# STP-Core Documentation

***Release 0.0.1***

**Sincrotrone Trieste S.C.p.A.**

**Sep 06, 2017**

# Contents

STP Core provides .....

Here is how to add a link to your documentation Docs and here is how to add a reference *[A1]*

CHAPTER 1

# Features

- Example of how to write documentation

CHAPTER 2

# Contribute

- Documentation: https://github.com/decarlof/pore3d/tree/master/doc
- Issue Tracker: https://github.com/decarlof/pore3d/docs/issues
- Source Code: https://github.com/decarlof/pore3d/

CHAPTER 3

# Content

## About

This section describes what the STP Core project is about ...

## Install

This section covers the basics of how to download and install STP Core

**Contents:**



### Installing from source

Clone the STP Core from GitHub repository:

```
git clone https://github.com/ElettraSciComp/STP-Core.git STP-Core
```

then:

```
cd STP-Core
python setup.py install
```

## API reference

### project Modules:

## io.tdf

### Functions:

| | |
|---|---|
| *parse_metadata*(f, xml_command) | Fill the specified HDF5 file with metadata according to the DataExchange initiative. |
| *read_tomo*(dataset, index) | Extract the tomographic projection at the specified relative index from the HDF5 dataset. |
| *parse_metadata*(f, xml_command) | Fill the specified HDF5 file with metadata according to the DataExchange initiative. |
| *read_sino*(dataset, index) | Extract the sinogram at the specified relative index from the HDF5 dataset. |
| *write_tomo*(dataset, index, im) | Modify the tomographic projection at the specified relative index from the HDF5 dataset with the image passed as input. |
| *write_sino*(dataset, index, im) | Modify the sinogram at the specified relative index from the HDF5 dataset with the image passed as input. |
| *get_nr_projs*(dataset) | Get the number of projections of the input dataset. |
| *get_nr_sinos*(dataset) | Get the number of sinograms (or slices) of the input dataset. |
| *get_det_size*(dataset) | Get the width of the detector (nr of pixels) of the input dataset. |
| *get_dset_shape*(det_size, fov_height, nr_proj) | Get the shape of the dataset by arranging the input parameters. |
| *get_dset_chunks*(det_size) | Get a good chunk combination. |

stp_core.io.tdf.**get_det_size**(*dataset*)
Get the width of the detector (nr of pixels) of the input dataset.

**Parameters** **dataset** (*HDF5 dataset*) – HDF5 dataset as returned by the h5py API.

stp_core.io.tdf.**get_dset_chunks**(*det_size*)
Get a good chunk combination. This function needs improvement...

**Parameters** **det_size** (*int*) – Width of the detector.

stp_core.io.tdf.**get_dset_shape**(*det_size*, *fov_height*, *nr_proj*)
Get the shape of the dataset by arranging the input parameters.

**Parameters**

- **det_size** (*int*) – Width of the detector.
- **fov_height** (*int*) – Height of the FOV, i.e. the number of sinograms (or slices) of the dataset.
- **nr_proj** (*int*) – Number of collected projections.

stp_core.io.tdf.**get_nr_projs**(*dataset*)
Get the number of projections of the input dataset.

**Parameters** **dataset** (*HDF5 dataset*) – HDF5 dataset as returned by the h5py API.

stp_core.io.tdf.**get_nr_sinos**(*dataset*)
Get the number of sinograms (or slices) of the input dataset.

**Parameters** **dataset** (*HDF5 dataset*) – HDF5 dataset as returned by the h5py API.

stp_core.io.tdf.**parse_metadata**(*f*, *xml_command*)
Fill the specified HDF5 file with metadata according to the DataExchange initiative. The metadata in input are described in a XML format.

**Parameters**

- **f** (*HDF5 file*) – HDF5 file open with h5py API
- **xml_command** (*string*) – Immaginary part of the complex X-ray refraction index.

stp_core.io.tdf.**read_sino**(*dataset*, *index*)
Extract the sinogram at the specified relative index from the HDF5 dataset.

**Parameters**

- **dataset** (*HDF5 dataset*) – HDF5 dataset as returned by the h5py API.
- **index** (*int*) – Relative position of the sinogram within the dataset.

stp_core.io.tdf.**read_tomo**(*dataset*, *index*)
Extract the tomographic projection at the specified relative index from the HDF5 dataset.

**Parameters**

- **dataset** (*HDF5 dataset*) – HDF5 dataset as returned by the h5py API.
- **index** (*int*) – Relative position of the tomographic projection within the dataset.

stp_core.io.tdf.**write_sino**(*dataset*, *index*, *im*)
Modify the sinogram at the specified relative index from the HDF5 dataset with the image passed as input.

**Parameters**

- **dataset** (*HDF5 dataset*) – HDF5 dataset as returned by the h5py API.
- **index** (*int*) – Relative position of the sinogram within the dataset.
- **im** (*array_like*) – Image data as numpy array.

stp_core.io.tdf.**write_tomo**(*dataset*, *index*, *im*)
Modify the tomographic projection at the specified relative index from the HDF5 dataset with the image passed as input.

**Parameters**

- **dataset** (*HDF5 dataset*) – HDF5 dataset as returned by the h5py API.
- **index** (*int*) – Relative position of the tomographic projection within the dataset.
- **im** (*array_like*) – Image data as numpy array.

## phaseretrieval.tiehom

### Functions:

| | |
|---|---|
| *tiehom_plan*(im, beta, delta, energy, ...) | Pre-compute data to save time in further execution of phase_retrieval with TIE-HOM (Paganin's) algorithm. |
| *tiehom*(im, plan[, nr_threads]) | Process a tomographic projection image with the TIE-HOM (Paganin's) phase retrieval algorithm. |

stp_core.phaseretrieval.tiehom.**tiehom**(*im*, *plan*, *nr_threads=2*)
Process a tomographic projection image with the TIE-HOM (Paganin's) phase retrieval algorithm.

**Parameters**

- **im** (*array_like*) – Flat corrected image data as numpy array.
- **plan** (*structure*) – Structure with pre-computed data (see tiehom_plan function).
- **nr_threads** (*int*) – Number of threads to be used in the computation of FFT by PyFFTW (default = 2).

stp_core.phaseretrieval.tiehom.**tiehom_plan**(*im*, *beta*, *delta*, *energy*, *distance*, *pixsize*, *padding*)

Pre-compute data to save time in further execution of phase_retrieval with TIE-HOM (Paganin's) algorithm.

**Parameters**

- **im** (*array_like*) – Image data as numpy array. Only image size (shape) is actually used.
- **beta** (*double*) – Immaginary part of the complex X-ray refraction index.
- **delta** (*double*) – Decrement from unity of the complex X-ray refraction index.
- **energy [KeV]** (*double*) – Energy in KeV of the incident X-ray beam.
- **distance [mm]** (*double*) – Sample-to-detector distance in mm.
- **pixsize [mm]** (*double*) – Size in mm of the detector element.
- **padding** (*bool*) – Apply image padding to better process the boundary of the image

## phaseretrieval.phrt

### Functions:

| | |
|---|---|
| *phrt_plan*(im, energy, distance, pixsize, ...) | Pre-compute data to save time in further execution of phase_retrieval. |
| *phrt*(im, plan[, method, nr_threads]) | Process a tomographic projection image with the selected phase retrieval algorithm. |

stp_core.phaseretrieval.phrt.**phrt**(*im*, *plan*, *method=4*, *nr_threads=2*)

Process a tomographic projection image with the selected phase retrieval algorithm.

**Parameters**

- **im** (*array_like*) – Flat corrected image data as numpy array.
- **plan** (*structure*) – Structure with pre-computed data (see prepare_plan function)
- **method** (*int*) – Phase retrieval filter {1 = TIE (default), 2 = CTF, 3 = CTF first-half sine, 4 = Quasiparticle, 5 = Quasiparticle first half sine}.
- **nr_threads** (*int*) – Number of threads to be used in the computation of FFT by PyFFTW
- **Credits**
- **——-**
- **Julian Moosmann, KIT (Germany) is acknowledged for this code**

stp_core.phaseretrieval.phrt.**phrt_plan**(*im*, *energy*, *distance*, *pixsize*, *regpar*, *thresh*, *method*, *padding*)

Pre-compute data to save time in further execution of phase_retrieval.

**Parameters**

- **im** (*array_like*) – Image data as numpy array. Only image size (shape) is actually used.
- **energy [KeV]** (*double*) – Energy in KeV of the incident X-ray beam.
- **distance [mm]** (*double*) – Sample-to-detector distance in mm.
- **pixsize [mm]** (*double*) – Size in mm of the detector element.
- **regpar** (*double*) – Regularization parameter: RegPar is - log10 of the constant to be added to the denominator to regularize the singularity at zero frequency, i.e. 1/sin(x) -> 1/(sin(x)+10^-RegPar). Typical values in the range [2.0, 3.0]. (Suggestion for default: 2.5).
- **thresh** (*double*) – Parameter for Quasiparticle phase retrieval which defines the width of the rings to be cropped around the zero crossing of the CTF denominator in Fourier space. Typical values in the range [0.01, 0.1]. (Suggestion for default: 0.1).
- **method** (*int*) – Phase retrieval algorithm {1 = TIE (default), 2 = CTF, 3 = CTF first-half sine, 4 = Quasiparticle, 5 = Quasiparticle first half sine}.
- **padding** (*bool*) – Apply image padding to better process the boundary of the image.

#### References

#### Notes

Credits to Julian Moosmann, KIT (Germany) is acknowledged for this code

## postprocess

### Functions:

| | |
|---|---|
| *postprocess*(im, convert_opt, crop_opt) | Post-process a reconstructed image. |

stp_core.postprocess.postprocess.**postprocess**(*im*, *convert_opt*, *crop_opt*)
Post-process a reconstructed image.

**Parameters** **im** (*array_like*) – Image data as numpy array.

**convert_opt** [string] String containing degradation method (8-bit or 16-bit) and min/max rescaling value (e.g. "linear8:-0.01;0.01"). In current version only "linear" for 16-bit and "linear8" are implemented.

**crop_opt** [double] String containing the parameters to crop an image separated by : with order top, bottom, left, right. (e.g. "100:100:100:100")

## preprocess.extfov_correction

### Functions:

| | |
|---|---|
| *extfov_correction*(im, ext_fov, ...) | Apply sinogram correction for extended FOV acquisition mode |

stp_core.preprocess.extfov_correction.**extfov_correction**(*im*, *ext_fov*, *ext_fov_rot_right*, *ext_fov_overlap*)

Apply sinogram correction for extended FOV acquisition mode

**Parameters**

- **im** (*array_like*) – Image data (sinogram) as numpy array.
- **ext** (*bool*) – True if the extended FOV mode has been performed.
- **ext_fov_rot_right** (*bool*) –

  **True if the extended FOV mode has been performed with rotation center** shifted to the right, left otherwise.

  **ext_fov_overlap** [int] Number of overlapping pixels.

## preprocess.extract_flatdark

### Functions:

| | |
|---|---|
| *extract_flatdark*(f_in, flat_end, logfilename) | Extract the flat and dark reference images to be used during the pre-processing step. |

stp_core.preprocess.extract_flatdark.**extract_flatdark**(*f_in*, *flat_end*, *logfilename*)

Extract the flat and dark reference images to be used during the pre-processing step.

**Parameters**

- **f_in** (*HDF5 data structure*) – The data structure containing the flat and dark acquired images.
- **flat_end** (*bool*) – Consider the flat/dark images acquired after the projections (if any).
- **logilename** (*string*) – Absolute file of a log text file where infos are appended.

## preprocess.flat_fielding

### Functions:

| | |
|---|---|
| *flat_fielding*(im, i, plan, flat_end, ...) | Process a sinogram with conventional flat fielding plus reference normalization. |

stp_core.preprocess.flat_fielding.**flat_fielding**(*im*, *i*, *plan*, *flat_end*, *half_half*, *half_half_line*, *norm_sx*, *norm_dx*)

Process a sinogram with conventional flat fielding plus reference normalization.

**Parameters**

- **im** (*array_like*) – Image data as numpy array
- **i** (*int*) – Index of the sinogram with reference to the height of a projection
- **plan** (*structure*) – Structure created by the extract_flatdark function (see extract_flatdark.py). This structure contains the flat/dark images acquired before the acqui-

sition of the projections and the flat/dark images acquired after the acquisition of the projections as well as a few flags.

- **flat_end** (*bool*) – True if the process considers the flat/dark images (if any) acquired after the acquisition of the projections.
- **half_half** (*bool*) – True if the process has to be separated by processing the first part of the sinogram with the flat/dark images acquired before the acquisition of the projections and the second part with the flat/dark images acquired after the acquisition of the projections.
- **half_half_line** (*int*) – Usually this value is equal to the height of the projection FOV / 2 but the two parts of the sinogram to process can have a different size.
- **norm_sx** (*int*) – Width in pixels of the left window to be consider for the normalization of the sinogram. This value has to be zero in the case of ROI-CT.
- **norm_dx** (*int*) – Width in pixels of the right window to be consider for the normalization of the sinogram. This value has to be zero in the case of ROI-CT.
- **Example (using h5py, tdf.py, tifffile.py)**
- **————————————**
- **>>> sino_idx = 512**
- **>>> f = getHDF5('dataset.h5', 'r')**
- **>>> im = tdf.read_sino(f['exchange/data'], sino_idx)**
- **>>> plan = extract_flatdark(f_in, True, False, False, 'tomo', 'dark', 'flat', 'logfile.txt')**
- **>>> im = flat_fielding(im, sino_idx, plan, True, True, 900, 0, 0)**
- **>>> imsave('sino_corr.tif', im)**

## preprocess.ring_correction

### Functions:

| | |
|---|---|
| *ring_correction*(im, ringrem, flat_end, ...) | Apply ring artifacts compensation by de-striping the input sinogram. |

stp_core.preprocess.ring_correction.**ring_correction**(*im*, *ringrem*, *flat_end*, *skip_flat_after*, *half_half*, *half_half_line*, *ext_fov*)

Apply ring artifacts compensation by de-striping the input sinogram.

**Parameters** **im** (*array_like*) – Image data (sinogram) as numpy array.

**ringrem** [string] String containing ring removal method and parameters

**half_half** [bool] True to separately process the sinogram in two parts

**half_half_line** [int] Line number considered to identify the two parts to be processed separately. (This parameter is ignored if half_half is False)

skip_flat_after e ext_fov SERVE???

## reconstruct.rec_astra

### Functions:

| | |
|---|---|
| *recon_astra_fbp*(im, angles, method, filter_type) | Reconstruct the input sinogram by using the FBP implemented in ASTRA toolbox. |
| *recon_astra_iterative*(im, angles, method, ...) | Reconstruct the input sinogram by using one of the iterative algorithms implemented in ASTRA toolbox. |

stp_core.reconstruct.rec_astra.**recon_astra_fbp**(*im*, *angles*, *method*, *filter_type*)

Reconstruct the input sinogram by using the FBP implemented in ASTRA toolbox.

**Parameters** **im** (*array_like*) – Image data (sinogram) as numpy array.

**angles** [double] Value in radians representing the number of angles of the sinogram.

**method** [string] A string with either "FBP" or "FBP_CUDA".

**filter_type** [string] The available options are "ram-lak", "shepp-logan", "cosine", "hamming", "hann", "tukey", "lanczos", "triangular", "gaussian", "barlett-hann", "blackman", "nuttall", "blackman-harris", "blackman-nuttall", "flat-top", "kaiser", "parzen".

stp_core.reconstruct.rec_astra.**recon_astra_iterative**(*im*, *angles*, *method*, *iterations*, *zerone_mode*)

Reconstruct the input sinogram by using one of the iterative algorithms implemented in ASTRA toolbox.

**Parameters** **im** (*array_like*) – Image data (sinogram) as numpy array.

**angles** [double] Value in radians representing the number of angles of the sinogram.

**method** [string] A string with e.g "SIRT" or "SIRT_CUDA" (see ASTRA documentation)

**iterations** [int] Number of iterations for the algebraic technique

**zerone_mode** [bool] True if the input sinogram has been rescaled to the [0,1] range (therefore positivity constraints are applied)

## reconstruct.rec_fista_tv

### Functions:

| | |
|---|---|
| *recon_fista_tv*(im, angles, lam, fista_iter, iter) | Reconstruct the input sinogram by using the FISTA-TV algorithm |

stp_core.reconstruct.rec_fista_tv.**recon_fista_tv**(*im*, *angles*, *lam*, *fista_iter*, *iter*)

Reconstruct the input sinogram by using the FISTA-TV algorithm

**Parameters** **im** (*array_like*) – Image data (sinogram) as numpy array.

**angles** [double] Value in radians representing the number of angles of the input sinogram.

**lam** [double] Regularization parameter of the FISTA algorithm.

**fista_iter** [int] Number of iterations of the FISTA algorihtm.

**iter** [int] Number of iterations of the TV minimization.

## reconstruct.rec_gridrec

### Functions:

| | |
|---|---|
| *recon_gridrec*(im1, im2, angles, oversampling) | Reconstruct two sinograms (of the same CT scan) with direct Fourier algorithm. |

stp_core.reconstruct.rec_gridrec.**recon_gridrec**(*im1*, *im2*, *angles*, *oversampling*)
Reconstruct two sinograms (of the same CT scan) with direct Fourier algorithm.

**Parameters**

- **im1** (*array_like*) – Sinogram image data as numpy array.
- **im2** (*array_like*) – Sinogram image data as numpy array.
- **angles** (*double*) – Value in radians representing the number of angles of the input sinogram.
- **oversampling** (*double*) – Input sinogram is rescaled to increase the sampling of the Fourier space and avoid artifacts. Suggested value in the range [1.2,1.6].

## reconstruct.rec_mr_fbp

### Functions:

| | |
|---|---|
| *recon_mr_fbp*(im, angles) | Reconstruct a sinogram with the Minimum Residual FBP algorithm (Pelt, 2013). |

stp_core.reconstruct.rec_mr_fbp.**recon_mr_fbp**(*im*, *angles*)
Reconstruct a sinogram with the Minimum Residual FBP algorithm (Pelt, 2013).

**Parameters**

- **im** (*array_like*) – Sinogram image data as numpy array.
- **angles** (*double*) – Value in radians representing the number of angles of the input sinogram.

## utils.caching

### Functions:

| | |
|---|---|
| *cache2plan*(infile, cachepath) | Read from cache the flat/dark images of the input TDF file. |
| *plan2cache*(corr_plan, infile, cachepath) | Write to cache the flat/dark images of the input TDF file. |

stp_core.utils.caching.**cache2plan**(*infile*, *cachepath*)
Read from cache the flat/dark images of the input TDF file.

**Parameters** **infile** (*string*) – Absolute path of the input TDF dataset.

**returns** *A structure with flat/dark images and related flags.*

stp_core.utils.caching.**plan2cache**(*corr_plan*, *infile*, *cachepath*)
Write to cache the flat/dark images of the input TDF file.

**Parameters** **infile** (*string*) – Absolute path of the input TDF dataset.

**corr_plan** [structure] The plan with flat/dark images and flags.

**returns** *No return value.*

## utils.findcenter

### Functions:

| | |
|---|---|
| *usecorrelation*(im1, im2) | Assess the offset (to be used for e.g. |

stp_core.utils.findcenter.**usecorrelation**(*im1*, *im2*)

**Assess the offset (to be used for e.g. the assessment of the center of rotation or the** ovarlap) by computation the peak of the correlation between the two input images.

**Parameters** **im1** (*array_like*) – Image data as numpy array.

**im2** [array_like] Image data as numpy array.

**returns** *An integer value of the location of the maximum peak correlation.*

## utils.padding

### Functions:

| | |
|---|---|
| *upperPowerOfTwo*(v) | Return the upper power of two of input value |
| *replicatePadImage*(im, marg0, marg1) | Pad the input image by replicating first and last column as well as first and last row the specified number of times. |
| *zeroPadImage*(im, marg0, marg1) | Pad the input image by adding zeros. |
| *padImage*(im, n_pad0, n_pad1) | Replicate pad the input image to the specified new dimensions. |
| *padSmoothWidth*(im, n_pad) | Pad the input image to the specified new width by replicate padding with Hanning smoothing to zero. |

stp_core.utils.padding.**padImage**(*im*, *n_pad0*, *n_pad1*)

Replicate pad the input image to the specified new dimensions.

**Parameters**

- **im** (*array_like*) – Image data as numpy array
- **n_pad0** (*int*) – The new height of the image
- **n_pad1** (*int*) – The new width of the image
- **Return value**
- **———-**
- **A padded image**

stp_core.utils.padding.**padSmoothWidth**(*im*, *n_pad*)

Pad the input image to the specified new width by replicate padding with Hanning smoothing to zero.

**Parameters**

- **im** (*array_like*) – Image data as numpy array.
- **n_pad** (*int*) – The new width of the image.
- **Return value**
- **———-**
- **A padded image**

stp_core.utils.padding.**replicatePadImage**(*im*, *marg0*, *marg1*)

**Pad the input image by replicating first and last column as well as first and last row** the specified number of times.

**Parameters**

- **im** (*array_like*) – Image data as numpy array.
- **marg0** (*int*) – The number of times first and last row have to be replicated.
- **marg1** (*int*) – The number of times first and last column have to be replicated.
- **Return value**
- **———-**
- **A replicated-padded image.**

stp_core.utils.padding.**upperPowerOfTwo**(*v*)

Return the upper power of two of input value

**Parameters**

- **v** (*int*) – A positive integer value
- **Return value**
- **———-**
- **An integer value**

stp_core.utils.padding.**zeroPadImage**(*im*, *marg0*, *marg1*)

Pad the input image by adding zeros.

**Parameters**

- **im** (*array_like*) – Image data as numpy array.
- **marg0** (*int*) – The number of zero rows to add before first and after last row.
- **marg1** (*int*) – The number of zero rows to add before first and after last column.
- **Return value**
- **———-**
- **A zero-padded image.**

## Examples

Here we describe what the examples are doing. You can cite with *[B1]*.

## exec_his2tdf

This section contains the exec_his2tdf script.

Download file: `exec_his2tdf.py`

```
##########################################################################
# (C) 2016 Elettra - Sincrotrone Trieste S.C.p.A.. All rights reserved.  #
#                                                                        #
#                                                                        #
# This file is part of STP-Core, the Python core of SYRMEP Tomo Project, #
# a software tool for the reconstruction of experimental CT datasets.    #
#                                                                        #
# STP-Core is free software: you can redistribute it and/or modify it    #
# under the terms of the GNU General Public License as published by the  #
# Free Software Foundation, either version 3 of the License, or (at your #
# option) any later version.                                             #
#                                                                        #
# STP-Core is distributed in the hope that it will be useful, but WITHOUT #
# ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or  #
# FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License   #
# for more details.                                                      #
#                                                                        #
# You should have received a copy of the GNU General Public License      #
# along with STP-Core. If not, see <http://www.gnu.org/licenses/>.       #
#                                                                        #
##########################################################################

#
# Author: Francesco Brun
# Last modified: July, 8th 2016
#

import datetime
import os
import os.path
import numpy
import time

from time import strftime
from sys import argv, exit
from glob import glob
from h5py import File as getHDF5
import stp_core.io.tdf as tdf

def _getHISdim ( HISfilename ):

    dim1 = 0
    dim2 = 0
    dimz = 0
    bytecode = numpy.uint16

    # Open file:
    try:
        infile = open(HISfilename, "rb")

        # Get file infos:
        tot_bytes = os.path.getsize(HISfilename)

```

```
        # Read header:
        Image_tag = infile.read(2)
        Comment_len = numpy.fromstring(infile.read(2), numpy.uint16)[0].astype(numpy.
↪int_)
        dim1 = numpy.fromstring(infile.read(2), numpy.uint16)[0].astype(numpy.int_)
        dim2 = numpy.fromstring(infile.read(2), numpy.uint16)[0].astype(numpy.int_)
        dim1_offset = numpy.fromstring(infile.read(2), numpy.uint16)[0].astype(numpy.
↪int_)
        dim2_offset = numpy.fromstring(infile.read(2), numpy.uint16)[0].astype(numpy.
↪int_)
        HeaderType = numpy.fromstring(infile.read(2), numpy.uint16)[0]
        Dump = infile.read(50)
        Comment = infile.read(Comment_len)

        # Set total number of bytes read so far:
        bytes_read = 64 + Comment_len

        # Set image type:
        bpp = len(numpy.array(0, bytecode).tostring())

        # Define chunk size:
        chunksize = dim1 * dim2 * bpp

        # Determine number of expected projections:
        dimz = (tot_bytes - bytes_read) / (chunksize + 64) + 1

    finally:
        # Close file:
        infile.close()

    return (dim1, dim2, dimz, bytecode)



def _processHIS( HISfilename, dset, dset_offset, provenance_dset, provenance_offset,␣
↪time_offset, prefix, crop_top, crop_bottom, crop_left, crop_right, logfilename, int_
↪from=0, int_to=-1):

    # Open file:
    infile = open(HISfilename, "rb")

    # Get file infos:
    tot_bytes = os.path.getsize(HISfilename)

    # Read header:
    Image_tag = infile.read(2)
    Comment_len = numpy.fromstring(infile.read(2), numpy.uint16)[0].astype(numpy.int_)
    dim1 = numpy.fromstring(infile.read(2), numpy.uint16)[0].astype(numpy.int_)
    dim2 = numpy.fromstring(infile.read(2), numpy.uint16)[0].astype(numpy.int_)
    dim1_offset = numpy.fromstring(infile.read(2), numpy.uint16)[0].astype(numpy.int_)
    dim2_offset = numpy.fromstring(infile.read(2), numpy.uint16)[0].astype(numpy.int_)
    HeaderType = numpy.fromstring(infile.read(2), numpy.uint16)[0]
    Dump = infile.read(50)
    Comment = infile.read(Comment_len)

    # Set total number of bytes read so far:
    bytes_read = 64 + Comment_len

