CWI

Centrum Wiskunde & Informatica

Tomosipo tutorial

Dirk Schut

21 June 2021

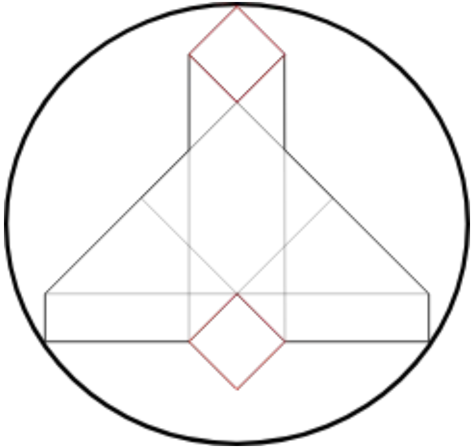
This presentation

1. What is Tomosipo
2. Installing Tomosipo
3. Three examples
   a. Projection operator in simple 2D geometry
   b. Unconventional geometry and visualization
   c. FleX-ray FDK reconstruction
4. Deep learning
5. Questions

All scripts are available on git: https://github.com/D1rk123/tomosipo_examples

What is Tomosipo?

- CT projection library in Python
  - Astra backend
- Pytonic interface
- Flexible in defining geometries
- Good integration with other libraries



Interacting libraries

- PyTorch compatibility
  - Use Tomosipo operators in neural networks
  - Very fast CPU/GPU operations from pytorch available
- Reconstruction algorithms available in ts_algorithms repository
  - Implemented in Python -> flexible and readable
  - Using PyTorch -> fast
- Visualisation options
  - Render your geometry to mp4 or svg file
  - Visualize data using PyQtGraph

My experience with Tomosipo

- Implemented Noise2Inverse neural network variations for Allard at the start of my PhD
- Used Tomosipo for my own research on carousel geometry CT
- Contributed FDK-implementation to ts_algorithms
    - Optimized this code to handle big FleX-Ray datasets

My experience with Tomosipo

- Implemented Noise2Inverse neural network variations for Allard at the start of my PhD
- Used Tomosipo for my own research on carousel geometry CT
- Contributed FDK-implementation to ts_algorithms
    - Optimized this code to handle big FleX-Ray datasets

- Disclaimer: I haven’t used Astra/ODL/Flexbox, so I can’t compare

Installing Tomosipo

- Tomosipo and ts_algorithms can be installed using pip
- Required packages:
  - Latest Astra 1.9.x development version
  - For ts_algorithms: pytorch version >= 1.7
  - For visualisation: ffmpeg, ffmpeg-python, pyqtgraph, pyqt, pyopengl

Installing Tomosipo

- Tomosipo and ts_algorithms can be installed using pip
- Required packages:
  - Latest Astra 1.9.x development version
  - For ts_algorithms: pytorch version >= 1.7
  - For visualisation: ffmpeg, ffmpeg-python, pyqtgraph, pyqt, pyopengl
- Both Pytorch and Astra rely on the cudatoolkit package:
  - at most version 10.1 for the cluster
  - at least version 11.0 for RTX3070 cards in new workstations
  - at most version 11.0 for Astra
  - version 11.0 not supported in pytorch 1.8 so use pytorch 1.7 instead

