 8506, Developments in X-Ray Tomography VII, p. 85060U (2012)
31. Schüle, T., Schnörr, C., Weber, S., Hornegger, J.: Discrete tomography by convex–concave regularization and D.C. programming. Discret. Appl. Math. **151**(1), 229–243 (2005)
32. Siddon, R.L.: Fast calculation of the exact radiological path for a three-dimensional CT array. Med. Phys. **12**(2), 252–255 (1985)

33. Sidky, E.Y., Jørgensen, J.H., Pan, X.: Convex optimization problem prototyping for image reconstruction in computed tomography with the Chambolle–Pock algorithm. Phys. Med. Biol **57**(10), 3065–3091 (2012)
34. Sidky, E.Y., Pan, X.: Image reconstruction in circular cone-beam computed tomography by constrained, total-variation minimization. Phys. Med. Biol. **53**(17), 4777–4807 (2008)
35. Slaney, M., Kak, A.: Principles of computerized tomographic imaging. Society for Industrial and Applied Mathematics (1988)
36. Thielemans, K., Tsoumpas, C., Mustafovic, S., Beisel, T., Aguiar, P., Dikaios, N., Jacobson, M.W.: STIR: software for tomographic image reconstruction release 2. Phys. Med. Biol. **57**(4), 867–883 (2012)
37. Zeng, G.L., Gullberg, G.T.: Unmatched projector/backprojector pairs in an iterative reconstruction algorithm. IEEE Trans. Med. Imaging **19**(5), 548–555 (2000)
38. Zhu, J., Li, X., Ye, Y., Wang, G.: Analysis on the strip-based projection model for discrete tomography. Discrete Applied Mathematics **156**(12), 2359–2367 (2008)