```

```
    # Set image type:
    bytecode = numpy.uint16
    bpp = len(numpy.array(0, bytecode).tostring())

    # Define chunk size:
    chunksize = dim1 * dim2 * bpp

    # Determine number of expected projections:
    num_proj = (tot_bytes - bytes_read) / (chunksize + 64) + 1

    # Read first projection:
    t1 = time.time()
    block = infile.read(chunksize)

    # Convert as numpy array:
    data = numpy.fromstring(block, bytecode)
    im = numpy.reshape( data, [dim2, dim1])
    im = im[crop_top:im.shape[0]-crop_bottom,crop_left:im.shape[1]-crop_right]

    print numpy.amax(im[:])
    print dset.attrs['max']

    # Set minimum and maximum:
    if ( numpy.amin(im[:]) < float(dset.attrs['min']) ):
        dset.attrs['min'] = str(numpy.amin(im[:]))
    if ( numpy.amax(im[:]) > float(dset.attrs['max'])):
        dset.attrs['max'] = str(numpy.amax(im[:]))

    print numpy.amax(im[:])
    print dset.attrs['max']

    # Check extrema (int_to == -1 means all files) for the projections:
    if ( (int_to >= num_proj) or (int_to <= 0) ):
        int_to = num_proj - 1
    if ( (int_from >= num_proj) or (int_from < 0) ):
        int_from = 0

    # Process first projection (fill HDF5):
    i = 0
    first_index = int(provenance_dset.attrs['first_index'])

    # Save processed image to HDF5 file:
    #tifffile.imsave('tomo_' + str(i).zfill(4) + '.tif', data)
    if (i >= int_from) and (i <= int_to):
        tdf.write_tomo(dset, i + dset_offset  - int_from,im)

        # Save provenance metadata:
        t = time.time() + time_offset*3600
        provenance_dset["filename", provenance_offset + i  - int_from] = prefix + '_'␣
↪+ str(i + dset_offset + first_index).zfill(4)
        provenance_dset["timestamp", provenance_offset + i  - int_from] = numpy.
↪string_(datetime.datetime.fromtimestamp(t).strftime('%Y-%m-%d %H:%M:%S.%f')[:-3])

        # Print out execution time:
        t2 = time.time()
        log = open(logfilename,"a")
        log.write(os.linesep + "\t%s converted in %0.3f sec." % (provenance_dset[
↪"filename", provenance_offset + i  - int_from], t2 - t1))
```

```
        log.close()

    # Read all the other projections:
    try:
        while block:

            # Skip a few bytes:
            t1 = time.time()
            dump = infile.read(64)

            # Read the meaningful data:
            block = infile.read(chunksize)

            # Convert as numpy array:
            data  = numpy.fromstring(block, bytecode)
            im    = numpy.reshape( data, [dim2, dim1])
            im    = im[crop_top:im.shape[0]-crop_bottom,crop_left:im.shape[1]-crop_
↪right]

            # Set minimum and maximum:
            if ( float(numpy.amin(im[:])) < float(dset.attrs['min']) ):
                dset.attrs['min'] = str(numpy.amin(im[:]))
            if ( float(numpy.amax(im[:])) > float(dset.attrs['max'])):
                dset.attrs['max'] = str(numpy.amax(im[:]))

            # Process first projection (fill HDF5):
            i = i + 1

            # Save processed image to HDF5 file:
            #tifffile.imsave('tomo_' + str(i).zfill(4) + '.tif', data)
            if (i >= int_from) and (i <= int_to):
                tdf.write_tomo( dset, i + dset_offset - int_from,im )

                # Save provenance metadata:
                t = time.time() + time_offset*3600
                provenance_dset["filename", provenance_offset + i  - int_from] =␣
↪prefix + '_' + str(i + dset_offset + first_index  - int_from).zfill(4)
                provenance_dset["timestamp", provenance_offset + i  - int_from] =␣
↪numpy.string_(datetime.datetime.fromtimestamp(t).strftime('%Y-%m-%d %H:%M:%S.%f')[:-
↪3])

                # Print out execution time:
                t2 = time.time()
                log = open(logfilename,"a")
                log.write(os.linesep + "\t%s converted in %0.3f sec." % (provenance_
↪dset["filename", provenance_offset + i  - int_from], t2 - t1))
                log.close()

    except  Exception, e:
        #log = open(logfilename,"a")
        #log.write(str(e))
        #log.close()
        pass

    finally:
        # Close file:
        infile.close()
```

```
    return provenance_offset + i + 1


def main(argv):
    """
    Converts a set of HIS files into a TDF file (HDF5 Tomo Data Format).

    Parameters
    ----------
    from : scalar, integer
        among all the projections (or sinogram) files, a subset of files can be specified,
        ranging from the parameter "from" to the parameter "to" (see next). In most
        cases, this parameter is 0.

    to : scalar, integer
        among all the projections (or sinogram) files, a subset of files can be specified,
        ranging from the parameter "from" (see previous parameter) to the parameter
        "to". If the value -1 is specified, all the projection files will be considered.

    data_in_path : string
        path of the HIS file of the projections (e.g. "Z:\\sample1.his").

    dark_in_path : string
        path of the HIS file of the flat (e.g. "Z:\\sample1_dark.his").

    flat_in_path : string
        path of the HIS file of the flat (e.g. "Z:\\sample1_flat.his").

    postdark_in_path : string
        path of the HIS file of the flat (e.g. "Z:\\sample1_postdark.his").

    postflat_in_path : string
        path of the HIS file of the flat (e.g. "Z:\\sample1_postflat.his").

    out_file : string
        path with filename of the TDF to create (e.g. "Z:\\sample1.tdf"). WARNING: the program
        does NOT automatically create non-existing folders and subfolders specified in the path.
        Moreover, if a file with the same name already exists it will be automatically deleted and
        overwritten.

    crop_top : scalar, integer
        during the conversion, images can be cropped if required. This parameter specifies the number
        of pixels to crop from the top of the image. Leave 0 for no cropping.

    crop_bottom : scalar, integer
        during the conversion, images can be cropped if required. This parameter specifies the number
        of pixels to crop from the bottom of the image. Leave 0 for no cropping.

    crop_left : scalar, integer
        during the conversion, images can be cropped if required. This parameter specifies the number
```

```
265         of pixels to crop from the left of the image. Leave 0 for no cropping.
266
267     crop_right : scalar, integer
268         during the conversion, images can be cropped if required. This parameter␣
↪specifies the number
269         of pixels to crop from the right of the image. Leave 0 for no cropping.
270
271     privilege_sino : boolean string
272         specify the string "True" if the TDF will privilege a fast read/write of␣
↪sinograms (the most common
273         case), "False" for fast read/write of projections.
274
275     compression : scalar, integer
276         an integer value in the range of [1,9] to be used as GZIP compression factor␣
↪in the HDF5 file, where
277         1 is the minimum compression (and maximum speed) and 9 is the maximum (and␣
↪slow) compression.
278         The value 0 can be specified with the meaning of no compression.
279
280     log_file : string
281         path with filename of a log file (e.g. "R:\\log.txt") where info about the␣
↪conversion is reported.
282
283     Returns
284     -------
285     no return value
286
287     Example
288     -------
289     Example call to convert all the tomo*.tif* projections to a TDF with no cropping␣
↪and minimum compression:
290
291         python his2tdf.py 0 -1 "tomo.his" "dark.his" "flat.his" "postdark.his"
↪"postflat.his" "dataset.tdf" 0 0 0 0
292         True True 1 "S:\\conversion.txt"
293
294     Requirements
295     -------
296     - Python 2.7 with the latest NumPy, SciPy, H5Py.
297     - tdf.py
298
299     Tests
300     -------
301     Tested with WinPython-64bit-2.7.6.3 (Windows) and Anaconda 2.1.0 (Linux 64-bit). ␣
↪
302
303     """
304
305     # Get the from and to number of files to process:
306     int_from = int(argv[0])
307     int_to = int(argv[1]) # -1 means "all files"
308
309     # Get paths:
310     tomo_file = argv[2]
311     dark_file = argv[3]
312     flat_file = argv[4]
313     darkpost_file = argv[5]
314     flatpost_file = argv[6]
```

```
    outfile = argv[7]

    crop_top      = int(argv[8])  # 0 for all means "no cropping"
    crop_bottom = int(argv[9])
    crop_left      = int(argv[10])
    crop_right    = int(argv[11])

    projorder = argv[12]
    if projorder == "True":
        projorder = True
    else:
        projorder = False

    privilege_sino = argv[13]
    if privilege_sino == "True":
        privilege_sino = True
    else:
        privilege_sino = False

    # Get compression factor:
    compr_opts = int(argv[14])
    compressionFlag = True;
    if (compr_opts <= 0):
        compressionFlag = False;
    elif (compr_opts > 9):
        compr_opts = 9

    logfilename = argv[15]

    # Get the files in inpath:
    log = open(logfilename,"w")
    log.write(os.linesep + "\tInput HIS files:")
    log.write(os.linesep + "\t\tProjections: %s" % (tomo_file))
    log.write(os.linesep + "\t\tDark: %s" % (dark_file))
    log.write(os.linesep + "\t\tFlat: %s" % (flat_file))
    log.write(os.linesep + "\t\tPost dark: %s" % (darkpost_file))
    log.write(os.linesep + "\t\tPost flat: %s" % (flatpost_file))
    log.write(os.linesep + "\tOutput TDF file: %s" % (outfile))
    log.write(os.linesep + "\t--------------")
    log.write(os.linesep + "\tCropping:")
    log.write(os.linesep + "\t\tTop: %d pixels" % (crop_top))
    log.write(os.linesep + "\t\tBottom: %d pixels" % (crop_bottom))
    log.write(os.linesep + "\t\tLeft: %d pixels" % (crop_left))
    log.write(os.linesep + "\t\tRight: %d pixels" % (crop_right))
    if (int_to != -1):
        log.write(os.linesep + "\tThe subset [%d,%d] of the input files will be␣
↪considered." % (int_from, int_to))

    if (projorder):
        log.write(os.linesep + "\tProjection order assumed.")
    else:
        log.write(os.linesep + "\tSinogram order assumed.")

    if (privilege_sino):
        log.write(os.linesep + "\tFast I/O for sinograms privileged.")
    else:
        log.write(os.linesep + "\tFast I/O for projections privileged.")
```

```
372
373     if (compressionFlag):
374         log.write(os.linesep + "\tTDF compression factor: %d" % (compr_opts))
375     else:
376         log.write(os.linesep + "\tTDF compression: none.")
377
378     log.write(os.linesep + "\t---------------")
379     log.close()
380
381     # Remove a previous copy of output:
382     if os.path.exists(outfile):
383         log = open(logfilename,"a")
384         log.write(os.linesep + "\tWarning: an output file with the same name was␣
↪overwritten.")
385         os.remove(outfile)
386         log.close()
387
388     # Check input file:
389     if not os.path.exists(tomo_file):
390         log = open(logfilename,"a")
391         log.write(os.linesep + "\tError: input HIS file for projections does not␣
↪exist. Process will end.")
392         log.close()
393         exit()
394
395     # First time get the plan:
396     log = open(logfilename,"a")
397     log.write(os.linesep + "\tPreparing the work plan...")
398     log.close()
399
400     # Get info from projection file:
401     dim1, dim2, dimz, dtype = _getHISdim ( tomo_file )
402
403
404     if ( ((int_to - int_from + 1) > 0) and ((int_to - int_from + 1) < dimz) ):
405         dimz = int_to - int_from + 1
406
407     #dsetshape = (num_files,) + im.shape
408     if projorder:
409         #dsetshape = tdf.get_dset_shape(privilege_sino, im.shape[1], im.shape[0], num_
↪files)
410         dsetshape = tdf.get_dset_shape(dim1 - crop_left - crop_right, dim2 - crop_top␣
↪- crop_bottom, dimz)
411     else:
412         #dsetshape = tdf.get_dset_shape(privilege_sino, im.shape[1], num_files, im.
↪shape[0])
413         dsetshape = tdf.get_dset_shape(dim1 - crop_left - crop_right, dim2 - crop_top␣
↪- crop_bottom, dimz)
414
415     f = getHDF5( outfile, 'w' )
416     print dsetshape
417
418     f.attrs['version'] = '1.0'
419     f.attrs['implements'] = "exchange:provenance"
420     echange_group  = f.create_group( 'exchange' )
421
422     if (compressionFlag):
423         dset = f.create_dataset('exchange/data', dsetshape, dtype, chunks=tdf.get_
↪dset_chunks(dim1 - crop_left - crop_right), compression="gzip", compression_
↪opts=compr_opts, shuffle=True, fletcher32=True)
```

```
    else:
        dset = f.create_dataset('exchange/data', dsetshape, dtype)

    if privilege_sino:
        dset.attrs['axes'] = "y:theta:x"
    else:
        dset.attrs['axes'] = "theta:y:x"

    dset.attrs['min'] = str(numpy.iinfo(dtype).max)
    dset.attrs['max'] = str(numpy.iinfo(dtype).min)

    # Get the total number of files to consider:
    num_darks = 0
    num_flats = 0
    num_postdarks = 0
    num_postflats = 0

    if os.path.exists(dark_file):
        dim1, dim2, num_darks, dtype = _getHISdim ( dark_file )
    if os.path.exists(flat_file):
        dim1, dim2, num_flats, dtype = _getHISdim ( flat_file )
    if os.path.exists(darkpost_file):
        dim1, dim2, num_postdarks, dtype = _getHISdim ( darkpost_file )
    if os.path.exists(flatpost_file):
        dim1, dim2, num_postflats, dtype = _getHISdim ( flatpost_file )

    tot_files = dimz + num_darks + num_flats + num_postdarks + num_postflats

    # Create provenance dataset:
    provenance_dt    = numpy.dtype([("filename", numpy.dtype("S255")), ("timestamp", ␣
↪numpy.dtype("S255"))])
    metadata_group  = f.create_group( 'provenance' )
    provenance_dset = metadata_group.create_dataset('detector_output', (tot_files,),␣
↪dtype=provenance_dt)

    provenance_dset.attrs['tomo_prefix'] = 'tomo';
    provenance_dset.attrs['dark_prefix'] = 'dark';
    provenance_dset.attrs['flat_prefix'] = 'flat';
    provenance_dset.attrs['first_index'] = 1;

    # Handle the metadata:
    if (os.path.isfile(os.path.dirname(tomo_file) + os.sep + 'logfile.xml')):
        with open (os.path.dirname(tomo_file) + os.sep + 'logfile.xml', "r") as file:
            xml_command = file.read()
        tdf.parse_metadata(f, xml_command)

    # Print out about plan preparation:
    first_done = True
    log = open(logfilename,"a")
    log.write(os.linesep + "\tWork plan prepared succesfully.")
    log.close()


    # Get the data from HIS:
    if (num_darks > 0) or (num_postdarks > 0):
        #dsetshape = (num_files,) + im.shape
        if projorder:
            #dsetshape = tdf.get_dset_shape(privilege_sino, im.shape[1], im.shape[0],␣
↪num_files)
```

```
480             dsetshape = tdf.get_dset_shape(dim1 - crop_left - crop_right, dim2 - crop_
↪top - crop_bottom, num_darks + num_postdarks)
481         else:
482             #dsetshape = tdf.get_dset_shape(privilege_sino, im.shape[1], num_files,␣
↪im.shape[0])
483             dsetshape = tdf.get_dset_shape(dim1 - crop_left - crop_right, dim2 - crop_
↪top - crop_bottom, num_darks + num_postdarks)
484 
485         if (compressionFlag):
486             darkdset = f.create_dataset('exchange/data_dark', dsetshape, dtype,␣
↪chunks=tdf.get_dset_chunks(dim1 - crop_left - crop_right), compression="gzip",␣
↪compression_opts=compr_opts, shuffle=True, fletcher32=True)
487         else:
488             darkdset = f.create_dataset('exchange/data_dark', dsetshape, dtype)
489 
490         if privilege_sino:
491             darkdset.attrs['axes'] = "y:theta:x"
492         else:
493             darkdset.attrs['axes'] = "theta:y:x"
494 
495         darkdset.attrs['min'] = str(numpy.iinfo(dtype).max)
496         darkdset.attrs['max'] = str(numpy.iinfo(dtype).min)
497     else:
498         log = open(logfilename,"a")
499         log.write(os.linesep + "\tWarning: dark images (if any) not considered.")
500         log.close()
501 
502     if (num_flats > 0) or (num_postflats > 0):
503 
504         #dsetshape = (num_files,) + im.shape
505         if projorder:
506             #dsetshape = tdf.get_dset_shape(privilege_sino, im.shape[1], im.shape[0],␣
↪num_files)
507             dsetshape = tdf.get_dset_shape(dim1 - crop_left - crop_right, dim2 - crop_
↪top - crop_bottom, num_flats + num_postflats)
508         else:
509             #dsetshape = tdf.get_dset_shape(privilege_sino, im.shape[1], num_files,␣
↪im.shape[0])
510             dsetshape = tdf.get_dset_shape(dim1 - crop_left - crop_right, dim2 - crop_
↪top - crop_bottom, num_flats + num_postflats)
511 
512         if (compressionFlag):
513             flatdset = f.create_dataset('exchange/data_white', dsetshape, dtype,␣
↪chunks=tdf.get_dset_chunks(dim1 - crop_left - crop_right), compression="gzip",␣
↪compression_opts=compr_opts, shuffle=True, fletcher32=True)
514         else:
515             flatdset = f.create_dataset('exchange/data_white', dsetshape, dtype)
516 
517         if privilege_sino:
518             flatdset.attrs['axes'] = "y:theta:x"
519         else:
520             flatdset.attrs['axes'] = "theta:y:x"
521 
522         flatdset.attrs['min'] = str(numpy.iinfo(dtype).max)
523         flatdset.attrs['max'] = str(numpy.iinfo(dtype).min)
524 
525     else:
526         log = open(logfilename,"a")
```

```
        log.write(os.linesep + "\tWarning: flat images (if any) not considered.")
        log.close()

    # Process the HIS:
    provenance_offset = 0

    if num_flats > 0:
        provenance_offset = _processHIS( flat_file, flatset, 0, provenance_dset,␣
↪provenance_offset,
            0, 'flat', crop_top, crop_bottom, crop_left, crop_right, logfilename )
    if num_postflats > 0:
        provenance_offset = _processHIS( flatpost_file, flatdset, num_flats,␣
↪provenance_dset, provenance_offset,
            7, 'flat', crop_top, crop_bottom, crop_left, crop_right, logfilename )

    if num_darks > 0:
        provenance_offset = _processHIS( dark_file, darkdset, 0, provenance_dset,␣
↪provenance_offset,
            0, 'dark', crop_top, crop_bottom, crop_left, crop_right, logfilename )
    if num_postdarks > 0:
        provenance_offset = _processHIS( darkpost_file, darkdset, num_darks,␣
↪provenance_dset, provenance_offset,
            7, 'dark', crop_top, crop_bottom, crop_left, crop_right, logfilename )

    provenance_offset = _processHIS( tomo_file, dset, 0, provenance_dset, provenance_
↪offset,
            0, 'tomo', crop_top, crop_bottom, crop_left, crop_right, logfilename, int_
↪from, int_to )


    # Close TDF:
    f.close()

if __name__ == "__main__":
    main(argv[1:])
```

## exec_preprocessing

This section contains the exec_preprocessing script.

Download file: `exec_preprocessing.py`

```
##########################################################################
#  (C) 2016 Elettra - Sincrotrone Trieste S.C.p.A.. All rights reserved.  #
#                                                                        #
#                                                                        #
#  This file is part of STP-Core, the Python core of SYRMEP Tomo Project,#
#  a software tool for the reconstruction of experimental CT datasets.   #
#                                                                        #
#  STP-Core is free software: you can redistribute it and/or modify it   #
#  under the terms of the GNU General Public License as published by the #
#  Free Software Foundation, either version 3 of the License, or (at your#
#  option) any later version.                                            #
#                                                                        #
#  STP-Core is distributed in the hope that it will be useful, but WITHOUT#
#  ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or #
```

```
# FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License    #
# for more details.                                                       #
#                                                                         #
# You should have received a copy of the GNU General Public License       #
# along with STP-Core. If not, see <http://www.gnu.org/licenses/>.        #
#                                                                         #
###########################################################################

#
# Author: Francesco Brun
# Last modified: August, 8th 2016
#

from sys import argv, exit
from os import remove, sep,  linesep
from os.path import exists
from numpy import float32, amin, amax, isscalar
from time import time
from multiprocessing import Process, Lock

# pystp-specific:
from stp_core.preprocess.extfov_correction import extfov_correction
from stp_core.preprocess.flat_fielding import flat_fielding
from stp_core.preprocess.dynamic_flatfielding import dff_prepare_plan, dynamic_flat_
↪fielding
from stp_core.preprocess.ring_correction import ring_correction
from stp_core.preprocess.extract_flatdark import extract_flatdark, _medianize

from h5py import File as getHDF5

# pystp-specific:
import stp_core.io.tdf as tdf


def _write_data(lock, im, index, outfile, outshape, outtype, logfilename, cputime,␣
↪itime):

    lock.acquire()
    try:
        t0 = time()
        f_out = getHDF5( outfile, 'a' )
        f_out_dset = f_out.require_dataset('exchange/data', outshape, outtype,␣
↪chunks=tdf.get_dset_chunks(outshape[0]))
        tdf.write_sino(f_out_dset,index,im.astype(float32))

        # Set minimum and maximum:
        if ( amin(im[:]) < float(f_out_dset.attrs['min']) ):
            f_out_dset.attrs['min'] = str(amin(im[:]))
        if ( amax(im[:]) > float(f_out_dset.attrs['max'])):
            f_out_dset.attrs['max'] = str(amax(im[:]))
        f_out.close()
        t1 = time()

        # Print out execution time:
        log = open(logfilename,"a")
        log.write(linesep + "\tsino_%s processed (CPU: %0.3f sec - I/O: %0.3f sec)."
↪% (str(index).zfill(4), cputime, t1 - t0 + itime))
```

```
68          log.close()
69
70      finally:
71          lock.release()
72
73  def _process (lock, int_from, int_to, infile, outfile, outshape, outtype, skipflat,␣
    ↪plan, norm_sx, norm_dx, flat_end,
74                half_half, half_half_line, ext_fov, ext_fov_rot_right, ext_fov_overlap,␣
    ↪ringrem, dynamic_ff, EFF,
75                filtEFF, im_dark, logfilename):
76
77      # Process the required subset of images:
78      for i in range(int_from, int_to + 1):
79
80          # Read input image:
81          t0 = time()
82          f_in = getHDF5(infile, 'r')
83          if "/tomo" in f_in:
84              dset = f_in['tomo']
85          else:
86              dset = f_in['exchange/data']
87          im = tdf.read_sino(dset,i).astype(float32)
88          f_in.close()
89          t1 = time()
90
91          # Perform pre-processing (flat fielding, extended FOV, ring removal):
92          if not skipflat:
93              if dynamic_ff:
94                  # Dynamic flat fielding with downsampling = 2:
95                  im = dynamic_flat_fielding(im, i, EFF, filtEFF, 2, im_dark, norm_sx,␣
    ↪norm_dx)
96              else:
97                  im = flat_fielding(im, i, plan, flat_end, half_half, half_half_line,␣
    ↪norm_sx, norm_dx)
98          im = extfov_correction(im, ext_fov, ext_fov_rot_right, ext_fov_overlap)
99          if not skipflat and not dynamic_ff:
100             im = ring_correction (im, ringrem, flat_end, plan['skip_flat_after'],␣
    ↪half_half, half_half_line, ext_fov)
101         else:
102             im = ring_correction (im, ringrem, False, False, half_half, half_half_
    ↪line, ext_fov)
103         t2 = time()
104
105         # Save processed image to HDF5 file (atomic procedure - lock used):
106         _write_data(lock, im, i, outfile, outshape, outtype, logfilename, t2 - t1, t1␣
    ↪- t0)
107
108
109 def main(argv):
110     """To do...
111
112     Usage
113     -----
114
115     Parameters
116     ----------
117
118     Example
```

```
    -------
    The following line processes the first ten TIFF files of input path
    "/home/in" and saves the processed files to "/home/out" with the
    application of the Boin and Haibel filter with smoothing via a Butterworth
    filter of order 4 and cutoff frequency 0.01:

    destripe /home/in /home/out 1 10 1 0.01 4

    """
    lock = Lock()

    # Get the from and to number of files to process:
    int_from = int(argv[0])
    int_to = int(argv[1])

    # Get paths:
    infile = argv[2]
    outfile = argv[3]

    # Normalization parameters:
    norm_sx = int(argv[4])
    norm_dx = int(argv[5])

    # Params for flat fielding with post flats/darks:
    flat_end = True if argv[6] == "True" else False
    half_half = True if argv[7] == "True" else False
    half_half_line = int(argv[8])

    # Params for extended FOV:
    ext_fov = True if argv[9] == "True" else False
    ext_fov_rot_right = argv[10]
    if ext_fov_rot_right == "True":
        ext_fov_rot_right = True
        if (ext_fov):
            norm_sx = 0
    else:
        ext_fov_rot_right = False
        if (ext_fov):
            norm_dx = 0
    ext_fov_overlap = int(argv[11])

    # Method and parameters coded into a string:
    ringrem = argv[12]

    # Flat fielding method (conventional or dynamic):
    dynamic_ff = True if argv[13] == "True" else False

    # Nr of threads and log file:
    nr_threads = int(argv[14])
    logfilename = argv[15]




    # Log input parameters:
    log = open(logfilename,"w")
    log.write(linesep + "\tInput TDF file: %s" % (infile))
```

```
    log.write(linesep + "\tOutput TDF file: %s" % (outfile))
    log.write(linesep + "\t---------------")
    log.write(linesep + "\tOpening input dataset...")
    log.close()

    # Remove a previous copy of output:
    if exists(outfile):
        remove(outfile)

    # Open the HDF5 file:
    f_in = getHDF5(infile, 'r')


    if "/tomo" in f_in:
        dset = f_in['tomo']

        tomoprefix = 'tomo'
        flatprefix = 'flat'
        darkprefix = 'dark'
    else:
        dset = f_in['exchange/data']
        if "/provenance/detector_output" in f_in:
            prov_dset = f_in['provenance/detector_output']

            tomoprefix = prov_dset.attrs['tomo_prefix']
            flatprefix = prov_dset.attrs['flat_prefix']
            darkprefix = prov_dset.attrs['dark_prefix']

    num_proj = tdf.get_nr_projs(dset)
    num_sinos = tdf.get_nr_sinos(dset)

    if (num_sinos == 0):
        log = open(logfilename,"a")
        log.write(linesep + "\tNo projections found. Process will end.")
        log.close()
        exit()

    # Check extrema (int_to == -1 means all files):
    if ((int_to >= num_sinos) or (int_to == -1)):
        int_to = num_sinos - 1

    # Prepare the work plan for flat and dark images:
    log = open(logfilename,"a")
    log.write(linesep + "\t---------------")
    log.write(linesep + "\tPreparing the work plan...")
    log.close()

    # Extract flat and darks:
    skipflat = False
    skipdark = False

    # Following variables make sense only for dynamic flat fielding:
    EFF = -1
    filtEFF = -1
    im_dark = -1

    # Following variable makes sense only for conventional flat fielding:
```

```
    plan = -1

    if not dynamic_ff:
        plan = extract_flatdark(f_in, flat_end, logfilename)
        if (isscalar(plan['im_flat']) and isscalar(plan['im_flat_after']) ):
            skipflat = True
        else:
            skipflat = False
    else:
        # Dynamic flat fielding:
        if "/tomo" in f_in:
            if "/flat" in f_in:
                flat_dset = f_in['flat']
                if "/dark" in f_in:
                    im_dark = _medianize(f_in['dark'])
                else:
                    skipdark = True
            else:
                skipflat = True # Nothing to do in this case
        else:
            if "/exchange/data_white" in f_in:
                flat_dset = f_in['/exchange/data_white']
                if "/exchange/data_dark" in f_in:
                    im_dark = _medianize(f_in['/exchange/data_dark'])
                else:
                    skipdark = True
            else:
                skipflat = True # Nothing to do in this case

        # Prepare plan for dynamic flat fielding with 16 repetitions:
        if not skipflat:
            EFF, filtEFF = dff_prepare_plan(flat_dset, 16, im_dark)

    # Outfile shape can be determined only after first processing in ext FOV mode:
    if (ext_fov):

        # Read input sino:
        idx = num_sinos / 2
        im = tdf.read_sino(dset,idx).astype(float32)
        im = extfov_correction(im, ext_fov, ext_fov_rot_right, ext_fov_overlap)

        # Get the corrected outshape:
        outshape = tdf.get_dset_shape(im.shape[1], num_sinos, im.shape[0])

    else:
        # Get the corrected outshape (in this case it's easy):
        im = tdf.read_tomo(dset,0).astype(float32)
        outshape = tdf.get_dset_shape(im.shape[1], im.shape[0], num_proj)

    # Create the output HDF5 file:
    f_out = getHDF5(outfile, 'w')
    f_out_dset = f_out.create_dataset('exchange/data', outshape, im.dtype)
    f_out_dset.attrs['min'] = str(amin(im[:]))
    f_out_dset.attrs['max'] = str(amax(im[:]))
    f_out_dset.attrs['version'] = '1.0'
    f_out_dset.attrs['axes'] = "y:theta:x"

```

```
    f_out.close()
    f_in.close()

    # Log infos:
    log = open(logfilename,"a")
    log.write(linesep + "\tWork plan prepared correctly.")
    log.write(linesep + "\t---------------")
    log.write(linesep + "\tPerforming pre processing...")
    log.close()

    # Run several threads for independent computation without waiting for threads completion:
    for num in range(nr_threads):
        start = (num_sinos / nr_threads)*num
        if (num == nr_threads - 1):
            end = num_sinos - 1
        else:
            end = (num_sinos / nr_threads)*(num + 1) - 1
        Process(target=_process, args=(lock, start, end, infile, outfile, outshape, im.dtype, skipflat, plan, norm_sx,
                norm_dx, flat_end, half_half, half_half_line, ext_fov, ext_fov_rot_right, ext_fov_overlap, ringrem,
                dynamic_ff, EFF, filtEFF, im_dark, logfilename )).start()


    #start = int_from # 0
    #end = int_to # num_sinos - 1
    #_process(lock, start, end, infile, outfile, outshape, im.dtype, skipflat, plan, norm_sx,
    #           norm_dx, flat_end, half_half, half_half_line, ext_fov, ext_fov_rot_right, ext_fov_overlap, ringrem,
    #           dynamic_ff, EFF, filtEFF, im_dark, logfilename)

    #255 256 C:\Temp\BrunGeorgos.tdf C:\Temp\BrunGeorgos_corr.tdf 0 0 True True 900 False False 0 rivers:11;0 False 1 C:\Temp\log_00.txt


if __name__ == "__main__":
    main(argv[1:])
```

## exec_reconstruct

This section contains the exec_reconstruct script.