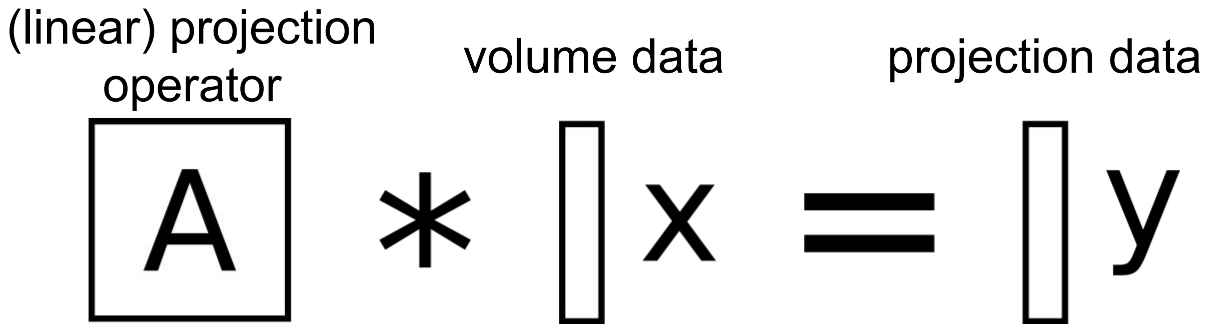
Conda installation script

11.0 for RTX30XX cards

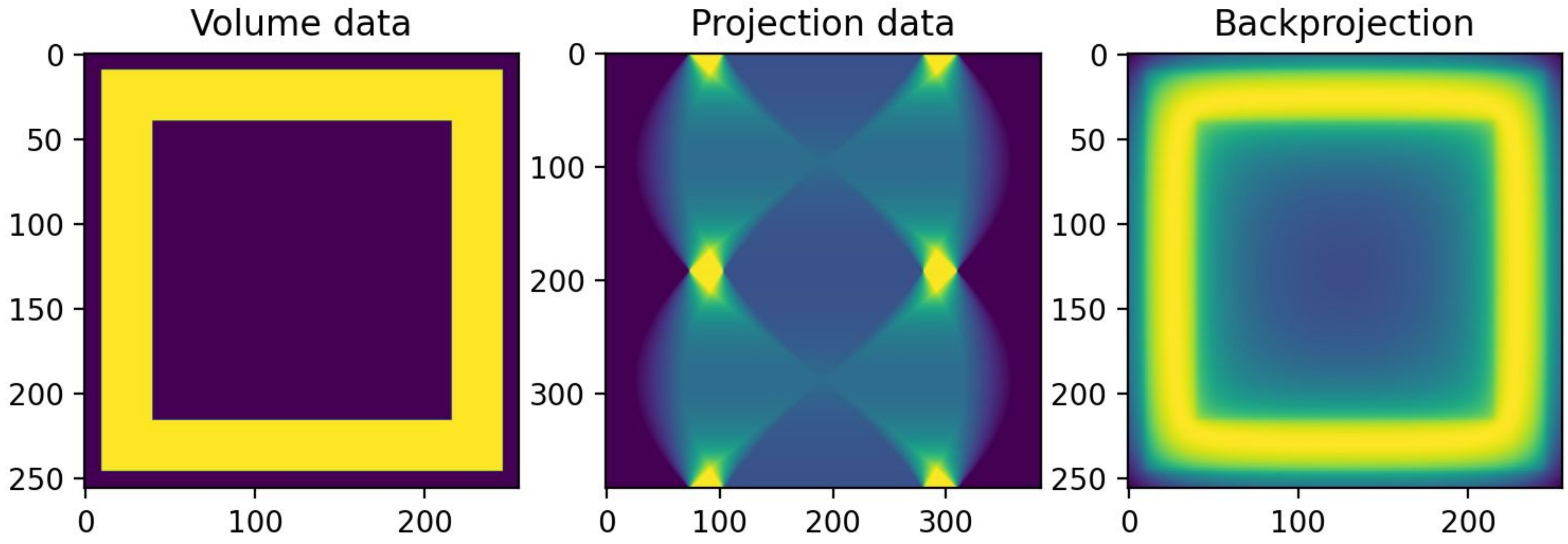
```
conda create -n to_ex python=3.8 cudatoolkit=10.1 pytorch=1.7
astra-toolbox numpy scikit-image matplotlib tifffile tqdm ffmpeg
ffmpeg-python pyqtgraph pyqt pyopengl -c pytorch -c
astra-toolbox/label/dev -c defaults -c conda-forge

conda activate to_ex

pip install git+https://github.com/ahendriksen/tomosipo.git@develop

pip install git+https://github.com/ahendriksen/ts_algorithms.git
```

