

ESRF tomography software

This document describes the ESRF tomography processing software under development: Nabu <https://gitlab.esrf.fr/tomotools/nabu/>

Features and scope

The features and scope of the `Nabu` package are translated in its structure (modules). For example there is a `nabu.preproc` module, `nabu.reconstruction`, `nabu.io`, etc.

Pre-processing:

- Intensity normalization (flat-field)
- Phase retrieval
- CCD corrections (hot spots)
- Projections-based rings artefacts removal ("double flat-field")
- Sinogram-based rings artefacts removal (Fourier-Wavelets filter)

Reconstruction:

- Center of rotation finding methods
- FBP
- in the future: iterative (can be designed flexibly with Projection and Backprojection available as operator)

Input/output:

- HDF5 with Nexus convention
- npy/npz (unofficial, for development and debug)
- in the future: tiff and jpeg2000

Miscellaneous image/volume processing features:

- unsharp mask (simple deconvolution of gaussian PSF)
- binning
- median filter

Post-processing (in not-so-near future !):

- post-reconstruction artefacts removal
- volume stitching

Computations distributions

- Computations distributions on the local machine or in a task scheduler (SLURM, ...) using dask.distributed.
- In Nabu, we try to decouple the "computation distribution logic" from actual implementation of processing steps.

Internal pipelines

- Default "full-field tomography pipeline"
- in the future: XRD-CT pipeline
- A general-purpose customizable pipeline where components (nabu.app.component) are plugged. It is experimental at this point.

processing-related features are implemented with three backends:

- python/numpy/scipy: reference implementations
- cuda
- opengl

The Cuda and Opengl implementations use pycuda and pyopengl respectively.

State of the art and requirements

There are several well-known tomography software used in various institutes, for example tomopy (APS, Elettra), Savu (DLS) and UFO (KIT). Each of them has its own "approach" when it comes to building an end-to-end tomography processing workflow:

- Tomopy provides building blocks of essential tomography features.
- UFO/Tofu provides an Opengl-based pipeline, trying to achieve bare-metal performances with a filter-based approach.
- Savu offers a very customizable toolkit by integrating many existing software through a plugins system.

Each of them has its own advantages and drawbacks, which out of the scope of the current document. In the end, it was decided to start a new project to fulfill the following requirements:

- A software being primarily a library of tomography processing, with "applications" built on top of it, usable by both non-experts and power-users
- High performance (parallelization with Cuda/OpenCL, computations distribution, memory re-use)
- Extensive documentation
- Support of different tomography modalities: absorption, phase contrast, diffraction and fluorescence
- Official support by the ESRF Data Analysis Unit, with a "bus factor" greater than one
- Compatible with ESRF legacy software, progressively replacing it

While the first three points would suggest to use an off-the-shelf software, the other points leave little choice to start a new project. In particular, the last point is critical. There are many tomography beamlines at ESRF, each using various in-house dedicated tools that is only known locally. The current project also aims at sharing tools and avoid duplicating efforts. Besides, starting a new project does not prevent from using other projects components and contributing to these projects.

Technical choices

High performances processing pipeline

Implementing an individual processing feature (ex. "FBP") is generally simple. In fact, there are chances that an implementation is already available in another software. What takes time is to

1. verify that it works well (validation on ground-truth, unit tests)
2. provide a high performance implementation
3. document this feature for the final user
4. integrate this feature in a wider processing pipeline

In the best case, points (1) to (3) are fulfilled when using an off-the-shelf implementation of another software.

The tricky point is to keep high performance when using various components together. Usually, tomography software assume that "everything fits in memory" or consider each component individually. We try to design Nabu so that memory transactions are minimized.

Usage of native code

Nabu uses Cuda and OpenCL to offer a high speed implementation of various processing features. Cuda and OpenCL are used through pycuda/pyopencl respectively.

Although OpenCL seems more appealing on many points, its uncertain future lead to choose Cuda as the default GPU backend. Having the two backends comes at the expense of possible code duplication.

An important point is the presence of native code in Nabu. By experience, a Python module having native code extensions (C/C++ wrapped with Cython, pybind11, etc) is cumbersome to distribute and deploy. Therefore, we want to avoid ahead-of-time compilation (ex. Cython). Pyopencl and pycuda enable to achieve this goal with Just-In-Time compilation.

Computations distribution

In Nabu, we try to decouple the "computation distribution logic" from actual implementation of processing steps. In particular, solutions that are "invasive" in the code like MPI are avoided. We use the Remote Procedural Call (RPC) approach to distribute the computations, with the goal to "move the computing resources to the data, not the other way around". This approach offers many advantages, but has also limitations to keep in mind (ex. sharing memory might be trickier).

The modules `dask.distributed` and `dask_jobqueue` are used, as we believe they are well-fit for the tomography processing use case.

Dependencies

Summary

Nabu uses the following python modules:

- Required: numpy, silx, tomoscan, pytest
- Optionally, for using the Cuda backend: `pycuda` and `scikit-cuda`
- Optionally, for using the OpenCL backend: `pyopencl`
- Optionally, for distributing the computations: `dask.distributed`
- Optionally, for distributing the computations on a task scheduler: `dask_jobqueue`

Although some modules listed are optional (meaning Nabu can work without them), the user experience will be degraded without them ; for example processing will be very slow with the default python/numpy-based implementations.

Of course python modules like `pyopencl` have in turn system dependencies (eg. a working OpenCL installation).

Full dependency trees

As of April 2020, the dependency trees of main python modules are the following.

Nabu

- dask-jobqueue
 - dask
 - distributed
 - click
 - cloudpickle
 - dask

- msgpack
 - psutil
 - pyyaml
 - setuptools
 - sortedcontainers
 - tblib
 - toolz
 - tornado
 - zict
 - heapdict
- distributed
 - click
 - cloudpickle
 - dask
 - msgpack
 - psutil
 - pyyaml
 - setuptools
 - sortedcontainers
 - tblib
 - toolz
 - tornado
 - zict
 - heapdict
- numpy
- psutil
- pytest
 - attrs
 - importlib-metadata
 - zipp
 - more-itertools
 - packaging
 - pyparsing
 - six
 - pluggy
 - importlib-metadata
 - zipp
 - py
 - wcwidth

- silx
 - fabio
 - numpy
 - setuptools
 - h5py
 - numpy
 - six
 - numpy
 - setuptools
 - six
- tomoscan
 - setuptools
 - lxml
- pycuda==2019.1.2
 - appdirs
 - decorator
 - mako
 - MarkupSafe
 - pytools
 - appdirs
 - decorator
 - numpy
 - six
- pyopencl==2019.1.2
 - appdirs
 - decorator
 - numpy
 - pytools
 - appdirs
 - decorator
 - numpy
 - six
 - six
- scikit-cuda==0.5.3
 - mako
 - MarkupSafe
 - numpy
 - pycuda
 - appdirs
 - decorator
 - mako
 - MarkupSafe

- pytools
 - appdirs
 - decorator
 - numpy
 - six