Download file: `exec_reconstruct.py`

```
#########################################################################
# (C) 2016 Elettra - Sincrotrone Trieste S.C.p.A.. All rights reserved.   #
#                                                                         #
#                                                                         #
# This file is part of STP-Core, the Python core of SYRMEP Tomo Project,  #
# a software tool for the reconstruction of experimental CT datasets.     #
#                                                                         #
# STP-Core is free software: you can redistribute it and/or modify it     #
# under the terms of the GNU General Public License as published by the   #
# Free Software Foundation, either version 3 of the License, or (at your  #
```

```
# option) any later version.                                                  #
#                                                                             #
# STP-Core is distributed in the hope that it will be useful, but WITHOUT     #
# ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or       #
# FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License        #
# for more details.                                                           #
#                                                                             #
# You should have received a copy of the GNU General Public License           #
# along with STP-Core. If not, see <http://www.gnu.org/licenses/>.            #
#                                                                             #
###############################################################################

#
# Author: Francesco Brun
# Last modified: July, 8th 2016
#

# python:
from sys import argv, exit
from os import remove, sep, makedirs, linesep
from os.path import basename, exists
from numpy import finfo, copy, float32, double, amin, amax, tile, concatenate, log as␣
↪nplog
from numpy import arange, meshgrid, isscalar, ndarray, pi, roll
from time import time
from multiprocessing import Process, Lock

# pystp-specific:
from stp_core.preprocess.extfov_correction import extfov_correction
from stp_core.preprocess.flat_fielding import flat_fielding
from stp_core.preprocess.ring_correction import ring_correction
from stp_core.preprocess.extract_flatdark import extract_flatdark, _medianize
from stp_core.preprocess.dynamic_flatfielding import dff_prepare_plan, dynamic_flat_
↪fielding

from stp_core.reconstruct.rec_astra import recon_astra_fbp, recon_astra_iterative
from stp_core.reconstruct.rec_fista_tv import recon_fista_tv
from stp_core.reconstruct.rec_mr_fbp import recon_mr_fbp
from stp_core.reconstruct.rec_gridrec import recon_gridrec

from stp_core.postprocess.postprocess import postprocess

from stp_core.utils.padding import upperPowerOfTwo, padImage, padSmoothWidth

from tifffile import imread, imsave
from h5py import File as getHDF5

# pystp-specific:
import stp_core.io.tdf as tdf


def reconstruct(im, angles, offset, logtransform, param1, circle, scale, pad, method,␣
↪rolling, roll_shift,
                zerone_mode, dset_min, dset_max, decim_factor, downsc_factor, corr_
↪offset):
    """Reconstruct a sinogram with FBP algorithm (from ASTRA toolbox).

    Parameters
```

```
    ----------
    im1 : array_like
        Sinogram image data as numpy array.
    center : float
        Offset of the center of rotation to use for the tomographic
        reconstruction with respect to the half of sinogram width
        (default=0, i.e. half width).
    logtransform : boolean
        Apply logarithmic transformation before reconstruction (default=True).
    filter : string
        Filter to apply before the application of the reconstruction algorithm.␣
↪Filter
        types are: ram-lak, shepp-logan, cosine, hamming, hann, tukey, lanczos,␣
↪triangular,
        gaussian, barlett-hann, blackman, nuttall, blackman-harris, blackman-nuttall,
        flat-top, kaiser, parzen.
    circle : boolean
        Create a circle in the reconstructed image and set to zero pixels outside the
        circle (default=False).

    Example (using tiffile.py)
    --------------------------
    >>> # Read input (uncorrected) sinogram
    >>> sino_im1 = imread('sino_0050.tif')
    >>>
    >>> # Get flat and dark correction images:
    >>> im_dark = medianize("\project\tomo", "dark*.tif")
    >>> im_flat = medianize("\project\tomo", "flat*.tif")
    >>>
    >>> # Perform flat fielding and normalization:
    >>> sino_im = normalize(sino_im1, (10,10), (0,0), im_dark, im_flat, 50)
    >>>
    >>> # Actual reconstruction:
    >>> out = reconstruct_fbp(sino_im, -3.0)
    >>>
    >>> # Save output slice:
    >>> imsave('slice_0050.tif', out)

    """

    # Copy images and ensure they are of type float32:
    #im_f = copy(im.astype(float32))
    im_f = im.astype(float32)

    # Decimate projections if required:
    if decim_factor > 1:
        im_f = im_f[::decim_factor,:]

    # Upscale projections (if required):
    if (abs(scale - 1.0) > finfo(float32).eps):
        siz_orig1 = im_f.shape[1]
        im_f = imresize(im_f, (im_f.shape[0], int(round(scale * im_f.shape[1]))),␣
↪interp='bicubic', mode='F')
        offset = int(offset * scale)

    # Apply transformation for changes in the center of rotation:
    if (offset != 0):
        if (offset >= 0):
```

```
            im_f = im_f[:,:-offset]

            tmp = im_f[:,0] # Get first column
            tmp = tile(tmp, (offset,1)) # Replicate the first column the right number of times
            im_f = concatenate((tmp.T,im_f), axis=1) # Concatenate tmp before the image

        else:
            im_f = im_f[:,abs(offset):]

            tmp = im_f[:,im_f.shape[1] - 1] # Get last column
            tmp = tile(tmp, (abs(offset),1)) # Replicate the last column the right number of times
            im_f = concatenate((im_f,tmp.T), axis=1) # Concatenate tmp after the image

    # Downscale projections (without pixel averaging):
    if downsc_factor > 1:
        im_f = im_f[:,::downsc_factor]

    # Sinogram rolling (if required).  It doesn't make sense in limited angle tomography, so check if 180 or 360:
    if ((rolling == True) and (roll_shift > 0)):
        if ( (angles - pi) < finfo(float32).eps ):
            # Flip the last rows:
            im_f[-roll_shift:,:] = im_f[-roll_shift:,::-1]
            # Now roll the sinogram:
            im_f = roll(im_f, roll_shift, axis=0)
        elif ((angles - pi*2.0) < finfo(float32).eps):
            # Only roll the sinogram:
            im_f = roll(im_f, roll_shift, axis=0)

    # Scale image to [0,1] range (if required):
    if (zerone_mode):

        #print dset_min
        #print dset_max
        #print numpy.amin(im_f[:])
        #print numpy.amax(im_f[:])
        #im_f = (im_f - dset_min) / (dset_max - dset_min)

        # Cheating the whole process:
        im_f = (im_f - numpy.amin(im_f[:])) / (numpy.amax(im_f[:]) - numpy.amin(im_f[:]))

    # Apply log transform:
    if (logtransform == True):
        im_f[im_f <= finfo(float32).eps] = finfo(float32).eps
        im_f = -nplog(im_f + corr_offset)

    # Replicate pad image to double the width:
    if (pad):

        dim_o = im_f.shape[1]
        n_pad = im_f.shape[1] + im_f.shape[1] / 2
        marg  = (n_pad - dim_o) / 2

        # Pad image:
```

```
        im_f = padSmoothWidth(im_f, n_pad)

    # Perform the actual reconstruction:
    if (method.startswith('FBP')):
        im_f = recon_astra_fbp(im_f, angles, method, param1)
    elif (method == 'MR-FBP_CUDA'):
        im_f = recon_mr_fbp(im_f, angles)
    elif (method == 'FISTA-TV_CUDA'):
        im_f = recon_fista_tv(im_f, angles, param1, param1)
    else:
        im_f = recon_astra_iterative(im_f, angles, method, param1, zerone_mode)


    # Crop:
    if (pad):
        im_f = im_f[marg:dim_o + marg, marg:dim_o + marg]

    # Resize (if necessary):
    if (abs(scale - 1.0) > finfo(float32).eps):
        im_f = imresize(im_f, (siz_orig1, siz_orig1), interp='nearest', mode='F')

    # Return output:
    return im_f.astype(float32)

def reconstruct_gridrec(im1, im2, angles, offset, logtransform, param1, circle, scale,
↪ pad, rolling, roll_shift,
                zerone_mode, dset_min, dset_max, decim_factor, downsc_factor, corr_
↪offset):
    """Reconstruct a sinogram with FBP algorithm (from ASTRA toolbox).

    Parameters
    ----------
    im1 : array_like
        Sinogram image data as numpy array.
    center : float
        Offset of the center of rotation to use for the tomographic
        reconstruction with respect to the half of sinogram width
        (default=0, i.e. half width).
    logtransform : boolean
        Apply logarithmic transformation before reconstruction (default=True).
    filter : string
        Filter to apply before the application of the reconstruction algorithm.␣
↪Filter
        types are: ram-lak, shepp-logan, cosine, hamming, hann, tukey, lanczos,␣
↪triangular,
        gaussian, barlett-hann, blackman, nuttall, blackman-harris, blackman-nuttall,
        flat-top, kaiser, parzen.
    circle : boolean
        Create a circle in the reconstructed image and set to zero pixels outside the
        circle (default=False).

    Example (using tiffile.py)
    --------------------------
    >>> # Read input (uncorrected) sinogram
    >>> sino_im1 = imread('sino_0050.tif')
    >>>
    >>> # Get flat and dark correction images:
    >>> im_dark = medianize("\project\tomo", "dark*.tif")
```

```
    >>> im_flat = medianize("\project\tomo", "flat*.tif")
    >>>
    >>> # Perform flat fielding and normalization:
    >>> sino_im = normalize(sino_im1, (10,10), (0,0), im_dark, im_flat, 50)
    >>>
    >>> # Actual reconstruction:
    >>> out = reconstruct_fbp(sino_im, -3.0)
    >>>
    >>> # Save output slice:
    >>> imsave('slice_0050.tif', out)

    """
    # Ensure images are of type float32:
    im_f1 = im1.astype(float32)
    im_f2 = im2.astype(float32)

    # Decimate projections if required:
    if decim_factor > 1:
        im_f1 = im_f1[::decim_factor,:]
        im_f2 = im_f2[::decim_factor,:]

    # Upscale projections (if required):
    if (abs(scale - 1.0) > finfo(float32).eps):
        siz_orig1 = im_f.shape[1]
        im_f1 = imresize(im_f1, (im_f1.shape[0], int(round(scale * im_f1.shape[1]))), interp='bicubic', mode='F')
        im_f2 = imresize(im_f2, (im_f2.shape[0], int(round(scale * im_f2.shape[1]))), interp='bicubic', mode='F')
        offset = int(offset * scale)

    # Apply transformation for changes in the center of rotation:
    if (offset != 0):
        if (offset >= 0):
            im_f1 = im_f1[:,:-offset]

            tmp = im_f1[:,0] # Get first column
            tmp = tile(tmp, (offset,1)) # Replicate the first column the right number of times
            im_f1 = concatenate((tmp.T,im_f1), axis=1) # Concatenate tmp before the image

            im_f2 = im_f2[:,:-offset]

            tmp = im_f2[:,0] # Get first column
            tmp = tile(tmp, (offset,1)) # Replicate the first column the right number of times
            im_f2 = concatenate((tmp.T,im_f2), axis=1) # Concatenate tmp before the image

        else:
            im_f1 = im_f1[:,abs(offset):]

            tmp = im_f1[:,im_f1.shape[1] - 1] # Get last column
            tmp = tile(tmp, (abs(offset),1)) # Replicate the last column the right number of times
            im_f1 = concatenate((im_f1,tmp.T), axis=1) # Concatenate tmp after the image

```

```
277             im_f2 = im_f2[:,abs(offset):]
278 
279             tmp = im_f2[:,im_f2.shape[1] - 1] # Get last column
280             tmp = tile(tmp, (abs(offset),1)) # Replicate the last column the right␣
↪number of times
281             im_f2 = concatenate((im_f2,tmp.T), axis=1) # Concatenate tmp after the␣
↪image
282 
283     # Downscale projections (without pixel averaging):
284     if downsc_factor > 1:
285         im_f1 = im_f1[:,::downsc_factor]
286         im_f2 = im_f2[:,::downsc_factor]
287 
288     # Sinogram rolling (if required).  It doesn't make sense in limited angle␣
↪tomography, so check if 180 or 360:
289     if ((rolling == True) and (roll_shift > 0)):
290         if ( (angles - pi) < finfo(float32).eps ):
291             # Flip the last rows:
292             im_f1[-roll_shift:,:] = im_f1[-roll_shift:,::-1]
293             im_f2[-roll_shift:,:] = im_f2[-roll_shift:,::-1]
294             # Now roll the sinogram:
295             im_f1 = roll(im_f1, roll_shift, axis=0)
296             im_f2 = roll(im_f2, roll_shift, axis=0)
297         elif ((angles - pi*2.0) < finfo(float32).eps):
298             # Only roll the sinogram:
299             im_f1 = roll(im_f1, roll_shift, axis=0)
300             im_f2 = roll(im_f2, roll_shift, axis=0)
301 
302     # Scale image to [0,1] range (if required):
303     if (zerone_mode):
304 
305         #print dset_min
306         #print dset_max
307         #print numpy.amin(im_f[:])
308         #print numpy.amax(im_f[:])
309         #im_f = (im_f - dset_min) / (dset_max - dset_min)
310 
311         # Cheating the whole process:
312         im_f1 = (im_f1 - numpy.amin(im_f1[:])) / (numpy.amax(im_f1[:]) - numpy.
↪amin(im_f1[:]))
313         im_f2 = (im_f2 - numpy.amin(im_f2[:])) / (numpy.amax(im_f2[:]) - numpy.
↪amin(im_f2[:]))
314 
315 
316     # Apply log transform:
317     if (logtransform == True):
318         im_f1[im_f1 <= finfo(float32).eps] = finfo(float32).eps
319         im_f1 = -nplog(im_f1 + corr_offset)
320 
321         im_f2[im_f2 <= finfo(float32).eps] = finfo(float32).eps
322         im_f2 = -nplog(im_f2 + corr_offset)
323 
324     # Replicate pad image to double the width:
325     if (pad):
326 
327         dim_o = im_f1.shape[1]
328         n_pad = im_f1.shape[1] + im_f1.shape[1] / 2
329         marg  = (n_pad - dim_o) / 2
```

```
330
331         # Pad image:
332         im_f1 = padSmoothWidth(im_f1, n_pad)
333         im_f2 = padSmoothWidth(im_f2, n_pad)
334
335     # Perform the actual reconstruction:
336     [im_f1, im_f2] = recon_gridrec(im_f1, im_f2, angles, param1)
337
338
339     # Crop:
340     if (pad):
341         im_f1 = im_f1[marg:dim_o + marg, marg:dim_o + marg]
342         im_f2 = im_f2[marg:dim_o + marg, marg:dim_o + marg]
343
344     # Resize (if necessary):
345     if (abs(scale - 1.0) > finfo(float32).eps):
346         im_f1 = imresize(im_f1, (siz_orig1, siz_orig1), interp='nearest', mode='F')
347         im_f2 = imresize(im_f2, (siz_orig1, siz_orig1), interp='nearest', mode='F')
348
349     # Return output:
350     return [im_f1.astype(float32), im_f2.astype(float32)]
351
352 def write_log(lock, fname, logfilename, cputime, iotime):
353     """To do...
354
355     """
356     lock.acquire()
357     try:
358         # Print out execution time:
359         log = open(logfilename,"a")
360         log.write(linesep + "\t%s reconstructed (CPU: %0.3f sec - I/O: %0.3f sec)." %␣
    ↪(basename(fname), cputime, iotime))
361         log.close()
362
363     finally:
364         lock.release()
365
366 def write_log_gridrec(lock, fname1, fname2, logfilename, cputime, iotime):
367     """To do...
368
369     """
370     lock.acquire()
371     try:
372         # Print out execution time:
373         log = open(logfilename,"a")
374         log.write(linesep + "\t%s reconstructed (CPU: %0.3f sec - I/O: %0.3f sec)." %␣
    ↪(basename(fname1), cputime/2, iotime/2))
375         log.write(linesep + "\t%s reconstructed (CPU: %0.3f sec - I/O: %0.3f sec)." %␣
    ↪(basename(fname2), cputime/2, iotime/2))
376         log.close()
377
378     finally:
379         lock.release()
380
381 def process_gridrec(lock, int_from, int_to, num_sinos, infile, outpath, preprocessing_
    ↪required, skipflat, corr_plan,
382             norm_sx, norm_dx, flat_end, half_half,
383             half_half_line, ext_fov, ext_fov_rot_right, ext_fov_overlap, ringrem,␣
    ↪angles, angles_projfrom, angles_projto,
```

```
            offset, logtransform, param1, circle, scale, pad, rolling, roll_shift, zerone_mode, dset_min, dset_max, decim_factor,
            downsc_factor, corr_offset, postprocess_required, convert_opt, crop_opt, dynamic_ff, EFF, filtEFF, im_dark,
            outprefix, logfilename):
    """To do...

    """
    # Process the required subset of images:
    for i in range(int_from, int_to + 1, 2):

        # Read two sinograms:
        t0 = time()
        f_in = getHDF5(infile, 'r')
        if "/tomo" in f_in:
            dset = f_in['tomo']
        else:
            dset = f_in['exchange/data']
        im1 = tdf.read_sino(dset,i).astype(float32)
        if ( (i + 1) <= (int_to + 1) ):
            im2 = tdf.read_sino(dset,i + 1).astype(float32)
        else:
            im2 = im1
        f_in.close()
        t1 = time()


        # Apply projection removal (if required):
        im1 = im1[angles_projfrom:angles_projto, :]
        im2 = im2[angles_projfrom:angles_projto, :]

        # Perform the preprocessing of the sinograms (if required):
        if (preprocessing_required):
            if not skipflat:
                if dynamic_ff:
                    # Dynamic flat fielding with downsampling = 2:
                    im1 = dynamic_flat_fielding(im1, i, EFF, filtEFF, 2, im_dark, norm_sx, norm_dx)
                else:
                    im1 = flat_fielding (im1, i, corr_plan, flat_end, half_half, half_half_line, norm_sx, norm_dx).astype(float32)
            im1 = extfov_correction (im1, ext_fov, ext_fov_rot_right, ext_fov_overlap)
            if not skipflat:
                im1 = ring_correction (im1, ringrem, flat_end, corr_plan['skip_flat_after'], half_half, half_half_line, ext_fov)
            else:
                im1 = ring_correction (im1, ringrem, False, False, half_half, half_half_line, ext_fov)

            if not skipflat:
                if dynamic_ff:
                    # Dynamic flat fielding with downsampling = 2:
                    im2 = dynamic_flat_fielding(im2, i, EFF, filtEFF, 2, im_dark, norm_sx, norm_dx)
                else:
                    im2 = flat_fielding (im2, i + 1, corr_plan, flat_end, half_half, half_half_line, norm_sx, norm_dx).astype(float32)
            im2 = extfov_correction (im2, ext_fov, ext_fov_rot_right, ext_fov_overlap)
```

```
            if not skipflat and not dynamic_ff:
                im2 = ring_correction (im2, ringrem, flat_end, corr_plan['skip_flat_
↪after'], half_half, half_half_line, ext_fov)
            else:
                im2 = ring_correction (im2, ringrem, False, False, half_half, half_
↪half_line, ext_fov)


        # Actual reconstruction:
        [im1, im2] = reconstruct_gridrec(im1, im2, angles, offset, logtransform,␣
↪param1, circle, scale, pad, rolling, roll_shift,
                        zerone_mode, dset_min, dset_max, decim_factor, downsc_factor,␣
↪corr_offset)

        # Appy post-processing (if required):
        if postprocess_required:
            im1 = postprocess(im1, convert_opt, crop_opt, circle)
            im2 = postprocess(im2, convert_opt, crop_opt, circle)
        else:
            # Create the circle mask for fancy output:
            if (circle == True):
                siz = im1.shape[1]
                if siz % 2:
                    rang = arange(-siz / 2 + 1, siz / 2 + 1)
                else:
                    rang = arange(-siz / 2,siz / 2)
                x,y = meshgrid(rang,rang)
                z = x ** 2 + y ** 2
                a = (z < (siz / 2 - int(round(abs(offset)/downsc_factor)) ) ** 2)

                im1 = im1 * a
                im2 = im2 * a

        # Write down reconstructed slices:
        t2 = time()

        fname1 = outpath + outprefix + '_' + str(i).zfill(4) + '.tif'
        imsave(fname1, im1)

        fname2 = outpath + outprefix + '_' + str(i + 1).zfill(4) + '.tif'
        imsave(fname2, im2)

        t3 = time()

        # Write log (atomic procedure - lock used):
        write_log_gridrec(lock, fname1, fname2, logfilename, t2 - t1, (t3 - t2) + (t1␣
↪- t0) )


def process(lock, int_from, int_to, num_sinos, infile, outpath, preprocessing_
↪required, skipflat, corr_plan, norm_sx, norm_dx,
            flat_end, half_half,
            half_half_line, ext_fov, ext_fov_rot_right, ext_fov_overlap, ringrem,␣
↪angles, angles_projfrom, angles_projto,
            offset, logtransform, param1, circle, scale, pad, method, rolling, roll_
↪shift, zerone_mode, dset_min, dset_max, decim_factor,
            downsc_factor, corr_offset, postprocess_required, convert_opt, crop_opt,␣
↪dynamic_ff, EFF, filtEFF, im_dark,
```

```
483             outprefix, logfilename):
484     """To do...
485 
486     """
487     # Process the required subset of images:
488     for i in range(int_from, int_to + 1):
489 
490         # Perform reconstruction (on-the-fly preprocessing and phase retrieval, if␣
    ↪required):
491         #if (phaseretrieval_required):
492 
493         #    # Load into memory a bunch of sinograms:
494         #    t0 = time()
495 
496         #    # Open the TDF file for reading:
497         #    f_in = getHDF5(infile, 'r')
498         #    if "/tomo" in f_in:
499         #        dset = f_in['tomo']
500         #    else:
501         #        dset = f_in['exchange/data']
502 
503         #    # Prepare the data structure according to the approximation window:
504         #    tmp_im = numpy.empty((tdf.get_nr_projs(dset),tdf.get_det_size(dset),␣
    ↪approx_win), dtype=float32)
505 
506         #    # Load the temporary data structure reading the input TDF file:
507         #    # (It can be parallelized Open-MP style)
508         #    ct = 0
509         #    for j in range(i - approx_win/2, i + approx_win/2 + 1):
510         #        if (j < 0):
511         #            j = 0
512         #        if (j >= num_sinos):
513         #            j = num_sinos - 1
514         #        a = tdf.read_sino(dset,j).astype(float32)
515         #        tmp_im[:,:,ct] = a
516         #        ct = ct + 1
517 
518         #    # Close the TDF file:
519         #    f_in.close()
520         #    t1 = time()
521 
522         #    # Perform the processing:
523         #    if (preprocessing_required):
524         #        ct = 0
525         #        # (It can be parallelized Open-MP style)
526         #        for j in range(i - approx_win/2, i + approx_win/2 + 1):
527         #            if (j < 0):
528         #                j = 0
529         #            if (j >= num_sinos):
530         #                j = num_sinos - 1
531 
532         #            tmp_im[:,:,ct] = flat_fielding (tmp_im[:,:,ct], j, corr_plan,␣
    ↪flat_end, half_half, half_half_line, norm_sx, norm_dx).astype(float32)
533         #            tmp_im[:,:,ct] = extfov_correction (tmp_im[:,:,ct], ext_fov, ext_
    ↪fov_rot_right, ext_fov_overlap).astype(float32)
534         #            tmp_im[:,:,ct] = ring_correction (tmp_im[:,:,ct], ringrem, flat_
    ↪end, corr_plan['skip_flat_after'], half_half, half_half_line, ext_fov).
    ↪astype(float32)
```

```
535         #           ct = ct + 1
536
537         #   # Perform phase retrieval:
538         #   # (It can be parallelized Open-MP style)
539         #   for ct in range(0, tmp_im.shape[0]):
540
541         #       tmp_im[ct,:,:] = phase_retrieval(tmp_im[ct,:,:].T, phrt_plan).
    ↪astype(float32).T
542         #       ct = ct + 1
543
544         #   # Extract the central processed sinogram:
545         #   im = tmp_im[:,:,approx_win/2]
546
547         #else:
548
549         # Read only one sinogram:
550         t0 = time()
551         f_in = getHDF5(infile, 'r')
552         if "/tomo" in f_in:
553             dset = f_in['tomo']
554         else:
555             dset = f_in['exchange/data']
556         im = tdf.read_sino(dset,i).astype(float32)
557         f_in.close()
558         t1 = time()
559
560         # Apply projection removal (if required):
561         im = im[angles_projfrom:angles_projto, :]
562
563         # Perform the preprocessing of the sinogram (if required):
564         if (preprocessing_required):
565             if not skipflat:
566                 if dynamic_ff:
567                     # Dynamic flat fielding with downsampling = 2:
568                     im = dynamic_flat_fielding(im, i, EFF, filtEFF, 2, im_dark, norm_
    ↪sx, norm_dx).astype(float32)
569                 else:
570                     im = flat_fielding (im, i, corr_plan, flat_end, half_half, half_
    ↪half_line, norm_sx, norm_dx).astype(float32)
571             im = extfov_correction (im, ext_fov, ext_fov_rot_right, ext_fov_overlap)
572             if not skipflat and not dynamic_ff:
573                 im = ring_correction (im, ringrem, flat_end, corr_plan['skip_flat_
    ↪after'], half_half, half_half_line, ext_fov)
574             else:
575                 im = ring_correction (im, ringrem, False, False, half_half, half_half_
    ↪line, ext_fov)
576
577
578         # Actual reconstruction:
579         im = reconstruct(im, angles, offset, logtransform, param1, circle, scale, pad,
    ↪ method, rolling, roll_shift,
580                          zerone_mode, dset_min, dset_max, decim_factor, downsc_factor,␣
    ↪corr_offset).astype(float32)
581
582         # Apply post-processing (if required):
583         if postprocess_required:
584             im = postprocess(im, convert_opt, crop_opt)
585         else:
```

```
            # Create the circle mask for fancy output:
            if (circle == True):
                siz = im.shape[1]
                if siz % 2:
                    rang = arange(-siz / 2 + 1, siz / 2 + 1)
                else:
                    rang = arange(-siz / 2,siz / 2)
                x,y = meshgrid(rang,rang)
                z = x ** 2 + y ** 2
                a = (z < (siz / 2 - abs(offset) ) ** 2)
                im = im * a

        # Write down reconstructed slice:
        t2 = time()
        fname = outpath + outprefix + '_' + str(i).zfill(4) + '.tif'
        imsave(fname, im)
        t3 = time()

        # Write log (atomic procedure - lock used):
        write_log(lock, fname, logfilename, t2 - t1, (t3 - t2) + (t1 - t0) )


def main(argv):
    """To do...

    Usage
    -----


    Parameters
    ---------

    Example
    --------------------------
    The following line processes the first ten TIFF files of input path
    "/home/in" and saves the processed files to "/home/out" with the
    application of the Boin and Haibel filter with smoothing via a Butterworth
    filter of order 4 and cutoff frequency 0.01:

    reconstruct 0 4 C:\Temp\Dullin_Aug_2012\sino_noflat C:\Temp\Dullin_Aug_2012\sino_
↪noflat\output
    9.0 10.0 0.0 0.0 0.0 true sino slice C:\Temp\Dullin_Aug_2012\sino_noflat\tomo_
↪conv flat dark

    """
    lock = Lock()
    skip_flat = False
    skip_flat_after = True

    # Get the from and to number of files to process:
    int_from = int(argv[0])
    int_to = int(argv[1])

    # Get paths:
    infile = argv[2]
    outpath = argv[3]

    # Essential reconstruction parameters:
```

```
    angles = float(argv[4])
    offset = float(argv[5])
    param1 = argv[6]
    scale  = int(float(argv[7]))

    overpad = True if argv[8] == "True" else False
    logtrsf = True if argv[9] == "True" else False
    circle = True if argv[10] == "True" else False

    outprefix = argv[11]

    # Parameters for on-the-fly pre-processing:
    preprocessing_required = True if argv[12] == "True" else False
    flat_end = True if argv[13] == "True" else False
    half_half = True if argv[14] == "True" else False

    half_half_line = int(argv[15])

    ext_fov = True if argv[16] == "True" else False

    norm_sx = int(argv[19])
    norm_dx = int(argv[20])

    ext_fov_rot_right = argv[17]
    if ext_fov_rot_right == "True":
        ext_fov_rot_right = True
        if (ext_fov):
            norm_sx = 0
    else:
        ext_fov_rot_right = False
        if (ext_fov):
            norm_dx = 0

    ext_fov_overlap = int(argv[18])

    skip_ringrem = True if argv[21] == "True" else False
    ringrem = argv[22]

    # Extra reconstruction parameters:
    zerone_mode = True if argv[23] == "True" else False
    corr_offset = float(argv[24])

    reconmethod = argv[25]

    decim_factor = int(argv[26])
    downsc_factor = int(argv[27])

    # Parameters for postprocessing:
    postprocess_required = True if argv[28] == "True" else False
    convert_opt = argv[29]
    crop_opt = argv[30]

    angles_projfrom = int(argv[31])
    angles_projto = int(argv[32])

    rolling = True if argv[33] == "True" else False
    roll_shift = int(argv[34])