Geometries in tomosipo

- Volume geometry specifies the volume
  - Properties: size, position, resolution
- Projection geometry specifies the projection setup
  - Type: Cone or parallel beam
- Geometries are always 3D
  - [z, y, x] coordinates are used for volume data, and [height, time, width] for projection data, so the first dimension can be 1 for 2D setups
- Operator can be derived from a volume and projection geometry



```python
import numpy as np
import tomosipo as ts
from matplotlib import pyplot as plt

# Setup 2D volume and parallel projection geometry
vg = ts.volume(shape=(1, 256, 256))
pg = ts.parallel(angles=384, shape=(1, 384))

# Create an operator from the geometries
A = ts.operator(vg, pg)

# Create hollow cube phantom
x = np.zeros(A.domain_shape)
x[:, 10:-10, 10:-10] = 1.0
x[:, 40:-40, 40:-40] = 0.0

# Project the volume data to obtain the projection data and backproject it again
y = A(x)
b = A.T(y)

plt.figure(figsize=(9, 3))
plt.subplot(131); plt.imshow(x[0, ...]); plt.title("Volume data")
plt.subplot(132); plt.imshow(y[0, ...]); plt.title("Projection data")
plt.subplot(133); plt.imshow(b[0, ...]); plt.title("Backprojection")
plt.show()
```


Volume data
Projection data
Backprojection
0
50
100
150
200
250
0
100
200
300
0
100
200
300
0
50
100
150
200
250
0
100
200

Making geometries just as complex as you need to

- All geometries have a standard form and a more flexible vector form
  - Standard forms assume a still volume and a rotating detector setup
  - In vector form the orientation can change for every projection
- Standard geometries have good defaults
  - If only shape is provided, 1x1x1 voxels are assumed
  - If no position is defined, position at the origin is assumed
- Vector form geometries can be created in two ways
  - Create a vector geometry directly by providing the parameters as a vector over time
  - Convert a standard geometry and apply a transformation

Visualizing geometries

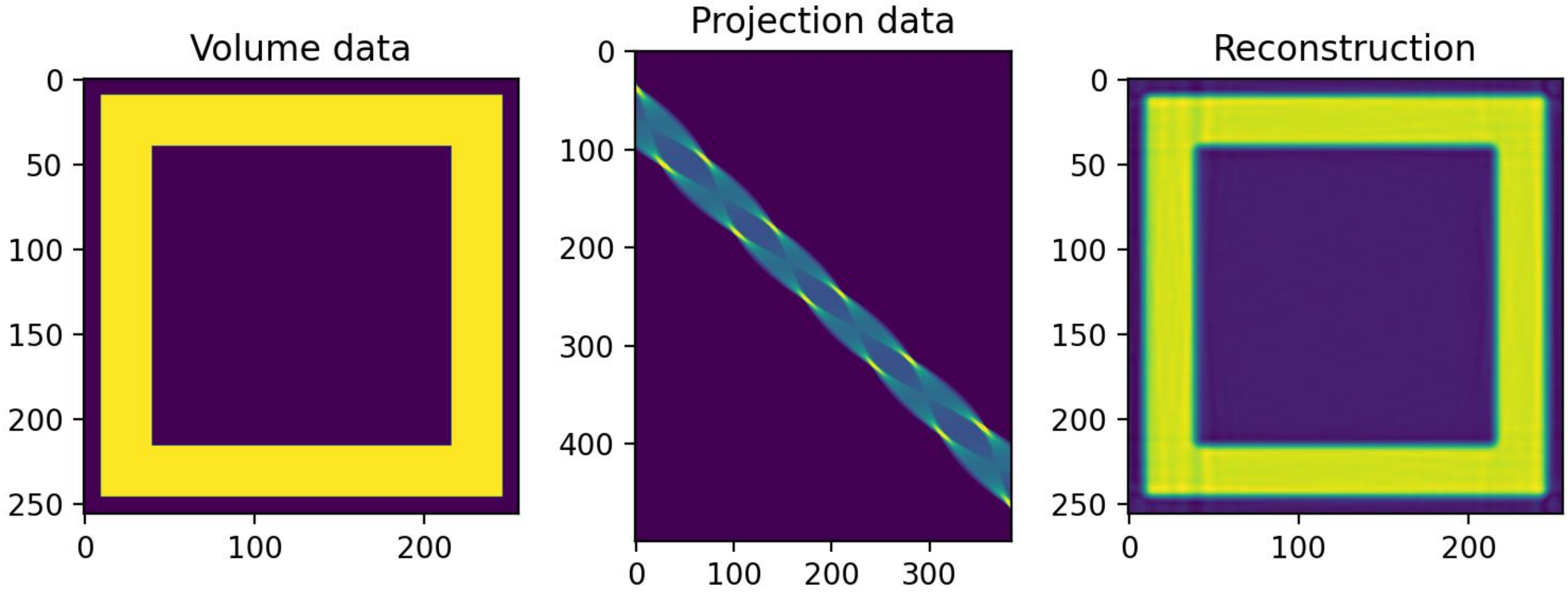
- Two possibilities:
  - .mp4 video
  - animated .svg vector graphics
- Very simple
  - Just call ts.qt.animate or ts.svg functions with any number of geometries as arguments
- Displaying
  - save to file
  - show in Jupyter notebook