```

```
    dynamic_ff  = True if argv[35] == "True" else False

    nr_threads = int(argv[36])
    logfilename = argv[37]
    process_id = int(logfilename[-6:-4])

    # Check prefixes and path:
    #if not infile.endswith(sep): infile += sep
    if not exists(outpath):
        makedirs(outpath)

    if not outpath.endswith(sep): outpath += sep

    # Open the HDF5 file:
    f_in = getHDF5(infile, 'r')
    if "/tomo" in f_in:
        dset = f_in['tomo']

        tomoprefix = 'tomo'
        flatprefix = 'flat'
        darkprefix = 'dark'
    else:
        dset = f_in['exchange/data']
        if "/provenance/detector_output" in f_in:
            prov_dset = f_in['provenance/detector_output']

            tomoprefix = prov_dset.attrs['tomo_prefix']
            flatprefix = prov_dset.attrs['flat_prefix']
            darkprefix = prov_dset.attrs['dark_prefix']

    dset_min = -1
    dset_max = -1
    if (zerone_mode):
        if ('min' in dset.attrs):
            dset_min = float(dset.attrs['min'])
        else:
            zerone_mode = False

        if ('max' in dset.attrs):
            dset_max = float(dset.attrs['max'])
        else:
            zerone_mode = False

    num_sinos = tdf.get_nr_sinos(dset) # Pay attention to the downscale factor

    if (num_sinos == 0):
        log = open(logfilename,"a")
        log.write(linesep + "\tNo projections found. Process will end.")
        log.close()
        exit()

    # Check extrema (int_to == -1 means all files):
    if ((int_to >= num_sinos) or (int_to == -1)):
        int_to = num_sinos - 1

    # Log info:
    log = open(logfilename,"w")
    log.write(linesep + "\tInput file: %s" % (infile))
```

```
    log.write(linesep + "\tOutput path: %s" % (outpath))
    log.write(linesep + "\t---------------")
    log.write(linesep + "\tPreparing the work plan...")
    log.close()

    # Get correction plan and phase retrieval plan (if required):
    corrplan = -1
    phrtplan = -1

    skipflat = False

    im_dark = -1
    EFF = -1
    filtEFF = -1
    if (preprocessing_required):
        if not dynamic_ff:
            # Load flat fielding plan either from cache (if required) or from TDF file and cache it for faster re-use:
            corrplan = extract_flatdark(f_in, flat_end, logfilename)
            if (isscalar(corrplan['im_flat']) and isscalar(corrplan['im_flat_after'])):
                skipflat = True

            # Dowscale flat and dark images if necessary:
            if isinstance(corrplan['im_flat'], ndarray):
                corrplan['im_flat'] = corrplan['im_flat'][::downsc_factor,::downsc_factor]
            if isinstance(corrplan['im_dark'], ndarray):
                corrplan['im_dark'] = corrplan['im_dark'][::downsc_factor,::downsc_factor]
            if isinstance(corrplan['im_flat_after'], ndarray):
                corrplan['im_flat_after'] = corrplan['im_flat_after'][::downsc_factor,::downsc_factor]
            if isinstance(corrplan['im_dark_after'], ndarray):
                corrplan['im_dark_after'] = corrplan['im_dark_after'][::downsc_factor,::downsc_factor]

        else:
            # Dynamic flat fielding:
            if "/tomo" in f_in:
                if "/flat" in f_in:
                    flat_dset = f_in['flat']
                    if "/dark" in f_in:
                        im_dark = _medianize(f_in['dark'])
                    else:
                        skipdark = True
                else:
                    skipflat = True # Nothing to do in this case
            else:
                if "/exchange/data_white" in f_in:
                    flat_dset = f_in['/exchange/data_white']
                    if "/exchange/data_dark" in f_in:
                        im_dark = _medianize(f_in['/exchange/data_dark'])
                    else:
                        skipdark = True
                else:
                    skipflat = True # Nothing to do in this case

```

```
            # Prepare plan for dynamic flat fielding with 16 repetitions:
            if not skipflat:
                EFF, filtEFF = dff_prepare_plan(flat_dset, 16, im_dark)

                # Downscale images if necessary:
                im_dark = im_dark[::downsc_factor,::downsc_factor]
                EFF = EFF[::downsc_factor,::downsc_factor,:]
                filtEFF = filtEFF[::downsc_factor,::downsc_factor,:]

    f_in.close()

    # Log infos:
    log = open(logfilename,"a")
    log.write(linesep + "\tWork plan prepared correctly.")
    log.write(linesep + "\t--------------")
    log.write(linesep + "\tPerforming reconstruction...")
    log.close()

    # Run several threads for independent computation without waiting for threads␣
↪completion:
    for num in range(nr_threads):
        start = ( (int_to - int_from + 1) / nr_threads)*num + int_from
        if (num == nr_threads - 1):
            end = int_to
        else:
            end = ( (int_to - int_from + 1) / nr_threads)*(num + 1) + int_from - 1
        if (reconmethod == 'GRIDREC'):
            Process(target=process_gridrec, args=(lock, start, end, num_sinos, infile,
↪ outpath, preprocessing_required, skipflat,
                        corrplan, norm_sx, norm_dx, flat_end, half_half, half_half_
↪line, ext_fov, ext_fov_rot_right,
                        ext_fov_overlap, ringrem,
                        angles, angles_projfrom, angles_projto, offset, logtrsf,␣
↪param1, circle, scale, overpad,
                        rolling, roll_shift,
                        zerone_mode, dset_min, dset_max, decim_factor, downsc_factor,␣
↪corr_offset,
                        postprocess_required, convert_opt, crop_opt, dynamic_ff, EFF,␣
↪filtEFF, im_dark, outprefix,
                        logfilename )).start()
        else:
            Process(target=process, args=(lock, start, end, num_sinos, infile,␣
↪outpath, preprocessing_required, skipflat,
                        corrplan, norm_sx,
                        norm_dx, flat_end, half_half, half_half_line, ext_fov, ext_
↪fov_rot_right, ext_fov_overlap, ringrem,
                        angles, angles_projfrom, angles_projto, offset, logtrsf,␣
↪param1, circle, scale, overpad,
                        reconmethod, rolling, roll_shift,
                        zerone_mode, dset_min, dset_max, decim_factor, downsc_factor,␣
↪corr_offset,
                        postprocess_required, convert_opt, crop_opt, dynamic_ff, EFF,␣
↪filtEFF, im_dark, outprefix,
                        logfilename )).start()

    #start = int_from
    #end = int_to
    #if (reconmethod == 'GRIDREC'):
```

```
    #   process_gridrec(lock, start, end, num_sinos, infile, outpath, preprocessing_required, skipflat, corrplan, norm_sx,
    #                   norm_dx, flat_end, half_half, half_half_line, ext_fov, ext_fov_rot_right, ext_fov_overlap, ringrem,
    #                   angles, angles_projfrom, angles_projto, offset, logtrsf, param1, circle, scale, overpad,
    #                   rolling, roll_shift,
    #                   zerone_mode, dset_min, dset_max, decim_factor, downsc_factor, corr_offset,
    #                   postprocess_required, convert_opt, crop_opt, dynamic_ff, EFF, filtEFF, im_dark, outprefix, logfilename)
    #else:
    #   process(lock, start, end, num_sinos, infile, outpath, preprocessing_required, skipflat, corrplan, norm_sx,
    #                   norm_dx, flat_end, half_half, half_half_line, ext_fov, ext_fov_rot_right, ext_fov_overlap, ringrem,
    #                   angles, angles_projfrom, angles_projto, offset, logtrsf, param1, circle, scale, overpad,
    #                   reconmethod, rolling, roll_shift, zerone_mode, dset_min, dset_max, decim_factor, downsc_factor, corr_offset,
    #                   postprocess_required, convert_opt, crop_opt, dynamic_ff, EFF, filtEFF, im_dark, outprefix, logfilename)

    # Example:
    # 255 255 C:\Temp\BrunGeorgos.tdf C:\Temp\BrunGeorgos 3.1416 -31.0 shepp-logan 1.0 False False True slice True True True 5 False False 100 0 0 False rivers:11;0 False 0.0 FBP_CUDA 1 1 False - - 0 1799 False 2 C:\Temp\log_00.txt


if __name__ == "__main__":
    main(argv[1:])
```

## exec_postprocessing

This section contains the exec_postprocessing script.

Download file: `exec_postprocessing.py`

```
#########################################################################
# (C) 2016 Elettra - Sincrotrone Trieste S.C.p.A.. All rights reserved.   #
#                                                                         #
#                                                                         #
# This file is part of STP-Core, the Python core of SYRMEP Tomo Project,  #
# a software tool for the reconstruction of experimental CT datasets.     #
#                                                                         #
# STP-Core is free software: you can redistribute it and/or modify it     #
# under the terms of the GNU General Public License as published by the   #
# Free Software Foundation, either version 3 of the License, or (at your  #
# option) any later version.                                              #
#                                                                         #
# STP-Core is distributed in the hope that it will be useful, but WITHOUT #
# ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or   #
# FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License    #
# for more details.                                                       #
#                                                                         #
# You should have received a copy of the GNU General Public License       #
# along with STP-Core. If not, see <http://www.gnu.org/licenses/>.        #
```

```
#                                                                             #
###############################################################################

#
# Author: Francesco Brun
# Last modified: July, 8th 2016
#

from sys import argv, exit
from glob import glob
from os import linesep
from os.path import sep, basename, exists
from time import time
from multiprocessing import Process, Lock

# pystp-specific:
from stp_core.postprocess.postprocess import postprocess

from tifffile import imread, imsave

def _write_log(lock, fname, logfilename, cputime, iotime):

    lock.acquire()
    try:
        # Print out execution time:
        log = open(logfilename,"a")
        log.write(linesep + "\t%s processed (CPU: %0.3f sec - I/O: %0.3f sec)." %
↪(basename(fname), cputime, iotime))
        log.close()

    finally:
        lock.release()

def _process(lock, int_from, int_to, files, outpath, convert_opt, crop_opt, outprefix,
↪ logfilename):

    # Process the required subset of images:
    for i in range(int_from, int_to + 1):

        # Read i-th slice:
        t0 = time()
        im = imread(files[i])
        t1 = time()

        # Post process the image:
        im = postprocess(im, convert_opt, crop_opt)

        # Write down post-processed slice:
        t2 = time()
        fname = outpath + outprefix + '_' + str(i).zfill(4) + '.tif'
        imsave(fname, im)
        t3 = time()

        # Write log (atomic procedure - lock used):
        _write_log(lock, fname, logfilename, t2 - t1, (t3 - t2) + (t1 - t0) )


def main(argv):
```

```
    """To do...

    Usage
    -----


    Parameters
    ---------

    Example
    --------------------------
    The following line processes the first ten TIFF files of input path
    "/home/in" and saves the processed files to "/home/out" with the
    application of the Boin and Haibel filter with smoothing via a Butterworth
    filter of order 4 and cutoff frequency 0.01:

    reconstruct 0 4 C:\Temp\Dullin_Aug_2012\sino_noflat C:\Temp\Dullin_Aug_2012\sino_
↪noflat\output
    9.0 10.0 0.0 0.0 0.0 true sino slice C:\Temp\Dullin_Aug_2012\sino_noflat\tomo_
↪conv flat dark

    """
    lock = Lock()
    # Get the from and to number of files to process:
    int_from = int(argv[0])
    int_to = int(argv[1])

    # Get input and output paths:
    inpath = argv[2]
    outpath = argv[3]

    if not inpath.endswith(sep): inpath += sep
    if not outpath.endswith(sep): outpath += sep

    # Get parameters:
    convert_opt = argv[4]
    crop_opt = argv[5]

    outprefix = argv[6]

    # Number of threads to use and logfile:
    nr_threads = int(argv[7])
    logfilename = argv[8]

    # Get the files in infile:
    log = open(logfilename,"w")
    log.write(linesep + "\tInput TIFF folder: %s" % (inpath))
    log.write(linesep + "\tOutput TIFF folder: %s" % (outpath))
    log.write(linesep + "\t---------------")
    if (int_to != -1):
        log.write(linesep + "\tThe subset [%d,%d] of the input files will be␣
↪considered." % (int_from, int_to))
    log.write(linesep + "\tCropping:")
    crop_opt_num = crop_opt.split(":")
    log.write(linesep + "\t\tTop: %s pixels" % (crop_opt_num[0]))
    log.write(linesep + "\t\tBottom: %s pixels" % (crop_opt_num[1]))
    log.write(linesep + "\t\tLeft: %s pixels" % (crop_opt_num[2]))
    log.write(linesep + "\t\tRight: %s pixels" % (crop_opt_num[3]))
```

```
    conv_method, conv_args = convert_opt.split(":", 1)
    if (conv_method == "linear8"):
        min, max = conv_args.split(";")
        log.write(linesep + "\tConversion to 8-bit by remapping range [%s,%s] to [0,
↪255]." % (min, max))
    elif (conv_method == "linear"):
        min, max = conv_args.split(";")
        log.write(linesep + "\tConversion to 16-bit by remapping range [%s,%s] to [0,
↪65535]." % (min, max))
    log.write(linesep + "\t---------------")
    log.write(linesep + "\tBrowsing input folder...")
    log.close()

    files = sorted(glob(inpath + '*.tif*'))
    num_files = len(files)

    if ((int_to >= num_files) or (int_to == -1)):
        int_to = num_files - 1

    # Log infos:
    log = open(logfilename,"a")
    log.write(linesep + "\tInput folder browsed correctly.")
    log.close()

    # Run several threads for independent computation without waiting for threads␣
↪completion:
    for num in range(nr_threads):
        start = ( (int_to - int_from + 1) / nr_threads)*num + int_from
        if (num == nr_threads - 1):
            end = int_to
        else:
            end = ( (int_to - int_from + 1) / nr_threads)*(num + 1) + int_from - 1
        Process(target=_process, args=(lock, start, end, files, outpath, convert_opt,␣
↪crop_opt, outprefix, logfilename )).start()

    #start = 0
    #end = num_files - 1
    #process(lock, start, end, files, outpath, convert_opt, crop_opt, outprefix,␣
↪logfilename )

    #0 -1 C:\Temp\BrunGeorgos C:\Temp\BrunGeorgos\slice_8 linear8:-0.01;0.01␣
↪10:10:10:20 slice C:\Temp\log_00_conv.txt


if __name__ == "__main__":
    main(argv[1:])
```

## exec_phaseretrieval

This section contains the exec_phaseretrieval script.

Download file: `exec_phaseretrieval.py`

```
##########################################################################
# (C) 2016 Elettra - Sincrotrone Trieste S.C.p.A.. All rights reserved.   #
#                                                                        #
#                                                                        #
```

```
# This file is part of STP-Core, the Python core of SYRMEP Tomo Project,  #
# a software tool for the reconstruction of experimental CT datasets.     #
#                                                                          #
# STP-Core is free software: you can redistribute it and/or modify it      #
# under the terms of the GNU General Public License as published by the    #
# Free Software Foundation, either version 3 of the License, or (at your   #
# option) any later version.                                               #
#                                                                          #
# STP-Core is distributed in the hope that it will be useful, but WITHOUT  #
# ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or    #
# FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License     #
# for more details.                                                        #
#                                                                          #
# You should have received a copy of the GNU General Public License        #
# along with STP-Core. If not, see <http://www.gnu.org/licenses/>.         #
#                                                                          #
############################################################################

#
# Author: Francesco Brun
# Last modified: July, 8th 2016
#

from sys import argv, exit
from os import remove, sep, linesep
from os.path import exists
from numpy import float32, double, amin, amax
from time import time
from multiprocessing import Process, Lock
from pyfftw.interfaces.cache import enable as pyfftw_cache_enable, disable as pyfftw_
↪cache_disable
from pyfftw.interfaces.cache import set_keepalive_time as pyfftw_set_keepalive_time

# pystp-specific:
from stp_core.phaseretrieval.tiehom import tiehom, tiehom_plan
from stp_core.phaseretrieval.phrt   import phrt, phrt_plan

from h5py import File as getHDF5

# pystp-specific:
import stp_core.io.tdf as tdf


def _write_data(lock, im, index, outfile, outshape, outtype, logfilename, cputime,␣
↪itime):

    lock.acquire()
    try:
        t0 = time()
        f_out = getHDF5( outfile, 'a' )
        f_out_dset = f_out.require_dataset('exchange/data', outshape, outtype,␣
↪chunks=tdf.get_dset_chunks(outshape[0]))
        tdf.write_tomo(f_out_dset,index,im.astype(float32))

        # Set minimum and maximum:
        if ( amin(im[:]) < float(f_out_dset.attrs['min']) ):
            f_out_dset.attrs['min'] = str(amin(im[:]))
        if ( amax(im[:]) > float(f_out_dset.attrs['max'])):
```

```
            f_out_dset.attrs['max'] = str(amax(im[:]))
        f_out.close()
        t1 = time()

        # Print out execution time:
        log = open(logfilename,"a")
        log.write(linesep + "\ttomo_%s processed (CPU: %0.3f sec - I/O: %0.3f sec)."
↪% (str(index).zfill(4), cputime, t1 - t0 + itime))
        log.close()

    finally:
        lock.release()


def _process(lock, int_from, int_to, infile, outfile, outshape, outtype, method, plan,
↪ logfilename):

    # Process the required subset of images:
    for i in range(int_from, int_to + 1):

        # Read input image:
        t0 = time()
        f_in = getHDF5(infile, 'r')
        if "/tomo" in f_in:
            dset = f_in['tomo']
        else:
            dset = f_in['exchange/data']
        im = tdf.read_tomo(dset,i).astype(float32)
        f_in.close()
        t1 = time()

        # Perform phase retrieval (first time also PyFFTW prepares a plan):
        if (method == 0):
            im = tiehom(im, plan).astype(float32)
        else:
            im = phrt(im, plan, method).astype(float32)
        t2 = time()

        # Save processed image to HDF5 file (atomic procedure - lock used):
        _write_data(lock, im, i, outfile, outshape, outtype, logfilename, t2 - t1, t1␣
↪- t0)


def main(argv):
    """To do...

    """
    lock = Lock()

    skip_flat = True
    first_done = False
    pyfftw_cache_disable()
    pyfftw_cache_enable()
    pyfftw_set_keepalive_time(1800)

    # Get the from and to number of files to process:
    int_from = int(argv[0])
    int_to = int(argv[1])
```

```
115
116     # Get full paths of input TDF and output TDF:
117     infile = argv[2]
118     outfile = argv[3]
119
120     # Get the phase retrieval parameters:
121     method = int(argv[4])
122     param1 = double(argv[5])   # e.g. regParam, or beta
123     param2 = double(argv[6])   # e.g. thresh or delta
124     energy = double(argv[7])
125     distance = double(argv[8])
126     pixsize = double(argv[9]) / 1000.0 # pixsixe from micron to mm:
127     pad = True if argv[10] == "True" else False
128
129     # Number of threads (actually processes) to use and logfile:
130     nr_threads = int(argv[11])
131     logfilename = argv[12]
132
133     # Log infos:
134     log = open(logfilename,"w")
135     log.write(linesep + "\tInput TDF file: %s" % (infile))
136     log.write(linesep + "\tOutput TDF file: %s" % (outfile))
137     log.write(linesep + "\t---------------")
138     if (method == 0):
139         log.write(linesep + "\tMethod: TIE-Hom (Paganin et al., 2002)")
140         log.write(linesep + "\t---------------")
141         log.write(linesep + "\tDelta/Beta: %0.1f" % ((param2/param1))   )
142     #else:
143     #   log.write(linesep + "\tMethod: Projected CTF (Moosmann et al., 2011)")
144     #   log.write(linesep + "\t---------------")
145     #   log.write(linesep + "\tDelta/Beta: %0.1f" % ((param2/param1))   )
146     log.write(linesep + "\tEnergy: %0.1f keV" % (energy))
147     log.write(linesep + "\tDistance: %0.1f mm" % (distance))
148     log.write(linesep + "\tPixel size: %0.3f micron" % (pixsize*1000))
149     log.write(linesep + "\t---------------")
150     log.write(linesep + "\tBrowsing input files...")
151     log.close()
152
153     # Remove a previous copy of output:
154     if exists(outfile):
155         remove(outfile)
156
157     # Open the HDF5 file:
158     f_in = getHDF5(infile, 'r')
159     if "/tomo" in f_in:
160         dset = f_in['tomo']
161     else:
162         dset = f_in['exchange/data']
163     num_proj = tdf.get_nr_projs(dset)
164     num_sinos = tdf.get_nr_sinos(dset)
165
166     if (num_proj == 0):
167         log = open(logfilename,"a")
168         log.write(linesep + "\tNo projections found. Process will end.")
169         log.close()
170         exit()
171
172     log = open(logfilename,"a")
```

```
    log.write(linesep + "\tInput files browsed correctly.")
    log.close()

    # Check extrema (int_to == -1 means all files):
    if ( (int_to >= num_proj) or (int_to == -1) ):
        int_to = num_proj - 1

    if ( (int_from < 0) ):
        int_from = 0

    # Prepare the plan:
    log = open(logfilename,"a")
    log.write(linesep + "\tPreparing the work plan...")
    log.close()

    im = tdf.read_tomo(dset,0).astype(float32)


    outshape = tdf.get_dset_shape(im.shape[1], im.shape[0], num_proj)
    f_out = getHDF5(outfile, 'w')
    f_out_dset = f_out.create_dataset('exchange/data', outshape, im.dtype)
    f_out_dset.attrs['min'] = str(amin(im[:]))
    f_out_dset.attrs['max'] = str(amax(im[:]))

    f_out_dset.attrs['version'] = '1.0'
    f_out_dset.attrs['axes'] = "y:theta:x"

    f_in.close()
    f_out.close()

    if (method == 0):
        # Paganin's:
        plan = tiehom_plan (im, param1, param2, energy, distance, pixsize, pad)
    else:
        plan = phrt_plan (im, energy, distance, pixsize, param2, param1,␣
↪method, pad)

    # Run several threads for independent computation without waiting for␣
↪threads completion:
    for num in range(nr_threads):
        start = (num_proj / nr_threads)*num
        if (num == nr_threads - 1):
            end = num_proj - 1
        else:
            end = (num_proj / nr_threads)*(num + 1) - 1
        Process(target=_process, args=(lock, start, end, infile, outfile,␣
↪outshape, im.dtype, method, plan, logfilename)).start()

    #start = 0
    #end = num_proj - 1
    #_process(lock, start, end, infile, outfile, outshape, im.dtype, method,␣
↪plan, logfilename)

    #255 256 C:\Temp\BrunGeorgos_corr.tdf C:\Temp\BrunGeorgos_corr_phrt.tdf␣
↪0 1.0 2000.0 22.0 300.0 2.2 False 1 C:\Temp\log_00.txt
```

```
    #255 256 C:\Temp\BrunGeorgos_corr.tdf C:\Temp\BrunGeorgos_corr_phrt.tdf 4 2.5 1.0 22.0 300.0 2.2 False 1 C:\Temp\log_00.txt

if __name__ == "__main__":
    main(argv[1:])
```

## exec_tdf2tiff

This section contains the exec_tdf2tiff script.

Download file: `exec_tdf2tiff.py`

```
################################################################################
# (C) 2016 Elettra - Sincrotrone Trieste S.C.p.A.. All rights reserved.        #
#                                                                              #
#                                                                              #
# This file is part of STP-Core, the Python core of SYRMEP Tomo Project,       #
# a software tool for the reconstruction of experimental CT datasets.          #
#                                                                              #
# STP-Core is free software: you can redistribute it and/or modify it          #
# under the terms of the GNU General Public License as published by the        #
# Free Software Foundation, either version 3 of the License, or (at your       #
# option) any later version.                                                   #
#                                                                              #
# STP-Core is distributed in the hope that it will be useful, but WITHOUT      #
# ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or        #
# FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License         #
# for more details.                                                            #
#                                                                              #
# You should have received a copy of the GNU General Public License            #
# along with STP-Core. If not, see <http://www.gnu.org/licenses/>.             #
#                                                                              #
################################################################################

#
# Author: Francesco Brun
# Last modified: July, 8th 2016
#

import datetime
import os
import os.path
import time

from sys import argv, exit
from numpy import float32, float64

from tifffile import imread, imsave
from h5py import File as getHDF5

# pystp-specific:
import stp_core.io.tdf as tdf

from multiprocessing import Process, Lock

def _write_log(lock, fname, logfilename, iotime):
```

```
    """To do...

    """
    lock.acquire()
    try:
        # Print out execution time:
        log = open(logfilename,"a")
        log.write(os.linesep + "\t%s converted in %0.3f sec." % (os.path.
↪basename(fname), iotime))
        log.close()

    finally:
        lock.release()

def _process(lock, int_from, int_to, infile, dset_str, TIFFFormat, projorder, outpath,
↪ outprefix, logfilename):
    """To do...

    """
    try:

        f = getHDF5( infile, 'r' )
        dset = f[dset_str]

        # Process the required subset of images:
        for i in range(int_from, int_to + 1):

            # Read input image:
            t0 = time.time()

            if projorder:
                im = tdf.read_tomo( dset, i )
            else:
                im = tdf.read_sino( dset, i )

            if ( TIFFFormat ):
                fname = outpath + outprefix + '_' + str(i).zfill(4) + '.tif'
            else:
                fname = outpath + outprefix + '_' + str(i).zfill(4) + '_' + str(im.
↪shape[1]) + \
                        'x' + str(im.shape[0]) + '_' + str(im.dtype)    + '.raw'

            # Cast type (if required but it should never occur):
            if ((im.dtype).type is float64):
                im = im.astype(float32, copy=False)

            if ( TIFFFormat ):
                imsave(fname, im)
            else:
                im.tofile(fname)

            t1 = time.time()

            # Print out execution time:
            _write_log(lock, fname, logfilename, t1 - t0)

        f.close()

```

```
    except Exception:

        pass


def main(argv):
    """
    Converts a TDF file (HDF5 Tomo Data Format) into a sequence of TIFF (uncompressed) files.

    Parameters
    ----------
    from : scalar, integer
        among all the projections (or sinogram) data, a subset of the volume can
        be specified, ranging from the parameter "from" to the parameter "to"
        (see next). In most cases, this parameter is 0.

    to : scalar, integer
        among all the projections (or sinogram) data, a subset of the volume can
        be specified, ranging from the parameter "from" (see previous parameter) to
        the parameter "to". If the value -1 is specified, all the projection (or sinogram)
        data will be considered.

    in_file : string
        path with filename of the TDF to read from (e.g. "Z:\\sample1.tdf").

    out_path : string
        path that will contain the sequence of TIFF files (e.g. "Z:\\sample1\\tomo\\"). WARNING:
        the program does NOT automatically create non-existing folders and subfolders specified
        in the path. Moreover, if files with the same name already exist they will be automatically
        overwritten.

    file_prefix : string
        string to be assumed as the filename prefix of the TIFF files to create for the projection (or
        sinogram) data. E.g. "tomo" will create files having name "tomo_0001.tif", "tomo_0002.tif".

    flat_prefix : string
        string to be assumed as the filename prefix of the TIFF files to create for the flat (white field)
        data. E.g. "flat" will create files having name "flat_1.tif", "flat_2.tif". If dark or flat data have
        to be skipped the string "-" can be specified.

    dark_prefix : string
        string to be assumed as the filename prefix of the TIFF files to create for the dark (dark field)
        data. E.g. "dark" will create files having name "dark_1.tif", "dark_2.tif". If dark or flat data have
        to be skipped the string "-" can be specified.

    projection_order : boolean string
        specify the string "True" to create TIFF files for projections (the most common case), "False"
```

```
147         for sinograms.
148
149     TIFF_format : boolean string
150         specify the string "True" to create TIFF files, "False" for RAW files.
151
152     nr_threads : int
153         number of multiple threads (actually processes) to consider to speed up the␣
↪whole conversion process.
154
155     log_file : string
156         path with filename of a log file (e.g. "R:\\log.txt") where info about the␣
↪conversion is reported.
157
158     Returns
159     -------
160     no return value
161
162     Example
163     -------
164     Example call to convert all the projections data to a sequence of tomo*.tif files:
165
166         python tdf2tiff.py 0 -1 "C:\Temp\wet12T4part2.tdf" "C:\Temp\tomo" tomo flat␣
↪dark True True 3 "C:\Temp\log.txt"
167
168     Requirements
169     -------
170     - Python 2.7 with the latest NumPy, SciPy, H5Py.
171     - TIFFFile from C. Gohlke
172     - tdf.py
173
174     Tests
175     -------
176     Tested with WinPython-64bit-2.7.6.3 (Windows) and Anaconda 2.1.0 (Linux 64-bit).
177
178     """
179
180     lock = Lock()
181
182     # To be used without flat fielding (just conversion):
183     first_done = False
184
185     # Get the from and to number of files to process:
186     int_from = int(argv[0])
187     int_to = int(argv[1]) # -1 means "all files"
188
189     # Get paths:
190     infile = argv[2]
191     outpath = argv[3]
192
193     fileprefix = argv[4]
194     flatprefix = argv[5]
195     darkprefix = argv[6]
196
197     if (flatprefix == "-"):
198         skipflat = True
199     else:
200         skipflat = False
201
```

```
    if (darkprefix == "-"):
        skipdark = True
    else:
        skipdark = False

    if (fileprefix == "-"):
        skiptomo = True
    else:
        skiptomo = False

    projorder = argv[7]
    if projorder == "True":
        projorder = True
    else:
        projorder = False

    TIFFFormat = argv[8]
    if TIFFFormat == "True":
        TIFFFormat = True
    else:
        TIFFFormat = False

    nr_threads  = int(argv[9])
    logfilename = argv[10]

    # Check prefixes and path:
    if not outpath.endswith(os.path.sep): outpath += os.path.sep

    # Init variables:
    num_files = 0
    num_flats = 0
    num_darks = 0

    # Get the files in infile:
    log = open(logfilename,"w")
    log.write(os.linesep + "\tInput TDF: %s" % (infile))
    if ( TIFFFormat ):
        log.write(os.linesep + "\tOutput path where TIFF files will be created: %s" %␣
↪(outpath))
    else:
        log.write(os.linesep + "\tOutput path where RAW files will be created: %s" %␣
↪(outpath))
    log.write(os.linesep + "\t---------------")
    log.write(os.linesep + "\tFile output prefix: %s"  % (fileprefix))
    log.write(os.linesep + "\tFlat images output prefix: %s" % (flatprefix))
    log.write(os.linesep + "\tDark images output prefix: %s" % (darkprefix))
    log.write(os.linesep + "\t---------------")

    if (not (skiptomo)):
        if (int_to != -1):
            log.write(os.linesep + "\tThe subset [%d,%d] of the data will be␣
↪considered." % (int_from, int_to))

        if (projorder):
            log.write(os.linesep + "\tProjection order assumed.")
        else:
            log.write(os.linesep + "\tSinogram order assumed.")

```

```
        log.write(os.linesep + "\t--------------")
    log.close()

    if not os.path.exists(infile):
        log = open(logfilename,"a")
        log.write(os.linesep + "\tError: input TDF file not found. Process will end.")
        log.close()
        exit()

    # Open the HDF5 file:
    f = getHDF5( infile, 'r' )

    oldTDF = False;

    try:
        dset = f['tomo']
        oldTDF = True

    except Exception:

        pass

    if not oldTDF:

        #try:
            dset = f['exchange/data']

        #except Exception:

        #   log = open(logfilename,"a")
        #   log.write(os.linesep + "\tError: invalid TDF format. Process will end.")
        #   log.close()
        #   exit()

    if projorder:
        num_files = tdf.get_nr_projs(dset)
    else:
        num_files = tdf.get_nr_sinos(dset)
    f.close()



    # Get attributes:
    try:
        f = getHDF5( infile, 'r' )
        if ('version' in f.attrs) and (f.attrs['version'] == 'TDF 1.0'):
            log = open(logfilename,"a")
            log.write(os.linesep + "\tTDF version 1.0 found.")
            log.write(os.linesep + "\t--------------")
            log.close()
        f.close()

    except:
        log = open(logfilename,"a")
        log.write(os.linesep + "\tWarning: TDF version unknown. Some features will␣
↪not be available.")
        log.write(os.linesep + "\t--------------")
        log.close()
```

```
314
315     # Check extrema (int_to == -1 means all files):
316     if ( (int_to >= num_files) or (int_to == -1) ):
317         int_to = num_files - 1
318
319
320
321     # Spawn the process for the conversion of flat images:
322     if not skipflat:
323
324         f = getHDF5( infile, 'r' )
325         if oldTDF:
326             dset_str = 'flat'
327         else:
328             dset_str = 'exchange/data_white'
329         num_flats = tdf.get_nr_projs(f[dset_str])
330         f.close()
331
332         if ( num_flats > 0):
333             Process(target=_process, args=(lock, 0, num_flats - 1, infile, dset_str,␣
↪TIFFFormat,
334                                            True, outpath, flatprefix, logfilename)).
↪start()
335             #_process(lock, 0, num_flats - 1, infile, dset_str, TIFFFormat, projorder,
↪ outpath, flatprefix, logfilename)
336
337     # Spawn the process for the conversion of dark images:
338     if not skipdark:
339
340         f = getHDF5( infile, 'r' )
341         if oldTDF:
342             dset_str = 'dark'
343         else:
344             dset_str = 'exchange/data_dark'
345         num_darks = tdf.get_nr_projs(f[dset_str])
346         f.close()
347
348         if ( num_darks > 0):
349             Process(target=_process, args=(lock, 0, num_darks - 1, infile, dset_str,␣
↪TIFFFormat,
350                                            True, outpath, darkprefix, logfilename)).
↪start()
351             #_process(lock, 0, num_darks - 1, infile, dset_str, TIFFFormat, projorder,
↪ outpath, darkprefix, logfilename)
352
353     # Spawn the processes for the conversion of projection or sinogram images:
354     if not skiptomo:
355
356         if oldTDF:
357             dset_str = 'tomo'
358         else:
359             dset_str = 'exchange/data'
360
361         # Start the process for the conversion of the projections (or sinograms) in a␣
↪multi-threaded way:
362         for num in range(nr_threads):
363             start = ( (int_to - int_from + 1) / nr_threads)*num + int_from
364             if (num == nr_threads - 1):
```

```
                end = int_to
            else:
                end = ( (int_to - int_from + 1) / nr_threads)*(num + 1) + int_from - 1

            Process(target=_process, args=(lock, start, end, infile, dset_str, TIFFFormat,
                                           projorder, outpath, fileprefix, logfilename)).start()

            #_process(lock, start, end, infile, dset_str, TIFFFormat, projorder, outpath, fileprefix, logfilename)


if __name__ == "__main__":
    main(argv[1:])
```

## exec_tiff2tdf

This section contains the exec_tiff2tdf script.

Download file: `exec_tiff2tdf.py`

```
##########################################################################
# (C) 2016 Elettra - Sincrotrone Trieste S.C.p.A.. All rights reserved.  #
#                                                                        #
#                                                                        #
# This file is part of STP-Core, the Python core of SYRMEP Tomo Project, #
# a software tool for the reconstruction of experimental CT datasets.    #
#                                                                        #
# STP-Core is free software: you can redistribute it and/or modify it    #
# under the terms of the GNU General Public License as published by the  #
# Free Software Foundation, either version 3 of the License, or (at your #
# option) any later version.                                             #
#                                                                        #
# STP-Core is distributed in the hope that it will be useful, but WITHOUT #
# ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or  #
# FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License   #
# for more details.                                                      #
#                                                                        #
# You should have received a copy of the GNU General Public License      #
# along with STP-Core. If not, see <http://www.gnu.org/licenses/>.       #
#                                                                        #
##########################################################################

#
# Author: Francesco Brun
# Last modified: July, 8th 2016
#

import datetime
import os
import os.path
import numpy
import time

from time import strftime
from sys import argv, exit
```

```
from glob import glob

from tifffile import imread, imsave
from h5py import File as getHDF5

# pystp-specific:
import stp_core.io.tdf as tdf
from multiprocessing import Process, Lock


def _write_data(lock, im, index, offset, abs_offset, imfilename, timestamp, projorder, tot_files,
                provenance_dt, outfile, dsetname, outshape, outtype, logfilename, itime):
    """To do...

    """
    lock.acquire()
    try:

        # Open the HDF5 file to be populated with projections (or sinograms):
        t0 = time.time()
        f_out = getHDF5( outfile, 'a' )
        f_out_dset = f_out.require_dataset(dsetname, outshape, outtype, chunks=tdf.get_dset_chunks(outshape[0]))

        # Write the projection file or sinogram file:
        if projorder:
            tdf.write_tomo(f_out_dset, index - abs_offset, im)
        else:
            tdf.write_sino(f_out_dset, index - abs_offset, im)

        # Set minimum and maximum:
        if ( numpy.amin(im[:]) < float(f_out_dset.attrs['min']) ):
            f_out_dset.attrs['min'] = str(numpy.amin(im[:]))
        if ( numpy.amax(im[:]) > float(f_out_dset.attrs['max'])):
            f_out_dset.attrs['max'] = str(numpy.amax(im[:]))

        # Save provenance metadata:
        provenance_dset =  f_out.require_dataset('provenance/detector_output', (tot_files,), dtype=provenance_dt)
        provenance_dset["filename", offset - abs_offset + index] = numpy.string_(os.path.basename(imfilename))
        provenance_dset["timestamp", offset - abs_offset + index] = numpy.string_(
            datetime.datetime.fromtimestamp(timestamp).strftime('%Y-%m-%d %H:%M:%S.%f')[:-3])

        # Close the HDF5:
        f_out.close()
        t1 = time.time()

        # Print out execution time:
        log = open(logfilename,"a")
        log.write(os.linesep + "\t%s processed (I: %0.3f sec - O: %0.3f sec)." % (os.path.basename(imfilename), itime, t1 - t0))
        log.close()

    finally:
```

```
        lock.release()

def _process(lock, int_from, int_to, offset, abs_offset, files, projorder, outfile,␣
↪dsetname, outshape, outtype,
             crop_top, crop_bottom, crop_left, crop_right, tot_files, provenance_dt,␣
↪logfilename):
    """To do...

    """
    # Process the required subset of images:
    for i in range(int_from, int_to + 1):

        # Read input image:
        t0 = time.time()
        im = imread(files[i])

        # Crop:
        im = im[crop_top:im.shape[0]-crop_bottom,crop_left:im.shape[1]-crop_right]

        # Get the timestamp:
        t = os.path.getmtime(files[i])
        t1 = time.time()

        # Save processed image to HDF5 file (atomic procedure - lock used):
        _write_data(lock, im, i, offset, abs_offset, files[i], t, projorder, tot_
↪files, provenance_dt,
                    outfile, dsetname, outshape, outtype, logfilename, t1 - t0)


def main(argv):
    """
    Converts a sequence of TIFF files into a TDF file (HDF5 Tomo Data Format).

    Parameters
    ----------
    from : scalar, integer
        among all the projections (or sinogram) files, a subset of files can be␣
↪specified,
        ranging from the parameter "from" to the parameter "to" (see next). In most
        cases, this parameter is 0.

    to : scalar, integer
        among all the projections (or sinogram) files, a subset of files can be␣
↪specified,
        ranging from the parameter "from" (see previous parameter) to the parameter
        "to". If the value -1 is specified, all the projection files will be␣
↪considered.

    in_path : string
        path containing the sequence of TIFF files to consider (e.g.
↪"Z:\\sample1\\tomo\\").

    out_file : string
        path with filename of the TDF to create (e.g. "Z:\\sample1.tdf"). WARNING:␣
↪the program
        does NOT automatically create non-existing folders and subfolders specified␣
↪in the path.
        Moreover, if a file with the same name already exists it will be␣
↪automatically deleted and
```

```
136         overwritten.
137
138     crop_top : scalar, integer
139         during the conversion, images can be cropped if required. This parameter␣
↪specifies the number
140         of pixels to crop from the top of the image. Leave 0 for no cropping.
141
142     crop_bottom : scalar, integer
143         during the conversion, images can be cropped if required. This parameter␣
↪specifies the number
144         of pixels to crop from the bottom of the image. Leave 0 for no cropping.
145
146     crop_left : scalar, integer
147         during the conversion, images can be cropped if required. This parameter␣
↪specifies the number
148         of pixels to crop from the left of the image. Leave 0 for no cropping.
149
150     crop_right : scalar, integer
151         during the conversion, images can be cropped if required. This parameter␣
↪specifies the number
152         of pixels to crop from the right of the image. Leave 0 for no cropping.
153
154     file_prefix : string
155         string to be assumed as the filename prefix of the TIFF files to consider for␣
↪the projection (or
156         sinogram) files.    E.g. "tomo" will consider files having name "tomo_0001.tif
↪", "tomo_0002.tif".
157
158     flat_prefix : string
159         string to be assumed as the filename prefix of the TIFF files to consider for␣
↪the flat (white field)
160         files. E.g. "flat" will consider files having name "flat_1.tif", "flat_2.tif".
↪ If dark or flat files have
161         to be skipped the string "-" can be specified.
162
163     dark_prefix : string
164         string to be assumed as the filename prefix of the TIFF files to consider for␣
↪the dark (dark field)
165         files. E.g. "dark" will consider files having name "dark_1.tif", "dark_2.tif".
↪ If dark or flat files have
166         to be skipped the string "-" can be specified.
167
168     projection_order : boolean string
169         specify the string "True" if the TIFF files represent projections (the most␣
↪common case), "False"
170         for sinograms.
171
172     privilege_sino : boolean string
173         specify the string "True" if the TDF will privilege a fast read/write of␣
↪sinograms (the most common
174         case), "False" for fast read/write of projections.
175
176     compression : scalar, integer
177         an integer value in the range of [1,9] to be used as GZIP compression factor␣
↪in the HDF5 file, where
178         1 is the minimum compression (and maximum speed) and 9 is the maximum (and␣
↪slow) compression.
179         The value 0 can be specified with the meaning of no compression.
```

```
180
181     nr_threads : int
182         number of multiple threads (actually processes) to consider to speed up the␣
    ↪whole conversion process.
183
184     log_file : string
185         path with filename of a log file (e.g. "R:\\log.txt") where info about the␣
    ↪conversion is reported.
186
187     Returns
188     -------
189     no return value
190
191     Example
192     -------
193     Example call to convert all the tomo*.tif* projections to a TDF with no cropping␣
    ↪and minimum compression:
194
195         python tiff2tdf.py 0 -1 "Z:\\rawdata\\c_1\\tomo\\" "Z:\\work\\c1_compr9.tdf"␣
    ↪0 0 0 0 tomo flat
196         dark True True 1 "S:\\conversion.txt"
197
198     Requirements
199     -------
200     - Python 2.7 with the latest NumPy, SciPy, H5Py.
201     - TIFFFile from C. Gohlke's website http://www.lfd.uci.edu/~gohlke/
202       (consider also to install TIFFFile.c for performances).
203     - tdf.py
204
205     Tests
206     -------
207     Tested with WinPython-64bit-2.7.6.3 (Windows) and Anaconda 2.1.0 (Linux 64-bit). ␣
    ↪
208     """
209
210     lock = Lock()
211
212     # Get the from and to number of files to process:
213     int_from = int(argv[0])
214     int_to   = int(argv[1]) # -1 means "all files"
215
216     # Get paths:
217     inpath  = argv[2]
218     outfile = argv[3]
219
220     crop_top    = int(argv[4])  # 0 for all means "no cropping"
221     crop_bottom = int(argv[5])
222     crop_left   = int(argv[6])
223     crop_right  = int(argv[7])
224
225     tomoprefix = argv[8]
226     flatprefix = argv[9]   # - means "do not consider flat or darks"
227     darkprefix = argv[10]  # - means "do not consider flat or darks"
228
229     if (flatprefix == "-") or (darkprefix == "-"):
230         skipflat = True
231     else:
232         skipflat = False
```

```
    projorder = True if argv[11] == "True" else False
    privilege_sino = True if argv[12] == "True" else False

    # Get compression factor:
    compr_opts = int(argv[13])
    compressionFlag = True;
    if (compr_opts <= 0):
        compressionFlag = False;
    elif (compr_opts > 9):
        compr_opts = 9

    nr_threads  = int(argv[14])
    logfilename = argv[15]

    # Check prefixes and path:
    if not inpath.endswith(os.path.sep): inpath += os.path.sep

    # Get the files in inpath:
    log = open(logfilename,"w")
    log.write(os.linesep + "\tInput path: %s" % (inpath))
    log.write(os.linesep + "\tOutput TDF file: %s" % (outfile))
    log.write(os.linesep + "\t---------------")
    log.write(os.linesep + "\tProjection file prefix: %s"  % (tomoprefix))
    log.write(os.linesep + "\tDark file prefix: %s" % (darkprefix))
    log.write(os.linesep + "\tFlat file prefix: %s" % (flatprefix))
    log.write(os.linesep + "\t---------------")
    log.write(os.linesep + "\tCropping:")
    log.write(os.linesep + "\t\tTop: %d pixels" % (crop_top))
    log.write(os.linesep + "\t\tBottom: %d pixels" % (crop_bottom))
    log.write(os.linesep + "\t\tLeft: %d pixels" % (crop_left))
    log.write(os.linesep + "\t\tRight: %d pixels" % (crop_right))
    if (int_to != -1):
        log.write(os.linesep + "\tThe subset [%d,%d] of the input files will be␣
↪considered." % (int_from, int_to))

    if (projorder):
        log.write(os.linesep + "\tProjection order assumed.")
    else:
        log.write(os.linesep + "\tSinogram order assumed.")

    if (privilege_sino):
        log.write(os.linesep + "\tFast I/O for sinograms privileged.")
    else:
        log.write(os.linesep + "\tFast I/O for projections privileged.")

    if (compressionFlag):
        log.write(os.linesep + "\tTDF compression factor: %d" % (compr_opts))
    else:
        log.write(os.linesep + "\tTDF compression: none.")

    if (skipflat):
        log.write(os.linesep + "\tWarning: flat/dark images (if any) will not be␣
↪considered.")
    log.write(os.linesep + "\t---------------")
    log.close()

    # Remove a previous copy of output:
```

```
    if os.path.exists(outfile):
        log = open(logfilename,"a")
        log.write(os.linesep + "\tWarning: an output file with the same name was overwritten.")
        os.remove(outfile)
        log.close()

    log = open(logfilename,"a")
    log.write(os.linesep + "\tBrowsing input files...")
    log.close()

    # Pythonic way to get file list:
    if os.path.exists(inpath):
        tomo_files = sorted(glob(inpath + tomoprefix + '*.tif*'))
        num_files = len(tomo_files)
    else:
        log = open(logfilename,"a")
        log.write(os.linesep + "\tError: input path does not exist. Process will end.")
        log.close()
        exit()

    if (num_files == 0):
        log = open(logfilename,"a")
        log.write(os.linesep + "\tError: no projection files found. Check input path and file prefixes.")
        log.close()
        exit()

    log = open(logfilename,"a")
    log.write(os.linesep + "\tInput files browsed correctly.")
    log.close()

    # Check extrema (int_to == -1 means all files):
    if ( (int_to >= num_files) or (int_to == -1) ):
        int_from = 0
        int_to   = num_files - 1

    # In case of subset specified:
    num_files = int_to - int_from + 1

    # Prepare output HDF5 output (should this be atomic?):
    im = imread(tomo_files[0])

    # Crop:
    im = im[crop_top:im.shape[0]-crop_bottom,crop_left:im.shape[1]-crop_right]

    log = open(logfilename,"a")
    log.write(os.linesep + "\tPreparing the work plan...")
    log.close()

    #dsetshape = (num_files,) + im.shape
    if projorder:
        #dsetshape = tdf.get_dset_shape(privilege_sino, im.shape[1], im.shape[0], num_files)
        datashape = tdf.get_dset_shape(im.shape[1], im.shape[0], num_files)
    else:
        #dsetshape = tdf.get_dset_shape(privilege_sino, im.shape[1], num_files, im.shape[0])
```

```
343         datashape = tdf.get_dset_shape(im.shape[1], num_files, im.shape[0])
344 
345     if not os.path.isfile(outfile):
346         f = getHDF5( outfile, 'w' )
347 
348         f.attrs['version'] = '1.0'
349         f.attrs['implements'] = "exchange:provenance"
350         echange_group  = f.create_group( 'exchange' )
351 
352         if (compressionFlag):
353             dset = f.create_dataset('exchange/data', datashape, im.dtype, chunks=tdf.
    ↪get_dset_chunks(im.shape[1]),
354                 compression="gzip", compression_opts=compr_opts, shuffle=True,␣
    ↪fletcher32=True)
355         else:
356             dset = f.create_dataset('exchange/data', datashape, im.dtype)
357 
358         if privilege_sino:
359             dset.attrs['axes'] = "y:theta:x"
360         else:
361             dset.attrs['axes'] = "theta:y:x"
362 
363         dset.attrs['min'] = str(numpy.amin(im[:]))
364         dset.attrs['max'] = str(numpy.amax(im[:]))
365 
366         # Get the total number of files to consider:
367         tot_files = num_files
368         if not skipflat:
369             num_flats = len(sorted(glob(inpath + flatprefix + '*.tif*')))
370             num_darks = len(sorted(glob(inpath + darkprefix + '*.tif*')))
371             tot_files = tot_files + num_flats + num_darks
372 
373         # Create provenance dataset:
374         provenance_dt   = numpy.dtype([("filename", numpy.dtype("S255")), ("timestamp
    ↪",  numpy.dtype("S255"))])
375         metadata_group  = f.create_group( 'provenance' )
376         provenance_dset = metadata_group.create_dataset('detector_output', (tot_files,
    ↪), dtype=provenance_dt)
377 
378         provenance_dset.attrs['tomo_prefix'] = tomoprefix;
379         provenance_dset.attrs['dark_prefix'] = darkprefix;
380         provenance_dset.attrs['flat_prefix'] = flatprefix;
381         provenance_dset.attrs['first_index'] = int(tomo_files[0][-8:-4]);
382 
383         # Handle the metadata:
384         if (os.path.isfile(inpath + 'logfile.xml')):
385             with open (inpath + 'logfile.xml', "r") as file:
386                 xml_command = file.read()
387             tdf.parse_metadata(f, xml_command)
388 
389         f.close()
390 
391     # Print out about plan preparation:
392     log = open(logfilename,"a")
393     log.write(os.linesep + "\tWork plan prepared succesfully.")
394     log.close()
395 
396     # Get the files in inpath:
```

```
    if not skipflat:

        #
        # Flat part
        #
        flat_files = sorted(glob(inpath + flatprefix + '*.tif*'))
        num_flats = len(flat_files)

        if ( num_flats > 0):

            # Create acquisition group:
            im = imread(flat_files[0])
            im = im[crop_top:im.shape[0]-crop_bottom,crop_left:im.shape[1]-crop_right]

            #flatshape = tdf.get_dset_shape(privilege_sino, im.shape[1], im.shape[0], num_flats)
            flatshape = tdf.get_dset_shape(im.shape[1], im.shape[0], num_flats)
            f = getHDF5( outfile, 'a' )
            if (compressionFlag):
                dset = f.create_dataset('exchange/data_white', flatshape, im.dtype, chunks=tdf.get_dset_chunks(im.shape[1]),
                    compression="gzip", compression_opts=compr_opts, shuffle=True, fletcher32=True)
            else:
                dset = f.create_dataset('exchange/data_white', flatshape, im.dtype)

            dset.attrs['min'] = str(numpy.amin(im[:]))
            dset.attrs['max'] = str(numpy.amax(im[:]))

            if privilege_sino:
                dset.attrs['axes'] = "y:theta:x"
            else:
                dset.attrs['axes'] = "theta:y:x"
            f.close()

            #process(lock, 0, num_flats - 1, 0, flat_files, True, outfile, 'exchange/data_white', dsetshape, im.dtype,
            #   crop_top, crop_bottom, crop_left, crop_right, tot_files, provenance_dt, logfilename )

        else:
            log = open(logfilename,"a")
            log.write(os.linesep + "\tWarning: flat files not found.")
            log.close()

        #
        # Dark part
        #
        dark_files = sorted(glob(inpath + darkprefix + '*.tif*'))
        num_darks = len(dark_files)

        if ( num_darks > 0):
            im = imread(dark_files[0])
            im = im[crop_top:im.shape[0]-crop_bottom,crop_left:im.shape[1]-crop_right]

            #darkshape = tdf.get_dset_shape(privilege_sino, im.shape[1], im.shape[0], num_flats)
            darkshape = tdf.get_dset_shape(im.shape[1], im.shape[0], num_darks)
```

```
449             f = getHDF5( outfile, 'a' )
450             if (compressionFlag):
451                 dset = f.create_dataset('exchange/data_dark', darkshape, im.dtype,␣
↪chunks=tdf.get_dset_chunks(im.shape[1]),
452                       compression="gzip", compression_opts=compr_opts, shuffle=True,␣
↪fletcher32=True)
453             else:
454                 dset = f.create_dataset('exchange/data_dark', darkshape, im.dtype)
455 
456             dset.attrs['min'] = str(numpy.amin(im))
457             dset.attrs['max'] = str(numpy.amax(im))
458 
459             if privilege_sino:
460                 dset.attrs['axes'] = "y:theta:x"
461             else:
462                 dset.attrs['axes'] = "theta:y:x"
463             f.close()
464 
465             #process(lock, 0, num_darks - 1, num_flats, dark_files, True, outfile,
↪'exchange/data_dark', dsetshape, im.dtype,
466             #   crop_top, crop_bottom, crop_left, crop_right,  tot_files, provenance_
↪dt, logfilename )
467 
468         else:
469 
470             log = open(logfilename,"a")
471             log.write(os.linesep + "\tWarning: dark files not found.")
472             log.close()
473 
474     # Process the required subset of images:
475     if not skipflat:
476         flatdark_offset = num_flats + num_darks
477     else:
478         flatdark_offset = 0
479 
480     # Spawn the process for the conversion of flat images:
481     if ( num_flats > 0):
482         Process(target=_process, args=(lock, 0, num_flats - 1, 0, 0, flat_files, True,
↪ outfile, 'exchange/data_white',
483             flatshape, im.dtype, crop_top, crop_bottom, crop_left, crop_right, tot_
↪files, provenance_dt, logfilename )).start()
484 
485     # Spawn the process for the conversion of dark images:
486     if ( num_darks > 0):
487         Process(target=_process, args=(lock, 0, num_darks - 1, num_flats, 0, dark_
↪files, True, outfile, 'exchange/data_dark',
488             darkshape, im.dtype, crop_top, crop_bottom, crop_left, crop_right, tot_
↪files, provenance_dt, logfilename )).start()
489 
490     # Start the process for the conversion of the projections (or sinograms) in a␣
↪multi-threaded way:
491     for num in range(nr_threads):
492         start = ( (int_to - int_from + 1) / nr_threads)*num + int_from
493         if (num == nr_threads - 1):
494             end = int_to
495         else:
496             end = ( (int_to - int_from + 1) / nr_threads)*(num + 1) + int_from - 1
497 
```

```
        Process(target=_process, args=(lock, start, end, flatdark_offset, int_from,␣
↪tomo_files, projorder, outfile, 'exchange/data',
                datashape, im.dtype, crop_top, crop_bottom, crop_left, crop_right,␣
↪tot_files, provenance_dt, logfilename )).start()

        #process(lock, start, end, offset, tomo_files, projorder, outfile, 'exchange/
↪data',
        #       datashape, im.dtype, crop_top, crop_bottom, crop_left, crop_right,␣
↪tot_files, provenance_dt, logfilename )

if __name__ == "__main__":
    main(argv[1:])
```