Working with PyTorch

- Torch Tensors behave very similarly to Numpy arrays
- Computations on the gpu also possible
  - Call tensor_name.cuda() to copy a tensor to the gpu
  - Call tensor_name.cpu() to copy a tensor to RAM
- Tomosipo operators can act on Torch Tensors just like Numpy arrays
  - You get back the same type as you put in (numpy, gpu tensor, cpu tensor)
- ts_algorithms requires tensor input

```python
import numpy as np
import torch
import tomosipo as ts
import tomosipo.torch_support
from tomosipo.qt import animate
import ts_algorithms as tsa
from matplotlib import pyplot as plt

nump_proj = 500

# Setup 2D volume and fan beam projection geometry
vg = ts.volume(shape=(1, 256, 256), size=(0.1, 0.5, 0.5))
pg = ts.cone(shape=(1, 384), size=(0.1, 5), src_orig_dist=2.25, src_det_dist=3)

# Setup the a rotation and translation transform
tra = ts.translate(axis=np.array((0, 0, 1)), alpha=np.linspace(-2.5, 2.5, nump_proj))
rot = ts.rotate(pos=0, axis=np.array((1, 0, 0)), angles=np.linspace(0, 3*np.pi, nump_proj))

# Apply transformations to the volume geometry
vg = tra * rot * vg.to_vec()
pg = pg.to_vec()

# Create an operator from the transformed geometries
A = ts.operator(vg, pg)
```

```python
# Make an animation of the geometries and save it
s = ts.scale(1.8)
animation = animate(s * vg, s * pg)
animation.save("geometry_video.mp4")

# Create hollow cube phantom and copy it to the GPU
x = torch.zeros(A.domain_shape)
x[:, 10:-10, 10:-10] = 1.0
x[:, 40:-40, 40:-40] = 0.0
x = x.cuda()

# Project the volume data to obtain the projection data
y = A(x)

# Reconstruct using SIRT and copy everything back to RAM
recon = tsa.sirt(A, y, num_iterations=100)
recon = recon.cpu(); x = x.cpu(); y = y.cpu()

plt.figure(figsize=(9, 3))
plt.subplot(131); plt.imshow(x[0, ...]); plt.title("Volume data")
plt.subplot(132); plt.imshow(y[0, ...]); plt.title("Projection data")
plt.subplot(133); plt.imshow(recon[0, ...]); plt.title("Reconstruction")
plt.show()
```


Volume data
Projection data
Reconstruction
0
50
100
150
200
250
0
100
200
0
100
200
300
400
0
100
200
300

FleX-ray example

- Datasets take up a lot of memory
  - FDK algorithm
- Load the geometry from scan_settings.txt file

Working with large FleX-ray data

- High resolution FleX-ray dataset
  - Full resolution 1912 x 1520
  - 3600 projections
  - 21 GB on harddisk stored as 16 bit ints
  - 42 GB in memory stored as 32 bit floats
- Reconstruction volume of 1912x1912x1520 is also 22GB
- FDK algorithm optimized to work on CWI workstations
  - 64GB memory available
- Only works when no other copies are made