## tools_autolimit

This section contains the tools_autolimit script.

Download file: `tools_autolimit.py`

```
################################################################################
# (C) 2016 Elettra - Sincrotrone Trieste S.C.p.A.. All rights reserved.        #
#                                                                              #
#                                                                              #
# This file is part of STP-Core, the Python core of SYRMEP Tomo Project,       #
# a software tool for the reconstruction of experimental CT datasets.          #
#                                                                              #
# STP-Core is free software: you can redistribute it and/or modify it          #
# under the terms of the GNU General Public License as published by the        #
# Free Software Foundation, either version 3 of the License, or (at your       #
# option) any later version.                                                   #
#                                                                              #
# STP-Core is distributed in the hope that it will be useful, but WITHOUT      #
# ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or        #
# FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License         #
# for more details.                                                            #
#                                                                              #
# You should have received a copy of the GNU General Public License            #
# along with STP-Core. If not, see <http://www.gnu.org/licenses/>.             #
#                                                                              #
################################################################################

#
# Author: Francesco Brun
# Last modified: July, 8th 2016
#

import os
import os.path
import numpy
import time

from glob import glob
from sys import argv, exit
from h5py import File as getHDF5
from numpy import float32
from tifffile import imread, imsave

```

```
# pystp-specific:
import stp_core.io.tdf as tdf

def main(argv):
    """Computes min/max limits to be used in image degradation to 8-bit or 16-bit.

    Parameters
    ----------
    argv[0] : string
        The absolute path of the input folder containing reconstructed TIFF files.

    argv[1] : string
        The absolute path of the output txt file with the proposed limits as string
↪"min:max".

    Example
    -------
    tools_autolimit "S:\\SampleA\\slices" "R:\\Temp\\autolimit.txt"

    """
    try:

        # Get input and output paths:
        inpath  = argv[0]
        outfile  = argv[1]  # The txt file with the proposed center

        if not inpath.endswith(os.path.sep): inpath += os.path.sep

        # Get the number of files in folder:
        files = sorted(glob(inpath + '*.tif*'))
        num_files = len(files)

        # Read the median slice from disk:
        im = imread(files[num_files/2])

        # Flat the image and sort it:
        im_flat = im.flatten()
        im_flat = numpy.sort(im_flat)

        # Return as minimum the value the skip 0.30% of "black" tail and 0.005% of
↪"white" tail:
        low_idx  = int(im_flat.shape[0] * 0.0030)
        high_idx = int(im_flat.shape[0] * 0.9995)

        min = im_flat[low_idx]
        max = im_flat[high_idx]

        # Print center to output file:
        text_file = open(outfile, "w")
        text_file.write( str(min) + ":" + str(max) )
        text_file.close()

    except:

        exit()

if __name__ == "__main__":
    main(argv[1:])
```

## tools_multiangle

This section contains the tools_multiangle script.

Download file: `tools_multiangle.py`

```
##########################################################################
# (C) 2016 Elettra - Sincrotrone Trieste S.C.p.A.. All rights reserved.   #
#                                                                        #
#                                                                        #
# This file is part of STP-Core, the Python core of SYRMEP Tomo Project, #
# a software tool for the reconstruction of experimental CT datasets.    #
#                                                                        #
# STP-Core is free software: you can redistribute it and/or modify it    #
# under the terms of the GNU General Public License as published by the  #
# Free Software Foundation, either version 3 of the License, or (at your #
# option) any later version.                                             #
#                                                                        #
# STP-Core is distributed in the hope that it will be useful, but WITHOUT #
# ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or  #
# FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License   #
# for more details.                                                      #
#                                                                        #
# You should have received a copy of the GNU General Public License      #
# along with STP-Core. If not, see <http://www.gnu.org/licenses/>.       #
#                                                                        #
##########################################################################

#
# Author: Francesco Brun
# Last modified: Sept, 28th 2016
#

# python:
from sys import argv, exit
from os import remove, sep, linesep, listdir, makedirs
from os.path import exists, dirname, basename, splitext
from numpy import array, finfo, copy, float32, double, amin, amax, tile, concatenate,␣
↪asarray
from numpy import empty, reshape, log as nplog, arange, squeeze, fromfile, ndarray,␣
↪where, meshgrid
from time import time
from multiprocessing import Process, Lock

# pystp-specific:
from stp_core.preprocess.extfov_correction import extfov_correction
from stp_core.preprocess.flat_fielding import flat_fielding
from stp_core.preprocess.ring_correction import ring_correction
from stp_core.preprocess.extract_flatdark import extract_flatdark

from stp_core.phaseretrieval.tiehom import tiehom, tiehom_plan
from stp_core.phaseretrieval.phrt   import phrt, phrt_plan

from stp_core.reconstruct.rec_astra import recon_astra_fbp, recon_astra_iterative
from stp_core.reconstruct.rec_fista_tv import recon_fista_tv
from stp_core.reconstruct.rec_mr_fbp import recon_mr_fbp
from stp_core.reconstruct.rec_gridrec import recon_gridrec

from stp_core.postprocess.postprocess import postprocess
```

```

from stp_core.utils.padding import upperPowerOfTwo, padImage, padSmoothWidth
from stp_core.utils.caching import cache2plan, plan2cache

from tifffile import imread, imsave
from h5py import File as getHDF5
import stp_core.io.tdf as tdf


def write_log(lock, fname, logfilename):
    """To do...

    """
    lock.acquire()
    try:
        # Print out execution time:
        log = open(logfilename,"a")
        log.write(linesep + "\t%s reconstructed." % basename(fname))
        log.close()

    finally:
        lock.release()

def reconstruct(im, angles, offset, logtransform, param1, circle, scale, pad, method,
                zerone_mode, dset_min, dset_max, corr_offset, postprocess_required,␣
↪convert_opt,
                crop_opt, start, end, outpath, sino_idx, downsc_factor, decim_factor,␣
↪logfilename, lock, slice_prefix):
    """Reconstruct a sinogram with FBP algorithm (from ASTRA toolbox).

    Parameters
    ----------
    im1 : array_like
        Sinogram image data as numpy array.
    center : float
        Offset of the center of rotation to use for the tomographic
        reconstruction with respect to the half of sinogram width
        (default=0, i.e. half width).
    logtransform : boolean
        Apply logarithmic transformation before reconstruction (default=True).
    filter : string
        Filter to apply before the application of the reconstruction algorithm.␣
↪Filter
        types are: ram-lak, shepp-logan, cosine, hamming, hann, tukey, lanczos,␣
↪triangular,
        gaussian, barlett-hann, blackman, nuttall, blackman-harris, blackman-nuttall,
        flat-top, kaiser, parzen.
    circle : boolean
        Create a circle in the reconstructed image and set to zero pixels outside the
        circle (default=False).

    """
    # Copy required due to multithreading:
    im_f = im

    # Decimate projections if required:
    #if decim_factor > 1:
    #    im = im[::decim_factor,:]
```

```
106
107     # Upscale projections (if required):
108     if (abs(scale - 1.0) > finfo(float32).eps):
109         siz_orig1 = im_f.shape[1]
110         im_f = imresize(im_f, (im_f.shape[0], int(round(scale * im_f.shape[1]))),␣
↪interp='bicubic', mode='F')
111         offset = int(offset * scale)
112
113     offset = int(round(offset))
114
115     # Apply transformation for changes in the center of rotation:
116     if (offset != 0):
117         if (offset >= 0):
118             im_f = im_f[:,:-offset]
119
120             tmp = im_f[:,0] # Get first column
121             tmp = tile(tmp, (offset,1)) # Replicate the first column the right number␣
↪of times
122             im_f = concatenate((tmp.T,im_f), axis=1) # Concatenate tmp before the␣
↪image
123
124         else:
125             im_f = im_f[:,abs(offset):]
126
127             tmp = im_f[:,im_f.shape[1] - 1] # Get last column
128             tmp = tile(tmp, (abs(offset),1)) # Replicate the last column the right␣
↪number of times
129             im_f = concatenate((im_f,tmp.T), axis=1) # Concatenate tmp after the image
130
131     # Downscale projections (without pixel averaging):
132     #if downsc_factor > 1:
133     #   im = im[:,::downsc_factor]
134
135     # Scale image to [0,1] range (if required):
136     if (zerone_mode):
137
138         #print dset_min
139         #print dset_max
140         #print numpy.amin(im_f[:])
141         #print numpy.amax(im_f[:])
142         #im_f = (im_f - dset_min) / (dset_max - dset_min)
143
144         # Cheating the whole process:
145         im_f = (im_f - numpy.amin(im_f[:])) / (numpy.amax(im_f[:]) - numpy.amin(im_
↪f[:]))
146
147     # Apply log transform:
148     if (logtransform == True):
149         im_f[im_f <= finfo(float32).eps] = finfo(float32).eps
150         im_f = -nplog(im_f + corr_offset)
151
152     # Replicate pad image to double the width:
153     if (pad):
154
155         dim_o = im_f.shape[1]
156         n_pad = im_f.shape[1] + im_f.shape[1] / 2
157         marg  = (n_pad - dim_o) / 2
158
```

```
159         # Pad image:
160         im_f = padSmoothWidth(im_f, n_pad)
161 
162     # Loop for all the required angles:
163     for i in range(int(round(start)), int(round(end)) + 1):
164 
165         # Save image for next step:
166         im = im_f
167 
168         # Apply projection removal (if required):
169         im_f = im_f[0:int(round(i/decim_factor)), :]
170 
171         # Perform the actual reconstruction:
172         if (method.startswith('FBP')):
173             im_f = recon_astra_fbp(im_f, angles, method, param1)
174         elif (method == 'MR-FBP_CUDA'):
175             im_f = recon_mr_fbp(im_f, angles)
176         elif (method == 'FISTA-TV_CUDA'):
177             im_f = recon_fista_tv(im_f, angles, param1, param1)
178         elif (method == 'MLEM'):
179             im_f = recon_tomopy_iterative(im_f, angles, method, param1)
180         elif (method == 'GRIDREC'):
181             [im_f, im_f] = recon_gridrec(im_f, im_f, angles, param1)
182         else:
183             im_f = recon_astra_iterative(im_f, angles, method, param1, zerone_mode)
184 
185         # Crop:
186         if (pad):
187             im_f = im_f[marg:dim_o + marg, marg:dim_o + marg]
188 
189         # Resize (if necessary):
190         if (abs(scale - 1.0) > finfo(float32).eps):
191             im_f = imresize(im_f, (siz_orig1, siz_orig1), interp='nearest', mode='F')
192 
193         # Apply post-processing (if required):
194         if postprocess_required:
195             im_f = postprocess(im_f, convert_opt, crop_opt)
196         else:
197             # Create the circle mask for fancy output:
198             if (circle == True):
199                 siz = im_f.shape[1]
200                 if siz % 2:
201                     rang = arange(-siz / 2 + 1, siz / 2 + 1)
202                 else:
203                     rang = arange(-siz / 2,siz / 2)
204                 x,y = meshgrid(rang,rang)
205                 z = x ** 2 + y ** 2
206                 a = (z < (siz / 2 - int(round(abs(offset)/downsc_factor)) ) ** 2)
207                 im_f = im_f * a
208 
209         # Write down reconstructed image (file name modified with metadata):
210         fname = outpath + slice_prefix + '_' + str(sino_idx).zfill(4) + '_off=' +␣
↪str(offset*downsc_factor).zfill(4) + '_proj=' + str(i).zfill(4) + '.tif'
211         imsave(fname, im_f)
212 
213         # Restore original image for next step:
214         im_f = im
215 
```

```
        # Write log (atomic procedure - lock used):
        write_log(lock, fname, logfilename )


def process(sino_idx, num_sinos, infile, outpath, preprocessing_required, corr_plan, norm_sx, norm_dx, flat_end, half_half,
            half_half_line, ext_fov, ext_fov_rot_right, ext_fov_overlap, ringrem, phaseretrieval_required, phrtmethod,
            phrt_param1, phrt_param2,
            energy, distance, pixsize, phrtpad, approx_win, angles, offset, logtransform, param1, circle, scale, pad, method,
            zerone_mode, dset_min, dset_max, decim_factor, downsc_factor, corr_offset, postprocess_required, convert_opt,
            crop_opt, nr_threads, angles_from, angles_to, logfilename, lock, slice_prefix):
    """To do...

    """
    slice_nr = sino_idx

    # Perform reconstruction (on-the-fly preprocessing and phase retrieval, if required):
    if (phaseretrieval_required):

        # In this case a bunch of sinograms is loaded into memory:

        #
        # Load the temporary data structure reading the input TDF file.
        # To know the right dimension the first sinogram is pre-processed.
        #

        # Open the TDF file and get the dataset:
        f_in = getHDF5(infile, 'r')
        if "/tomo" in f_in:
            dset = f_in['tomo']
        else:
            dset = f_in['exchange/data']

        # Downscaling and decimation factors considered when determining the approximation window:
        zrange = arange(sino_idx - approx_win*downsc_factor/2, sino_idx + approx_win*downsc_factor/2, downsc_factor)
        zrange = zrange[ (zrange >= 0) ]
        zrange = zrange[ (zrange < num_sinos) ]
        approx_win = zrange.shape[0]

        # Approximation window cannot be odd:
        if (approx_win % 2 == 1):
            approx_win = approx_win-1
            zrange     = zrange[0:approx_win]

        # Read one sinogram to get the proper dimensions:
        test_im = tdf.read_sino(dset, zrange[0]).astype(float32)
        test_im = test_im[::decim_factor, ::downsc_factor]

        # Perform the pre-processing of the first sinogram to get the right dimension:
        if (preprocessing_required):
            test_im = flat_fielding (test_im, zrange[0]/downsc_factor, corr_plan, flat_end, half_half,
```

```
                                        half_half_line/decim_factor, norm_sx, norm_
↪dx).astype(float32)
            test_im = extfov_correction (test_im, ext_fov, ext_fov_rot_right, ext_fov_
↪overlap/downsc_factor).astype(float32)
            test_im = ring_correction (test_im, ringrem, flat_end, corr_plan['skip_
↪flat_after'], half_half,
                                        half_half_line/decim_factor, ext_fov).
↪astype(float32)

        # Now we can allocate memory for the bunch of slices:
        tmp_im = empty((approx_win, test_im.shape[0], test_im.shape[1]),␣
↪dtype=float32)
        tmp_im[0,:,:] = test_im

        # Reading all the the sinos from TDF file and close:
        for ct in range(1, approx_win):

            test_im = tdf.read_sino(dset, zrange[ct]).astype(float32)
            test_im = test_im[::decim_factor, ::downsc_factor]

            # Perform the pre-processing for each sinogram of the bunch:
            if (preprocessing_required):
                test_im = flat_fielding (test_im, zrange[ct]/downsc_factor, corr_plan,
↪ flat_end, half_half,
                                        half_half_line/decim_factor, norm_sx,␣
↪norm_dx).astype(float32)
                test_im = extfov_correction (test_im, ext_fov, ext_fov_rot_right, ext_
↪fov_overlap/downsc_factor).astype(float32)
                test_im = ring_correction (test_im, ringrem, flat_end, corr_plan[
↪'skip_flat_after'], half_half,
                                        half_half_line/decim_factor, ext_fov).
↪astype(float32)

            tmp_im[ct,:,:] = test_im

        f_in.close()

        # Now everything has to refer to a downscaled dataset:
        sino_idx = ((zrange == sino_idx).nonzero())

        #
        # Perform phase retrieval:
        #

        # Prepare the plan:
        if (phrtmethod == 0):
            # Paganin's:
            phrtplan = tiehom_plan (tmp_im[:,0,:], phrt_param1, phrt_param2, energy,␣
↪distance, pixsize*downsc_factor, padding=phrtpad)
        else:
            phrtplan = phrt_plan (tmp_im[:,0,:], energy, distance, pixsize*downsc_
↪factor, phrt_param2, phrt_param1, phrtmethod, padding=phrtpad)

        # Process each projection (whose height depends on the size of the bunch):
        for ct in range(0, tmp_im.shape[1]):
            if (phrtmethod == 0):
                tmp_im[:,ct,:] = tiehom(tmp_im[:,ct,:], phrtplan).astype(float32)
↪
```

```
            else:
                tmp_im[:,ct,:] = phrt(tmp_im[:,ct,:], phrtplan, phrtmethod).
↪astype(float32)

        # Extract the requested sinogram:
        im = tmp_im[sino_idx[0],:,:].squeeze()

    else:

        # Read only one sinogram:
        f_in = getHDF5(infile, 'r')
        if "/tomo" in f_in:
            dset = f_in['tomo']
        else:
            dset = f_in['exchange/data']
        im = tdf.read_sino(dset,sino_idx).astype(float32)
        f_in.close()

        # Downscale and decimate the sinogram:
        im = im[::decim_factor,::downsc_factor]
        sino_idx = sino_idx/downsc_factor

        # Perform the preprocessing of the sinogram (if required):
        if (preprocessing_required):
            im = flat_fielding (im, sino_idx, corr_plan, flat_end, half_half, half_
↪half_line/decim_factor,
                                norm_sx, norm_dx).astype(float32)
            im = extfov_correction (im, ext_fov, ext_fov_rot_right, ext_fov_overlap)
            im = ring_correction (im, ringrem, flat_end, corr_plan['skip_flat_after'],
↪ half_half,
                                half_half_line/decim_factor, ext_fov)

    # Log infos:
    log = open(logfilename,"a")
    log.write(linesep + "\tPerforming reconstruction with multiple centers of␣
↪rotation...")
    log.write(linesep + "\t--------------")
    log.close()

    # Split the computation into multiple processes:
    for num in range(nr_threads):
        start = ( (angles_to - angles_from + 1) / nr_threads)*num + angles_from
        if (num == nr_threads - 1):
            end = angles_to
        else:
            end = ( (angles_to - angles_from + 1) / nr_threads)*(num + 1) + angles_
↪from - 1

        #Process(target=reconstruct, args=(im, angles, offset/downsc_factor,␣
↪logtransform, param1, circle, scale, pad, method,
        #               zerone_mode, dset_min, dset_max, corr_offset, postprocess_
↪required, convert_opt, crop_opt, start, end,
        #               outpath, slice_nr, downsc_factor, decim_factor, logfilename,␣
↪lock, slice_prefix)).start()


        # Actual reconstruction:
        reconstruct(im, angles, offset/downsc_factor, logtransform, param1, circle,␣
↪scale, pad, method,
```

```
                        zerone_mode, dset_min, dset_max, corr_offset, postprocess_
↪required, convert_opt, crop_opt,
                        start, end, outpath, slice_nr, downsc_factor, decim_factor,␣
↪logfilename, lock, slice_prefix)



def main(argv):
    """To do...


    """
    lock = Lock()
    skip_flat = False
    skip_flat_after = True

    # Get the from and to number of files to process:
    sino_idx = int(argv[0])

    # Get paths:
    infile  = argv[1]
    outpath = argv[2]

    # Essential reconstruction parameters::
    angles   = float(argv[3])
    offset   = float(argv[4])
    param1   = argv[5]
    scale    = int(float(argv[6]))

    overpad  = True if argv[7] == "True" else False
    logtrsf  = True if argv[8] == "True" else False
    circle   = True if argv[9] == "True" else False

    # Parameters for on-the-fly pre-processing:
    preprocessing_required = True if argv[10] == "True" else False
    flat_end = True if argv[11] == "True" else False
    half_half = True if argv[12] == "True" else False

    half_half_line = int(argv[13])

    ext_fov = True if argv[14] == "True" else False

    norm_sx = int(argv[17])
    norm_dx = int(argv[18])

    ext_fov_rot_right = argv[15]
    if ext_fov_rot_right == "True":
        ext_fov_rot_right = True
        if (ext_fov):
            norm_sx = 0
    else:
        ext_fov_rot_right = False
        if (ext_fov):
            norm_dx = 0

    ext_fov_overlap = int(argv[16])

```

```
    skip_ringrem = True if argv[19] == "True" else False
    ringrem = argv[20]

    # Extra reconstruction parameters:
    zerone_mode = True if argv[21] == "True" else False
    corr_offset = float(argv[22])

    reconmethod = argv[23]

    decim_factor = int(argv[24])
    downsc_factor = int(argv[25])

    # Parameters for postprocessing:
    postprocess_required = True if argv[26] == "True" else False
    convert_opt = argv[27]
    crop_opt = argv[28]

    # Parameters for on-the-fly phase retrieval:
    phaseretrieval_required = True if argv[29] == "True" else False
    phrtmethod = int(argv[30])
    phrt_param1 = double(argv[31])   # param1( e.g. regParam, or beta)
    phrt_param2 = double(argv[32])   # param2( e.g. thresh or delta)
    energy = double(argv[33])
    distance = double(argv[34])
    pixsize = double(argv[35]) / 1000.0 # pixsixe from micron to mm:
    phrtpad = True if argv[36] == "True" else False
    approx_win = int(argv[37])

    preprocessingplan_fromcache = True if argv[38] == "True" else False
    tmppath    = argv[39]
    if not tmppath.endswith(sep): tmppath += sep

    nr_threads = int(argv[40])
    angles_from = float(argv[41])
    angles_to   = float(argv[42])

    slice_prefix = argv[43]

    logfilename = argv[44]

    if not exists(outpath):
        makedirs(outpath)

    if not outpath.endswith(sep): outpath += sep


    # Log info:
    log = open(logfilename,"w")
    log.write(linesep + "\tInput dataset: %s" % (infile))
    log.write(linesep + "\tOutput path: %s" % (outpath))
    log.write(linesep + "\t--------------")
    log.write(linesep + "\tLoading flat and dark images...")
    log.close()

    # Open the HDF5 file:
    f_in = getHDF5(infile, 'r')
    if "/tomo" in f_in:
        dset = f_in['tomo']
```

```
475     else:
476         dset = f_in['exchange/data']
477         if "/provenance/detector_output" in f_in:
478             prov_dset = f_in['provenance/detector_output']
479 
480     dset_min = -1
481     dset_max = -1
482     if (zerone_mode):
483         if ('min' in dset.attrs):
484             dset_min = float(dset.attrs['min'])
485         else:
486             zerone_mode = False
487 
488         if ('max' in dset.attrs):
489             dset_max = float(dset.attrs['max'])
490         else:
491             zerone_mode = False
492 
493     num_sinos = tdf.get_nr_sinos(dset) # Pay attention to the downscale factor
494 
495     if (num_sinos == 0):
496         exit()
497 
498     # Check extrema:
499     if (sino_idx >= num_sinos):
500         sino_idx = num_sinos - 1
501 
502     # Get correction plan and phase retrieval plan (if required):
503     corrplan = 0
504     if (preprocessing_required):
505         # Load flat fielding plan either from cache (if required) or from TDF file␣
↪and cache it for faster re-use:
506         if (preprocessingplan_fromcache):
507             try:
508                 corrplan = cache2plan(infile, tmppath)
509             except Exception as e:
510                 #print "Error(s) when reading from cache"
511                 corrplan = extract_flatdark(f_in, flat_end, logfilename)
512                 plan2cache(corrplan, infile, tmppath)
513         else:
514             corrplan = extract_flatdark(f_in, flat_end, logfilename)
515             plan2cache(corrplan, infile, tmppath)
516 
517         # Dowscale flat and dark images if necessary:
518         if isinstance(corrplan['im_flat'], ndarray):
519             corrplan['im_flat'] = corrplan['im_flat'][::downsc_factor,::downsc_
↪factor]
520         if isinstance(corrplan['im_dark'], ndarray):
521             corrplan['im_dark'] = corrplan['im_dark'][::downsc_factor,::downsc_
↪factor]
522         if isinstance(corrplan['im_flat_after'], ndarray):
523             corrplan['im_flat_after'] = corrplan['im_flat_after'][::downsc_factor,
↪::downsc_factor]
524         if isinstance(corrplan['im_dark_after'], ndarray):
525             corrplan['im_dark_after'] = corrplan['im_dark_after'][::downsc_factor,
↪::downsc_factor]
526 
527     f_in.close()
```

```
528
529     # Log infos:
530     log = open(logfilename,"a")
531     log.write(linesep + "\tPerforming preprocessing...")
532     log.close()
533
534     # Run computation:
535     process( sino_idx, num_sinos, infile, outpath, preprocessing_required, corrplan,␣
    ↪norm_sx,
536              norm_dx, flat_end, half_half, half_half_line, ext_fov, ext_fov_rot_right,␣
    ↪ext_fov_overlap, ringrem,
537              phaseretrieval_required, phrtmethod, phrt_param1, phrt_param2, energy,␣
    ↪distance, pixsize, phrtpad,
538              approx_win, angles, offset, logtrsf, param1, circle, scale, overpad,␣
    ↪reconmethod, zerone_mode,
539              dset_min, dset_max, decim_factor, downsc_factor, corr_offset, postprocess_
    ↪required, convert_opt,
540              crop_opt, nr_threads, angles_from, angles_to, logfilename, lock, slice_
    ↪prefix )
541
542     # Sample:
543     # 26 C:\Temp\dataset42.tdf C:\Temp\test_angles 3.1416 -72.0 shepp-logan 1.0 False␣
    ↪True True True True False 5 False False 100 0 0 False rivers:11;0 False 0.0 FBP_
    ↪CUDA 1 4 False - - False 0 1.0 1000.0 22 150 2.2 True 16 False C:\Temp 1 1148 1248␣
    ↪slice C:\Temp\log_angles_00.txt
544
545
546
547 if __name__ == "__main__":
548     main(argv[1:])
```

## tools_extractdata

This section contains the tools_extractdata script.

Download file: `tools_extractdata.py`

```
1  ############################################################################
2  # (C) 2016 Elettra - Sincrotrone Trieste S.C.p.A.. All rights reserved.   #
3  #                                                                          #
4  #                                                                          #
5  # This file is part of STP-Core, the Python core of SYRMEP Tomo Project,  #
6  # a software tool for the reconstruction of experimental CT datasets.     #
7  #                                                                          #
8  # STP-Core is free software: you can redistribute it and/or modify it     #
9  # under the terms of the GNU General Public License as published by the   #
10 # Free Software Foundation, either version 3 of the License, or (at your  #
11 # option) any later version.                                               #
12 #                                                                          #
13 # STP-Core is distributed in the hope that it will be useful, but WITHOUT #
14 # ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or   #
15 # FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License    #
16 # for more details.                                                        #
17 #                                                                          #
18 # You should have received a copy of the GNU General Public License       #
19 # along with STP-Core. If not, see <http://www.gnu.org/licenses/>.        #
20 #                                                                          #
```

```
################################################################################

#
# Author: Francesco Brun
# Last modified: July, 8th 2016
#


import os
import os.path
import numpy
import time

from sys import argv, exit
from h5py import File as getHDF5
from numpy import float32

# pystp-specific:
import stp_core.io.tdf as tdf

def main(argv):
    """Extract a 2D image (projection or sinogram) from the input TDF file␣
↪(DataExchange HDF5) and
    creates a 32-bit RAW file to disk.

    Parameters
    ----------
    argv[0] : string
        The absolute path of the input TDF.

    argv[1] : int
        The relative position of the image within the dataset.

    argv[2] : string
        One of the following options: 'tomo', 'sino', 'flat', 'dark'.

    argv[3] : string
        The absolute path of the output 32-bit RAW image file. Filename will be␣
↪modified by adding
        image width, image height, minimum and maximum value of the input TDF dataset.

    Example
    -------
    tools_extractdata "S:\\dataset.tdf" 128 tomo "R:\\proj"

    """
    try:
        #
        # Get input parameters:
        #
        infile   = argv[0]
        index    = int(argv[1])
        imtype   = argv[2]
        outfile  = argv[3]

        #
        # Body
        #
```

```
77
78          # Check if file exists:
79          if not os.path.exists(infile):
80              #log = open(logfilename,"a")
81              #log.write(os.linesep + "\tError: input TDF file not found. Process will␣
    ↪end.")
82              #log.close()
83              exit()
84
85          # Open the HDF5 file:
86
87          f = getHDF5( infile, 'r' )
88          if (imtype == 'sino'):
89              if "/tomo" in f:
90                  dset = f['tomo']
91              else:
92                  dset = f['exchange/data']
93              im = tdf.read_sino( dset, index )
94          elif (imtype == 'dark'):
95              if "/dark" in f:
96                  dset = f['dark']
97              else:
98                  dset = f['exchange/data_dark']
99              im = tdf.read_tomo( dset, index )
100         elif (imtype == 'flat'):
101             if "/flat" in f:
102                 dset = f['flat']
103             else:
104                 dset = f['exchange/data_white']
105             im = tdf.read_tomo( dset, index )
106         else:
107             if "/tomo" in f:
108                 dset = f['tomo']
109             else:
110                 dset = f['exchange/data']
111             im = tdf.read_tomo( dset, index )
112
113
114         min = float(numpy.nanmin(im[:]))
115         max = float(numpy.nanmax(im[:]))
116
117         # Get global attributes (if any):
118         try:
119             if ('version' in f.attrs):
120                 if (f.attrs['version'] == '1.0'):
121                     min = float(dset_tomo.attrs['min'])
122                     max = float(dset_tomo.attrs['max'])
123         except:
124             pass
125
126         f.close()
127
128         # Cast type:
129         im = im.astype(float32)
130
131         # Modify file name:
132         outfile = outfile + '_' + str(im.shape[1]) + 'x' + str(im.shape[0]) + '_' +␣
    ↪str(min) + '$' + str(max)
```

```

        # Write RAW data to disk:
        im.tofile(outfile)

    except:

        exit()

if __name__ == "__main__":
    main(argv[1:])
```

## tools_guesscenter

This section contains the tools_guesscenter script.

Download file: `tools_guesscenter.py`

```
##########################################################################
# (C) 2016 Elettra - Sincrotrone Trieste S.C.p.A.. All rights reserved.  #
#                                                                        #
#                                                                        #
# This file is part of STP-Core, the Python core of SYRMEP Tomo Project, #
# a software tool for the reconstruction of experimental CT datasets.    #
#                                                                        #
# STP-Core is free software: you can redistribute it and/or modify it    #
# under the terms of the GNU General Public License as published by the  #
# Free Software Foundation, either version 3 of the License, or (at your #
# option) any later version.                                             #
#                                                                        #
# STP-Core is distributed in the hope that it will be useful, but WITHOUT#
# ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or  #
# FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License   #
# for more details.                                                      #
#                                                                        #
# You should have received a copy of the GNU General Public License      #
# along with STP-Core. If not, see <http://www.gnu.org/licenses/>.       #
#                                                                        #
##########################################################################

#
# Author: Francesco Brun
# Last modified: Sept, 28th 2016
#

from math import pi
from numpy import float32, double, finfo, ndarray
from scipy.misc import imresize #scipy 0.12
from os import sep, remove
from os.path import basename
from sys import argv
from h5py import File as getHDF5

from pyfftw.interfaces.cache import enable as pyfftw_cache_enable, disable as pyfftw_
↪cache_disable
from pyfftw.interfaces.cache import set_keepalive_time as pyfftw_set_keepalive_time

import time
```

```

# pystp-specific:
import stp_core.io.tdf as tdf
import stp_core.utils.findcenter as findcenter
from stp_core.utils.caching import cache2plan, plan2cache
from stp_core.preprocess.extract_flatdark import extract_flatdark
from stp_core.preprocess.flat_fielding import flat_fielding

from tifffile import imread, imsave # only for debug

def main(argv):
    """Try to guess the center of rotation of the input CT dataset.

    Parameters
    ----------
    infile  : array_like
        HDF5 input dataset

    outfile : string
        Full path where the identified center of rotation will be written as output

    scale   : int
        If sub-pixel precision is interesting, use e.g. 2.0 to get a center of␣
↪rotation
        of .5 value. Use 1.0 if sub-pixel precision is not required

    angles  : int
        Total number of angles of the input dataset

    proj_from : int
        Initial projections to consider for the assumed angles

    proj_to : int
        Final projections to consider for the assumed angles

    method : string
        (not implemented yet)

    tmppath : string
        Temporary path where look for cached flat/dark files

    """
    # Get path:
    infile  = argv[0]          # The HDF5 file on the
    outfile = argv[1]          # The txt file with the proposed center
    scale   = float(argv[2])
    angles  = float(argv[3])
    proj_from  = int(argv[4])
    proj_to  = int(argv[5])
    method  = argv[6]
    tmppath = argv[7]
    if not tmppath.endswith(sep): tmppath += sep

    pyfftw_cache_disable()
    pyfftw_cache_enable()
    pyfftw_set_keepalive_time(1800)

    # Create a silly temporary log:
```

```
    tmplog  = tmppath + basename(infile) + str(time.time())

    # Open the HDF5 file (take into account also older TDF versions):
    f_in = getHDF5( infile, 'r' )
    if "/tomo" in f_in:
        dset = f_in['tomo']
    else:
        dset = f_in['exchange/data']
    num_proj = tdf.get_nr_projs(dset)
    num_sinos = tdf.get_nr_sinos(dset)

    # Get flats and darks from cache or from file:
    try:
        corrplan = cache2plan(infile, tmppath)
    except Exception as e:
        #print "Error(s) when reading from cache"
        corrplan = extract_flatdark(f_in, True, tmplog)
        remove(tmplog)
        plan2cache(corrplan, infile, tmppath)

    # Get first and the 180 deg projections:
    im1 = tdf.read_tomo(dset,proj_from).astype(float32)

    idx = int(round( (proj_to - proj_from)/angles * pi)) + proj_from
    im2 = tdf.read_tomo(dset,idx).astype(float32)

    # Apply simple flat fielding (if applicable):
    if (isinstance(corrplan['im_flat_after'], ndarray) and isinstance(corrplan['im_
↪flat'], ndarray) and
        isinstance(corrplan['im_dark'], ndarray) and isinstance(corrplan['im_dark_
↪after'], ndarray)) :
        im1 = ((abs(im1 - corrplan['im_dark'])) / (abs(corrplan['im_flat'] - corrplan[
↪'im_dark'])
            + finfo(float32).eps)).astype(float32)
        im2 = ((abs(im2 - corrplan['im_dark_after'])) / (abs(corrplan['im_flat_after
↪'] - corrplan['im_dark_after'])
            + finfo(float32).eps)).astype(float32)

    # Scale projections (if required) to get subpixel estimation:
    if ( abs(scale - 1.0) > finfo(float32).eps ):
        im1 = imresize(im1, (int(round(scale*im1.shape[0])), int(round(scale*im1.
↪shape[1]))), interp='bicubic', mode='F');
        im2 = imresize(im2, (int(round(scale*im2.shape[0])), int(round(scale*im2.
↪shape[1]))), interp='bicubic', mode='F');

    # Find the center (flipping left-right im2):
    cen = findcenter.usecorrelation(im1, im2[ :,::-1])
    cen = cen / scale

    # Print center to output file:
    text_file = open(outfile, "w")
    text_file.write(str(int(cen)))
    text_file.close()

    # Close input HDF5:
    f_in.close()

if __name__ == "__main__":
```

```
149     main(argv[1:])
```

## tools_raw2tiff32

This section contains the tools_raw2tiff32 script.

Download file: `tools_raw2tiff32.py`

```
1  ##########################################################################
2  # (C) 2016 Elettra - Sincrotrone Trieste S.C.p.A.. All rights reserved.   #
3  #                                                                        #
4  #                                                                        #
5  # This file is part of STP-Core, the Python core of SYRMEP Tomo Project, #
6  # a software tool for the reconstruction of experimental CT datasets.    #
7  #                                                                        #
8  # STP-Core is free software: you can redistribute it and/or modify it    #
9  # under the terms of the GNU General Public License as published by the  #
10 # Free Software Foundation, either version 3 of the License, or (at your #
11 # option) any later version.                                             #
12 #                                                                        #
13 # STP-Core is distributed in the hope that it will be useful, but WITHOUT #
14 # ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or  #
15 # FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License   #
16 # for more details.                                                      #
17 #                                                                        #
18 # You should have received a copy of the GNU General Public License      #
19 # along with STP-Core. If not, see <http://www.gnu.org/licenses/>.       #
20 #                                                                        #
21 ##########################################################################
22
23 #
24 # Author: Francesco Brun
25 # Last modified: July, 8th 2016
26 #
27
28 import os
29 import os.path
30
31 from sys import argv, exit
32 from tifffile import *
33 from numpy import zeros, fromfile, float32
34
35 def main(argv):
36     """Convert an input 32-bit RAW image to TIFF format
37
38     Parameters
39     ----------
40     argv[0] : string
41         The absolute path of input 32-bit RAW image file.
42
43     argv[1] : string
44         The absolute path of output 32-bit TIFF image file.
45
46     argv[2] : int
47         Width of the input RAW image.
48
49     argv[3] : int
```

```
        Height of the input RAW image.

    Example
    -------
    tools_raw2tiff32 "R:\\slice.raw" "R:\\slice.tiff" 2048 2048

    """
    #
    # Get the parameters:
    #
    infile  = argv[0]
    outfile = argv[1]
    width   = int(argv[2])
    height  = int(argv[3])

    #
    # Body
    #

    # Check if file exists:
    if not os.path.exists(infile):
        exit()

    try:
        # Prepare RAW matrix:
        im = zeros((width,height), dtype=float32)

        # Read RAW file:
        im = fromfile(infile, float32).reshape((height,width))

        # Save TIFF 32:
        imsave(outfile, im)

    except:
        exit()

if __name__ == "__main__":
    main(argv[1:])
```

## tools_multioffset

This section contains the tools_multioffset script.

Download file: `tools_multioffset.py`

```
##########################################################################
# (C) 2016 Elettra - Sincrotrone Trieste S.C.p.A.. All rights reserved.   #
#                                                                        #
#                                                                        #
# This file is part of STP-Core, the Python core of SYRMEP Tomo Project, #
# a software tool for the reconstruction of experimental CT datasets.    #
#                                                                        #
# STP-Core is free software: you can redistribute it and/or modify it    #
# under the terms of the GNU General Public License as published by the  #
# Free Software Foundation, either version 3 of the License, or (at your #
# option) any later version.                                             #
#                                                                        #
```

```
# STP-Core is distributed in the hope that it will be useful, but WITHOUT #
# ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or   #
# FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License    #
# for more details.                                                       #
#                                                                         #
# You should have received a copy of the GNU General Public License       #
# along with STP-Core. If not, see <http://www.gnu.org/licenses/>.        #
#                                                                         #
###########################################################################

#
# Author: Francesco Brun
# Last modified: Sept, 28th 2016
#

# python:
from sys import argv, exit
from os import remove, sep, linesep, listdir, makedirs
from os.path import exists, dirname, basename, splitext
from numpy import array, finfo, copy, float32, double, amin, amax, tile, concatenate, asarray
from numpy import empty, reshape, log as nplog, arange, squeeze, fromfile, ndarray, where, meshgrid
from time import time
from multiprocessing import Process, Lock

# pystp-specific:
from stp_core.preprocess.extfov_correction import extfov_correction
from stp_core.preprocess.flat_fielding import flat_fielding
from stp_core.preprocess.ring_correction import ring_correction
from stp_core.preprocess.extract_flatdark import extract_flatdark

from stp_core.phaseretrieval.tiehom import tiehom, tiehom_plan
from stp_core.phaseretrieval.phrt   import phrt, phrt_plan

from stp_core.reconstruct.rec_astra import recon_astra_fbp, recon_astra_iterative
from stp_core.reconstruct.rec_fista_tv import recon_fista_tv
from stp_core.reconstruct.rec_mr_fbp import recon_mr_fbp
from stp_core.reconstruct.rec_gridrec import recon_gridrec

from stp_core.postprocess.postprocess import postprocess

from stp_core.utils.padding import upperPowerOfTwo, padImage, padSmoothWidth
from stp_core.utils.caching import cache2plan, plan2cache

from tifffile import imread, imsave
from h5py import File as getHDF5
import stp_core.io.tdf as tdf


def write_log(lock, fname, logfilename):
    """To do...

    """
    lock.acquire()
    try:
        # Print out execution time:
        log = open(logfilename,"a")
```

```
69          log.write(linesep + "\t%s reconstructed." % basename(fname))
70          log.close()
71
72      finally:
73          lock.release()
74
75  def reconstruct(im, angles, angles_projfrom, angles_projto, offset, logtransform, param1, circle, scale, pad, method,
76                  zerone_mode, dset_min, dset_max, corr_offset, postprocess_required, convert_opt,
77                  crop_opt, start, end, outpath, sino_idx, downsc_factor, logfilename, lock, slice_prefix):
78      """Reconstruct a sinogram with FBP algorithm (from ASTRA toolbox).
79
80      Parameters
81      ----------
82      im1 : array_like
83          Sinogram image data as numpy array.
84      center : float
85          Offset of the center of rotation to use for the tomographic
86          reconstruction with respect to the half of sinogram width
87          (default=0, i.e. half width).
88      logtransform : boolean
89          Apply logarithmic transformation before reconstruction (default=True).
90      filter : string
91          Filter to apply before the application of the reconstruction algorithm. Filter
92          types are: ram-lak, shepp-logan, cosine, hamming, hann, tukey, lanczos, triangular,
93          gaussian, barlett-hann, blackman, nuttall, blackman-harris, blackman-nuttall,
94          flat-top, kaiser, parzen.
95      circle : boolean
96          Create a circle in the reconstructed image and set to zero pixels outside the
97          circle (default=False).
98
99      """
100     # Copy required due to multithreading:
101     im_f = im
102
103     # Decimate projections if required:
104     #if decim_factor > 1:
105     #    im = im[::decim_factor,:]
106
107     # Upscale projections (if required):
108     if (abs(scale - 1.0) > finfo(float32).eps):
109         siz_orig1 = im_f.shape[1]
110         im_f = imresize(im_f, (im_f.shape[0], int(round(scale * im_f.shape[1]))), interp='bicubic', mode='F')
111         offset = int(offset * scale)
112
113     # Loop for all the required offsets for the center of rotation:
114     for i in range(int(round(start)), int(round(end)) + 1, downsc_factor):
115
116         offset = int(round(i/downsc_factor))
117
118         # Apply transformation for changes in the center of rotation:
119         if (offset != 0):
120             if (offset >= 0):
```

```
                im_f = im_f[:,:-offset]

                tmp = im_f[:,0] # Get first column
                tmp = tile(tmp, (offset,1)) # Replicate the first column the right number of times
                im_f = concatenate((tmp.T,im_f), axis=1) # Concatenate tmp before the image

            else:
                im_f = im_f[:,abs(offset):]

                tmp = im_f[:,im_f.shape[1] - 1] # Get last column
                tmp = tile(tmp, (abs(offset),1)) # Replicate the last column the right number of times
                im_f = concatenate((im_f,tmp.T), axis=1) # Concatenate tmp after the image

        # Downscale projections (without pixel averaging):
        #if downsc_factor > 1:
        #    im = im[:,::downsc_factor]

        # Scale image to [0,1] range (if required):
        if (zerone_mode):

            #print dset_min
            #print dset_max
            #print numpy.amin(im_f[:])
            #print numpy.amax(im_f[:])
            #im_f = (im_f - dset_min) / (dset_max - dset_min)

            # Cheating the whole process:
            im_f = (im_f - numpy.amin(im_f[:])) / (numpy.amax(im_f[:]) - numpy.amin(im_f[:]))

        # Apply log transform:
        if (logtransform == True):
            im_f[im_f <= finfo(float32).eps] = finfo(float32).eps
            im_f = -nplog(im_f + corr_offset)

        # Replicate pad image to double the width:
        if (pad):

            dim_o = im_f.shape[1]
            n_pad = im_f.shape[1] + im_f.shape[1] / 2
            marg  = (n_pad - dim_o) / 2

            # Pad image:
            im_f = padSmoothWidth(im_f, n_pad)

        # Perform the actual reconstruction:
        if (method.startswith('FBP')):
            im_f = recon_astra_fbp(im_f, angles, method, param1)
        elif (method == 'MR-FBP_CUDA'):
            im_f = recon_mr_fbp(im_f, angles)
        elif (method == 'FISTA-TV_CUDA'):
            im_f = recon_fista_tv(im_f, angles, param1, param1)
        elif (method == 'MLEM'):
            im_f = recon_tomopy_iterative(im_f, angles, method, param1)
```

```
        elif (method == 'GRIDREC'):
            [im_f, im_f] = recon_gridrec(im_f, im_f, angles, param1)
        else:
            im_f = recon_astra_iterative(im_f, angles, method, param1, zerone_mode)


        # Crop:
        if (pad):
            im_f = im_f[marg:dim_o + marg, marg:dim_o + marg]

        # Resize (if necessary):
        if (abs(scale - 1.0) > finfo(float32).eps):
            im_f = imresize(im_f, (siz_orig1, siz_orig1), interp='nearest', mode='F')

        # Apply post-processing (if required):
        if postprocess_required:
            im_f = postprocess(im_f, convert_opt, crop_opt)
        else:
            # Create the circle mask for fancy output:
            if (circle == True):
                siz = im_f.shape[1]
                if siz % 2:
                    rang = arange(-siz / 2 + 1, siz / 2 + 1)
                else:
                    rang = arange(-siz / 2,siz / 2)
                x,y = meshgrid(rang,rang)
                z = x ** 2 + y ** 2
                a = (z < (siz / 2 - int(round(abs(offset)/downsc_factor)) ) ** 2)
                im_f = im_f * a

        # Write down reconstructed image (file name modified with metadata):
        if ( i >= 0 ):
            fname = outpath + slice_prefix + '_' + str(sino_idx).zfill(4) + '_proj='␣
↪+ str(angles_projto - angles_projfrom) + '_col=' + str((im_f.shape[1] +␣
↪offset)*downsc_factor).zfill(4) + '_off=+' + str(abs(offset*downsc_factor)).
↪zfill(4) + '.tif'
        else:
            fname = outpath + slice_prefix + '_' + str(sino_idx).zfill(4) + '_proj='␣
↪+ str(angles_projto - angles_projfrom) + '_col=' + str((im_f.shape[1] +␣
↪offset)*downsc_factor).zfill(4) + '_off=-' + str(abs(offset*downsc_factor)).
↪zfill(4) + '.tif'

        imsave(fname, im_f)

        # Restore original image for next step:
        im_f = im

        # Write log (atomic procedure - lock used):
        write_log(lock, fname, logfilename )


def process(sino_idx, num_sinos, infile, outpath, preprocessing_required, corr_plan,␣
↪norm_sx, norm_dx, flat_end, half_half,
            half_half_line, ext_fov, ext_fov_rot_right, ext_fov_overlap, ringrem,␣
↪phaseretrieval_required, phrtmethod,
            phrt_param1, phrt_param2, energy, distance, pixsize, phrtpad, approx_win,␣
↪angles, angles_projfrom, angles_projto,
            offset, logtransform, param1, circle, scale, pad, method,
```

```
            zerone_mode, dset_min, dset_max, decim_factor, downsc_factor, corr_offset,
↪ postprocess_required, convert_opt,
            crop_opt, nr_threads, off_from, off_to, logfilename, lock, slice_prefix):
    """To do...

    """
    slice_nr = sino_idx

    # Perform reconstruction (on-the-fly preprocessing and phase retrieval, if␣
↪required):
    if (phaseretrieval_required):

        # In this case a bunch of sinograms is loaded into memory:

        #
        # Load the temporary data structure reading the input TDF file.
        # To know the right dimension the first sinogram is pre-processed.
        #

        # Open the TDF file and get the dataset:
        f_in = getHDF5(infile, 'r')
        if "/tomo" in f_in:
            dset = f_in['tomo']
        else:
            dset = f_in['exchange/data']

        # Downscaling and decimation factors considered when determining the␣
↪approximation window:
        zrange = arange(sino_idx - approx_win*downsc_factor/2, sino_idx + approx_
↪win*downsc_factor/2, downsc_factor)
        zrange = zrange[ (zrange >= 0) ]
        zrange = zrange[ (zrange < num_sinos) ]
        approx_win = zrange.shape[0]

        # Approximation window cannot be odd:
        if (approx_win % 2 == 1):
            approx_win = approx_win-1
            zrange     = zrange[0:approx_win]

        # Read one sinogram to get the proper dimensions:
        test_im = tdf.read_sino(dset, zrange[0]).astype(float32)
        test_im = test_im[::decim_factor, ::downsc_factor]

        # Perform the pre-processing of the first sinogram to get the right dimension:
        if (preprocessing_required):
            test_im = flat_fielding (test_im, zrange[0]/downsc_factor, corr_plan,␣
↪flat_end, half_half,
                                     half_half_line/decim_factor, norm_sx, norm_
↪dx).astype(float32)
            test_im = extfov_correction (test_im, ext_fov, ext_fov_rot_right, ext_fov_
↪overlap/downsc_factor).astype(float32)
            test_im = ring_correction (test_im, ringrem, flat_end, corr_plan['skip_
↪flat_after'], half_half,
                                       half_half_line/decim_factor, ext_fov).
↪astype(float32)

        # Now we can allocate memory for the bunch of slices:
        tmp_im = empty((approx_win, test_im.shape[0], test_im.shape[1]),␣
↪dtype=float32)
```

```
        tmp_im[0,:,:] = test_im

        # Reading all the the sinos from TDF file and close:
        for ct in range(1, approx_win):

            test_im = tdf.read_sino(dset, zrange[ct]).astype(float32)

            # Apply projection removal (if required):
            test_im = test_im[angles_projfrom:angles_projto, :]

            # Apply decimation and downscaling (if required):
            test_im = test_im[::decim_factor, ::downsc_factor]

            # Perform the pre-processing for each sinogram of the bunch:
            if (preprocessing_required):
                test_im = flat_fielding (test_im, zrange[ct]/downsc_factor, corr_plan, flat_end, half_half,
                                         half_half_line/decim_factor, norm_sx, norm_dx).astype(float32)
                test_im = extfov_correction (test_im, ext_fov, ext_fov_rot_right, ext_fov_overlap/downsc_factor).astype(float32)
                test_im = ring_correction (test_im, ringrem, flat_end, corr_plan['skip_flat_after'], half_half,
                                           half_half_line/decim_factor, ext_fov).astype(float32)

            tmp_im[ct,:,:] = test_im

        f_in.close()

        # Now everything has to refer to a downscaled dataset:
        sino_idx = ((zrange == sino_idx).nonzero())

        #
        # Perform phase retrieval:
        #

        # Prepare the plan:
        if (phrtmethod == 0):
            # Paganin's:
            phrtplan = tiehom_plan (tmp_im[:,0,:], phrt_param1, phrt_param2, energy, distance, pixsize*downsc_factor, padding=phrtpad)
        else:
            phrtplan = phrt_plan (tmp_im[:,0,:], energy, distance, pixsize*downsc_factor, phrt_param2, phrt_param1, phrtmethod, padding=phrtpad)

        # Process each projection (whose height depends on the size of the bunch):
        for ct in range(0, tmp_im.shape[1]):
            if (phrtmethod == 0):
                tmp_im[:,ct,:] = tiehom(tmp_im[:,ct,:], phrtplan).astype(float32)
            else:
                tmp_im[:,ct,:] = phrt(tmp_im[:,ct,:], phrtplan, phrtmethod).astype(float32)

        # Extract the requested sinogram:
        im = tmp_im[sino_idx[0],:,:].squeeze()