Working with large FleX-ray data

- High resolution FleX-ray dataset
  - Full resolution 1912 x 1520
  - 3600 projections
  - 21 GB on harddisk stored as 16 bit ints
  - 42 GB in memory stored as 32 bit floats
- Reconstruction volume of 1912x1912x1520 is also 22GB
- FDK algorithm optimized to work on CWI workstations
  - 64GB memory available
- Only works when no other copies are made
- Memory use not the runtime bottleneck
  - +-13.5 minutes on workstation, +-26 minutes on voxel1 (with 128GB memory)
  - 72.5s with half the resolution and half the projections

Avoiding copies

- Load images directly into array
- Preprocess in place
- Do FDK weighting and filtering in place
  - Use overwrite_y=True in the FDK call
- Converting between torch and numpy does not copy the data

```python
# Preprocesses the projection data without making copies
def preprocess_in_place(y, dark, flat):
    dark = dark[:, None, :]
    flat = flat[:, None, :]
    y -= dark
    y /= (flat - dark)
    torch.log_(y)
    y *= -1
```

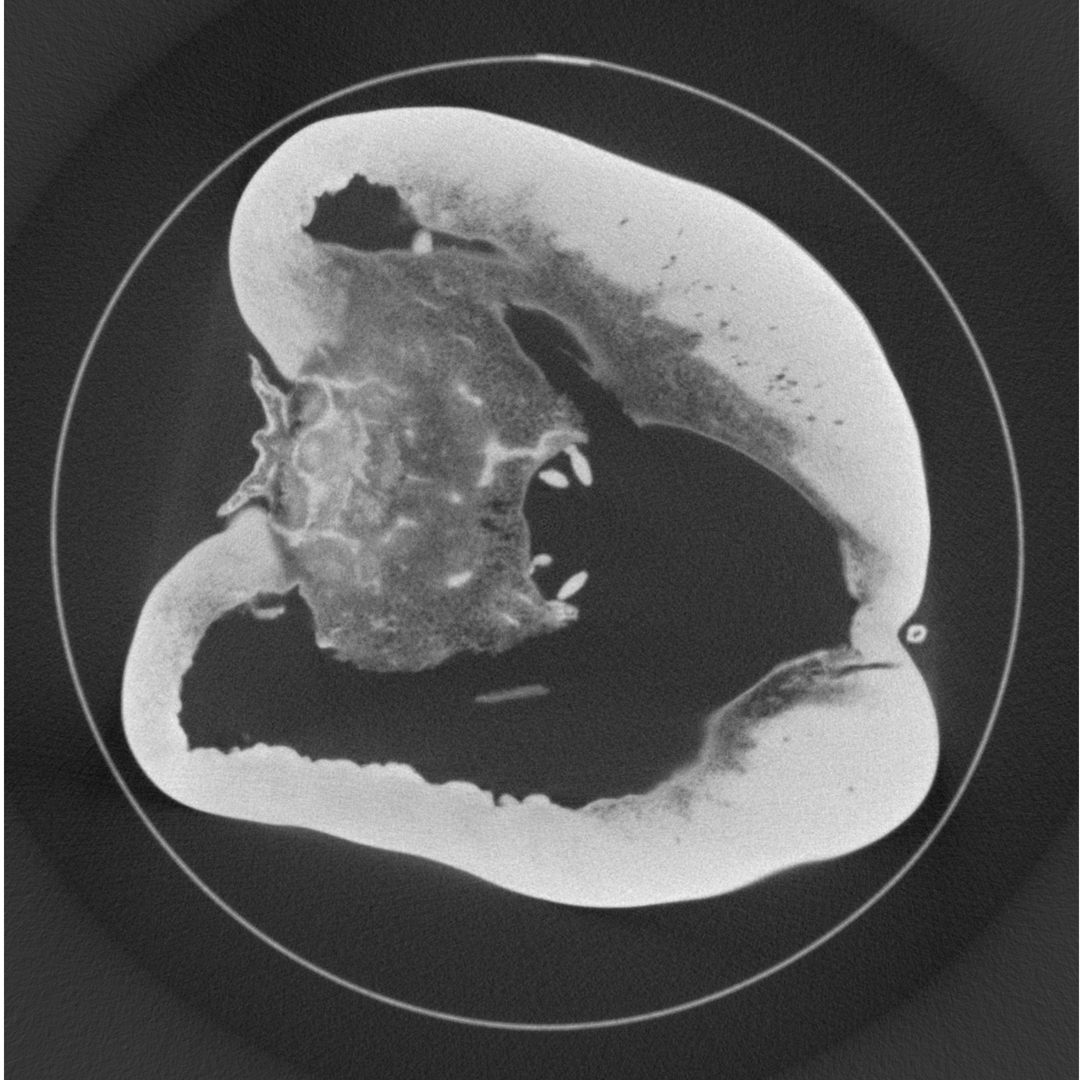
Parse the scan settings

- Three settings needed for FDK reconstruction
  - Source detector distance
  - Source object distance
  - Binned pixel size
- Available in `scan settings.txt` file
  - `Name : value` format

```
SCAN DURATION : 7 minutes
COR : 478.000497
VC : 383.056383
HC : 478.000394
SDD : 1098.000000
SOD : 1001.305645
Voxel size : 136.425614
Magnification : 1.096568
```

```python
# The function parse_scan_settings reads the scan settings.txt into a dictionary
scan_settings = parse_scan_settings(data_path / "scan settings.txt")
# Read the required settings from the file
src_det_dist = float(scan_settings["SDD"])
src_obj_dist = float(scan_settings["SOD"])
pixel_size = float(scan_settings["Binned pixel size"])
# Derive the other settings from the input data
detector_hor_res = y.shape[2]
detector_ver_res = y.shape[0]
num_angles = y.shape[1]

# Make volume and projection geometries with the parameters
vg = ts.volume(
    shape=(detector_ver_res, detector_hor_res, detector_hor_res),
    size=np.array((detector_ver_res, detector_hor_res, detector_hor_res))*pixel_size
)
pg = ts.cone(
    angles=num_angles,
    shape=(detector_ver_res, detector_hor_res),
    size=np.array((detector_ver_res, detector_hor_res))*pixel_size,
    src_det_dist = src_det_dist,
    src_orig_dist = src_obj_dist
)
# Combine the geometries into an operator
op = ts.operator(vg, pg)
```

```python
# Make an FDK reconstruction
# If you are using large projection data you may want to use overwrite_y=True
reconstruction = tsa.fdk(A=op, y=y, overwrite_y=True)
print("Finished reconstruction")
# Variable y was overwritten with a filtered version of y because overwrite_y=True
# You probably don't want to use this so delete y to free up memory
del y

save_stack(save_path, reconstruction.numpy(), exist_ok=True)
```



Calibrating the projection geometry

- Use a vector cone beam projection geometry to describe the exact setup

- FDK implementation is flexible
  - Allows off axis rotations
  - Allows non centered detectors
  - FDK approximations get worse further from the cone beam center

Including operators in PyTorch neural networks

- Tomosipo operators can be converted to a differentiable PyTorch layer
    - tomosipo.torch_support.to_autograd(operator)
- Usable in a neural network like any other layer

Recap

- Tomosipo is useful in many situations
  - FleX-ray reconstructions
  - Unconventional geometries
  - CT operators in neural networks
- Pytorch integration allows fast reconstruction algorithms in Python
  - Both on CPU and GPU
- ts_algorithms already implements some reconstruction algorithms
  - FBP, FDK, SIRT, Chambolle-Pock based 2D TV minimization

Useful links

- Tomosipo examples
  - https://github.com/D1rk123/tomosipo_examples
- Tomosipo documentation
  - https://aahendriksen.gitlab.io/tomosipo/topics/index.html
- Tomosipo online tutorial by Allard
  - https://blog.allardhendriksen.nl/cwi-ci-group/advent-of-tomosipo-introduction/
- Git repositories
  - https://github.com/ahendriksen/tomosipo
  - https://github.com/ahendriksen/ts_algorithms

Centrum Wiskunde & Informatica

Tomosipo tutorial

Dirk Schut

21 June 2021