```

```
    else:

        # Read only one sinogram:
        f_in = getHDF5(infile, 'r')
        if "/tomo" in f_in:
            dset = f_in['tomo']
        else:
            dset = f_in['exchange/data']
        im = tdf.read_sino(dset,sino_idx).astype(float32)
        f_in.close()

        # Apply projection removal (if required):
        im = im[angles_projfrom:angles_projto, :]

        # Downscale and decimate the sinogram:
        im = im[::decim_factor,::downsc_factor]
        sino_idx = sino_idx/downsc_factor

        # Perform the preprocessing of the sinogram (if required):
        if (preprocessing_required):
            im = flat_fielding (im, sino_idx, corr_plan, flat_end, half_half, half_
↪half_line/decim_factor,
                                norm_sx, norm_dx).astype(float32)
            im = extfov_correction (im, ext_fov, ext_fov_rot_right, ext_fov_overlap)
            im = ring_correction (im, ringrem, flat_end, corr_plan['skip_flat_after'],
↪ half_half,
                                half_half_line/decim_factor, ext_fov)

    # Log infos:
    log = open(logfilename,"a")
    log.write(linesep + "\tPerforming reconstruction with multiple centers of␣
↪rotation...")
    log.write(linesep + "\t--------------")
    log.close()

    # Split the computation into multiple processes:
    for num in range(nr_threads):
        start = ( (off_to - off_from + 1) / nr_threads)*num + off_from
        if (num == nr_threads - 1):
            end = off_to
        else:
            end = ( (off_to - off_from + 1) / nr_threads)*(num + 1) + off_from - 1

        #Process(target=reconstruct, args=(im, angles, angles_projfrom, angles_projto,
↪ offset/downsc_factor,
        #               logtransform, param1, circle, scale, pad, method,
        #               zerone_mode, dset_min, dset_max, corr_offset, postprocess_
↪required, convert_opt,
        #               crop_opt, start, end, outpath, slice_nr, downsc_factor,␣
↪logfilename, lock, slice_prefix)).start()


        # Actual reconstruction:
        reconstruct(im, angles, angles_projfrom, angles_projto, offset/downsc_factor,␣
↪logtransform,
                        param1, circle, scale, pad, method,
                        zerone_mode, dset_min, dset_max, corr_offset, postprocess_
↪required, convert_opt,
```

```
                        crop_opt, start, end, outpath, slice_nr, downsc_factor,␣
↪logfilename, lock, slice_prefix)



def main(argv):
    """To do...



    """
    lock = Lock()
    skip_flat = False
    skip_flat_after = True

    # Get the from and to number of files to process:
    sino_idx = int(argv[0])

    # Get paths:
    infile  = argv[1]
    outpath = argv[2]

    # Essential reconstruction parameters::
    angles   = float(argv[3])
    off_step = float(argv[4])
    param1   = argv[5]
    scale    = int(float(argv[6]))

    overpad  = True if argv[7] == "True" else False
    logtrsf  = True if argv[8] == "True" else False
    circle   = True if argv[9] == "True" else False

    # Parameters for on-the-fly pre-processing:
    preprocessing_required = True if argv[10] == "True" else False
    flat_end = True if argv[11] == "True" else False
    half_half = True if argv[12] == "True" else False

    half_half_line = int(argv[13])

    ext_fov = True if argv[14] == "True" else False

    norm_sx = int(argv[17])
    norm_dx = int(argv[18])

    ext_fov_rot_right = argv[15]
    if ext_fov_rot_right == "True":
        ext_fov_rot_right = True
        if (ext_fov):
            norm_sx = 0
    else:
        ext_fov_rot_right = False
        if (ext_fov):
            norm_dx = 0

    ext_fov_overlap = int(argv[16])

    skip_ringrem = True if argv[19] == "True" else False
```

```
    ringrem = argv[20]

    # Extra reconstruction parameters:
    zerone_mode = True if argv[21] == "True" else False
    corr_offset = float(argv[22])

    reconmethod = argv[23]

    decim_factor = int(argv[24])
    downsc_factor = int(argv[25])

    # Parameters for postprocessing:
    postprocess_required = True if argv[26] == "True" else False
    convert_opt = argv[27]
    crop_opt = argv[28]

    # Parameters for on-the-fly phase retrieval:
    phaseretrieval_required = True if argv[29] == "True" else False
    phrtmethod = int(argv[30])
    phrt_param1 = double(argv[31])   # param1( e.g. regParam, or beta)
    phrt_param2 = double(argv[32])   # param2( e.g. thresh or delta)
    energy = double(argv[33])
    distance = double(argv[34])
    pixsize = double(argv[35]) / 1000.0 # pixsixe from micron to mm:
    phrtpad = True if argv[36] == "True" else False
    approx_win = int(argv[37])

    angles_projfrom = int(argv[38])
    angles_projto = int(argv[39])

    preprocessingplan_fromcache = True if argv[40] == "True" else False
    tmppath    = argv[41]
    if not tmppath.endswith(sep): tmppath += sep

    nr_threads = int(argv[42])
    off_from   = float(argv[43])
    off_to     = float(argv[44])

    slice_prefix = argv[45]

    logfilename = argv[46]

    if not exists(outpath):
        makedirs(outpath)

    if not outpath.endswith(sep): outpath += sep


    # Log info:
    log = open(logfilename,"w")
    log.write(linesep + "\tInput dataset: %s" % (infile))
    log.write(linesep + "\tOutput path: %s" % (outpath))
    log.write(linesep + "\t---------------")
    log.write(linesep + "\tLoading flat and dark images...")
    log.close()

    # Open the HDF5 file:
    f_in = getHDF5(infile, 'r')
```

```
    if "/tomo" in f_in:
        dset = f_in['tomo']
    else:
        dset = f_in['exchange/data']
        if "/provenance/detector_output" in f_in:
            prov_dset = f_in['provenance/detector_output']

    dset_min = -1
    dset_max = -1
    if (zerone_mode):
        if ('min' in dset.attrs):
            dset_min = float(dset.attrs['min'])
        else:
            zerone_mode = False

        if ('max' in dset.attrs):
            dset_max = float(dset.attrs['max'])
        else:
            zerone_mode = False

    num_sinos = tdf.get_nr_sinos(dset) # Pay attention to the downscale factor

    if (num_sinos == 0):
        exit()

    # Check extrema:
    if (sino_idx >= num_sinos):
        sino_idx = num_sinos - 1

    # Get correction plan and phase retrieval plan (if required):
    corrplan = 0
    if (preprocessing_required):
        # Load flat fielding plan either from cache (if required) or from TDF file␣
↪and cache it for faster re-use:
        if (preprocessingplan_fromcache):
            try:
                corrplan = cache2plan(infile, tmppath)
            except Exception as e:
                #print "Error(s) when reading from cache"
                corrplan = extract_flatdark(f_in, flat_end, logfilename)
                plan2cache(corrplan, infile, tmppath)
        else:
            corrplan = extract_flatdark(f_in, flat_end, logfilename)
            plan2cache(corrplan, infile, tmppath)

        # Dowscale flat and dark images if necessary:
        if isinstance(corrplan['im_flat'], ndarray):
            corrplan['im_flat'] = corrplan['im_flat'][::downsc_factor,::downsc_
↪factor]
        if isinstance(corrplan['im_dark'], ndarray):
            corrplan['im_dark'] = corrplan['im_dark'][::downsc_factor,::downsc_
↪factor]
        if isinstance(corrplan['im_flat_after'], ndarray):
            corrplan['im_flat_after'] = corrplan['im_flat_after'][::downsc_factor,
↪::downsc_factor]
        if isinstance(corrplan['im_dark_after'], ndarray):
            corrplan['im_dark_after'] = corrplan['im_dark_after'][::downsc_factor,
↪::downsc_factor]
```

```
    f_in.close()

    # Log infos:
    log = open(logfilename,"a")
    log.write(linesep + "\tPerforming preprocessing...")
    log.close()

    # Run computation:
    process( sino_idx, num_sinos, infile, outpath, preprocessing_required, corrplan, norm_sx,
             norm_dx, flat_end, half_half, half_half_line, ext_fov, ext_fov_rot_right, ext_fov_overlap, ringrem,
             phaseretrieval_required, phrtmethod, phrt_param1, phrt_param2, energy, distance, pixsize, phrtpad,
             approx_win, angles, angles_projfrom, angles_projto,
             off_step, logtrsf, param1, circle, scale, overpad, reconmethod, zerone_mode,
             dset_min, dset_max, decim_factor, downsc_factor, corr_offset, postprocess_required, convert_opt,
             crop_opt, nr_threads, off_from, off_to, logfilename, lock, slice_prefix )

    # Sample:
    # 26 C:\Temp\dataset42.tdf C:\Temp\test_offset 3.1416 1.0 shepp-logan 1.0 False True True True True False 5 False False 100 0 0 False rivers:11;0 False 0.0 FBP_CUDA 1 4 False - - False 0 1.0 1000.0 22 150 2.2 True 16 0 1163 False C:\Temp 1 -78 -70 slice C:\Temp\log_angles_00.txt



if __name__ == "__main__":
    main(argv[1:])
```

## tools_guessoverlap

This section contains the tools_guessoverlap script.

Download file: `tools_guessoverlap.py`

```
#########################################################################
# (C) 2016 Elettra - Sincrotrone Trieste S.C.p.A.. All rights reserved.   #
#                                                                         #
#                                                                         #
# This file is part of STP-Core, the Python core of SYRMEP Tomo Project,  #
# a software tool for the reconstruction of experimental CT datasets.     #
#                                                                         #
# STP-Core is free software: you can redistribute it and/or modify it     #
# under the terms of the GNU General Public License as published by the   #
# Free Software Foundation, either version 3 of the License, or (at your  #
# option) any later version.                                              #
#                                                                         #
# STP-Core is distributed in the hope that it will be useful, but WITHOUT #
# ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or   #
# FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License    #
# for more details.                                                       #
#                                                                         #
# You should have received a copy of the GNU General Public License       #
```

```
# along with STP-Core. If not, see <http://www.gnu.org/licenses/>.        #
#                                                                         #
###########################################################################

#
# Author: Francesco Brun
# Last modified: July, 8th 2016
#

from math import pi
from numpy import float32, double, finfo, ndarray
from scipy.misc import imresize #scipy 0.12
from os import sep, remove
from os.path import basename
from sys import argv
from h5py import File as getHDF5

import time

# pystp-specific:
import stp_core.io.tdf as tdf
import stp_core.utils.findcenter as findcenter
from stp_core.utils.caching import cache2plan, plan2cache
from stp_core.preprocess.extract_flatdark import extract_flatdark


def main(argv):
    """Try to guess the amount of overlap in the case of extended FOV CT.

    Parameters
    ----------
    infile  : array_like
        HDF5 input dataset

    outfile : string
        Full path where the identified overlap will be written as output

    scale   : int
        If sub-pixel precision is interesting, use e.g. 2.0 to get an overlap
        of .5 value. Use 1.0 if sub-pixel precision is not required

    tmppath : int
        Temporary path where look for cached flat/dark files

    """

    # Get path:
    infile  = argv[0]  # The HDF5 file on the SSD
    outfile  = argv[1]  # The txt file with the proposed center
    scale  = float(argv[2])
    tmppath = argv[3]
    if not tmppath.endswith(sep): tmppath += sep

    # Create a silly temporary log:
    tmplog  = tmppath + basename(infile) + str(time.time())


    # Open the HDF5 file:
```

```
    f_in = getHDF5( infile, 'r' )
    if "/tomo" in f_in:
        dset = f_in['tomo']
    else:
        dset = f_in['exchange/data']
    num_proj = tdf.get_nr_projs(dset)


    # Get first and 180 deg projections:
    im1 = tdf.read_tomo(dset,0).astype(float32)
    im2 = tdf.read_tomo(dset,num_proj/2).astype(float32)


    # Get flats and darks from cache or from file:
    try:
        corrplan = cache2plan(infile, tmppath)
    except Exception as e:
        #print "Error(s) when reading from cache"
        corrplan = extract_flatdark(f_in, True, tmplog)
        remove(tmplog)
        plan2cache(corrplan, infile, tmppath)

    # Apply simple flat fielding (if applicable):
    if (isinstance(corrplan['im_flat_after'], ndarray) and isinstance(corrplan['im_
↪flat'], ndarray) and
        isinstance(corrplan['im_dark'], ndarray) and isinstance(corrplan['im_dark_
↪after'], ndarray)) :
        im1 = ((abs(im1 - corrplan['im_dark'])) / (abs(corrplan['im_flat'] - corrplan[
↪'im_dark'])  + finfo(float32).eps)).astype(float32)
        im2 = ((abs(im2 - corrplan['im_dark_after'])) / (abs(corrplan['im_flat_after
↪'] - corrplan['im_dark_after'])  + finfo(float32).eps)).astype(float32)


    # Scale projections (if required) to get subpixel estimation:
    if ( abs(scale - 1.0) > finfo(float32).eps ):
        im1 = imresize(im1, (int(round(scale*im1.shape[0])), int(round(scale*im1.
↪shape[1]))), interp='bicubic', mode='F');
        im2 = imresize(im2, (int(round(scale*im2.shape[0])), int(round(scale*im2.
↪shape[1]))), interp='bicubic', mode='F');


    # Find the center (flipping left-right im2): DISTINGUISH BETWEEN AIR ON THE RIGHT␣
↪AND ON THE LEFT??????
    cen = findcenter.usecorrelation(im1, im2[ :,::-1])
    cen = (cen / scale)*2.0

    # Print center to output file:
    text_file = open(outfile, "w")
    text_file.write(str(int(abs(cen))))
    text_file.close()

    # Close input HDF5:
    f_in.close()

if __name__ == "__main__":
    main(argv[1:])
```

## preview_preprocessing

This section contains the preview_preprocessing script.

Download file: `preview_preprocessing.py`

```
##########################################################################
#  (C) 2016 Elettra - Sincrotrone Trieste S.C.p.A.. All rights reserved.  #
#                                                                        #
#                                                                        #
#  This file is part of STP-Core, the Python core of SYRMEP Tomo Project,#
#  a software tool for the reconstruction of experimental CT datasets.   #
#                                                                        #
#  STP-Core is free software: you can redistribute it and/or modify it   #
#  under the terms of the GNU General Public License as published by the #
#  Free Software Foundation, either version 3 of the License, or (at your#
#  option) any later version.                                            #
#                                                                        #
#  STP-Core is distributed in the hope that it will be useful, but WITHOUT#
#  ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or #
#  FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License  #
#  for more details.                                                     #
#                                                                        #
#  You should have received a copy of the GNU General Public License     #
#  along with STP-Core. If not, see <http://www.gnu.org/licenses/>.      #
#                                                                        #
##########################################################################

#
# Author: Francesco Brun
# Last modified: Sept, 28th 2016
#

from sys import argv, exit
from os import remove, sep,  linesep
from os.path import exists
from numpy import float32, nanmin, nanmax, isscalar
from time import time
from multiprocessing import Process, Lock

# pystp-specific:
from stp_core.preprocess.extfov_correction import extfov_correction
from stp_core.preprocess.flat_fielding import flat_fielding
from stp_core.preprocess.dynamic_flatfielding import dff_prepare_plan, dynamic_flat_
↪fielding
from stp_core.preprocess.ring_correction import ring_correction
from stp_core.preprocess.extract_flatdark import extract_flatdark, _medianize

from h5py import File as getHDF5
from stp_core.utils.caching import cache2plan, plan2cache
import stp_core.io.tdf as tdf

def main(argv):
    """To do...


    """
```

```
    # Get the zero-order index of the sinogram to pre-process:
    idx = int(argv[0])

    # Get paths:
    infile = argv[1]
    outfile = argv[2]

    # Normalization parameters:
    norm_sx = int(argv[3])
    norm_dx = int(argv[4])

    # Params for flat fielding with post flats/darks:
    flat_end = True if argv[5] == "True" else False
    half_half = True if argv[6] == "True" else False
    half_half_line = int(argv[7])

    # Params for extended FOV:
    ext_fov = True if argv[8] == "True" else False
    ext_fov_rot_right = argv[9]
    if ext_fov_rot_right == "True":
        ext_fov_rot_right = True
        if (ext_fov):
            norm_sx = 0
    else:
        ext_fov_rot_right = False
        if (ext_fov):
            norm_dx = 0
    ext_fov_overlap = int(argv[10])

    # Method and parameters coded into a string:
    ringrem = argv[11]

    # Flat fielding method (conventional or dynamic):
    dynamic_ff = True if argv[12] == "True" else False

    # Tmp path and log file:
    tmppath = argv[13]
    if not tmppath.endswith(sep): tmppath += sep
    logfilename = argv[14]


    # Open the HDF5 file:
    f_in = getHDF5(infile, 'r')

    try:
        if "/tomo" in f_in:
            dset = f_in['tomo']
        else:
            dset = f_in['exchange/data']

    except:
        log = open(logfilename,"a")
        log.write(linesep + "\tError reading input dataset. Process will end.")
        log.close()
        exit()

    num_proj = tdf.get_nr_projs(dset)
```

```
    num_sinos = tdf.get_nr_sinos(dset)

    # Check if the HDF5 makes sense:
    if (num_sinos == 0):
        log = open(logfilename,"a")
        log.write(linesep + "\tNo projections found. Process will end.")
        log.close()
        exit()

    # Get flat and darks from cache or from file:
    skipflat = False
    skipdark = False
    if not dynamic_ff:
        try:
            corrplan = cache2plan(infile, tmppath)
        except Exception as e:
            #print "Error(s) when reading from cache"
            corrplan = extract_flatdark(f_in, flat_end, logfilename)
            if (isscalar(corrplan['im_flat']) and isscalar(corrplan['im_flat_after'])
↪):
                skipflat = True
            else:
                plan2cache(corrplan, infile, tmppath)
    else:
        # Dynamic flat fielding:
        if "/tomo" in f_in:
            if "/flat" in f_in:
                flat_dset = f_in['flat']
                if "/dark" in f_in:
                    im_dark = _medianize(f_in['dark'])
                else:
                    skipdark = True
            else:
                skipflat = True # Nothing to do in this case
        else:
            if "/exchange/data_white" in f_in:
                flat_dset = f_in['/exchange/data_white']
                if "/exchange/data_dark" in f_in:
                    im_dark = _medianize(f_in['/exchange/data_dark'])
                else:
                    skipdark = True
            else:
                skipflat = True # Nothing to do in this case

        # Prepare plan for dynamic flat fielding with 16 repetitions:
        if not skipflat:
            EFF, filtEFF = dff_prepare_plan(flat_dset, 16, im_dark)

    # Read input image:
    im = tdf.read_sino(dset,idx).astype(float32)
    f_in.close()

    # Perform pre-processing (flat fielding, extended FOV, ring removal):
    if not skipflat:
        if dynamic_ff:
            # Dynamic flat fielding with downsampling = 2:
            im = dynamic_flat_fielding(im, idx, EFF, filtEFF, 2, im_dark, norm_sx,
↪norm_dx)
```

```
        else:
            im = flat_fielding(im, idx, corrplan, flat_end, half_half, half_half_line,
↪ norm_sx, norm_dx)

    im = extfov_correction(im, ext_fov, ext_fov_rot_right, ext_fov_overlap)
    if not skipflat and not dynamic_ff:
        im = ring_correction (im, ringrem, flat_end, corrplan['skip_flat_after'],␣
↪half_half, half_half_line, ext_fov)
    else:
        im = ring_correction (im, ringrem, False, False, half_half, half_half_line,␣
↪ext_fov)

    # Write down reconstructed preview file (file name modified with metadata):
    im = im.astype(float32)
    outfile = outfile + '_' + str(im.shape[1]) + 'x' + str(im.shape[0]) + '_' + str(␣
↪nanmin(im)) + '$' + str( nanmax(im) )
    im.tofile(outfile)

    # 255 C:\Temp\Pippo.tdf C:\Temp\pippo 0 0 True True 900 False False 0 rivers:3;0␣
↪False C:\Temp C:\Temp\log_00.txt


if __name__ == "__main__":
    main(argv[1:])
```

## preview_postprocessing

This section contains the preview_postprocessing script.

Download file: `preview_postprocessing.py`

```
#########################################################################
# (C) 2016 Elettra - Sincrotrone Trieste S.C.p.A.. All rights reserved.   #
#                                                                         #
#                                                                         #
# This file is part of STP-Core, the Python core of SYRMEP Tomo Project,  #
# a software tool for the reconstruction of experimental CT datasets.     #
#                                                                         #
# STP-Core is free software: you can redistribute it and/or modify it     #
# under the terms of the GNU General Public License as published by the   #
# Free Software Foundation, either version 3 of the License, or (at your  #
# option) any later version.                                              #
#                                                                         #
# STP-Core is distributed in the hope that it will be useful, but WITHOUT #
# ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or   #
# FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License    #
# for more details.                                                       #
#                                                                         #
# You should have received a copy of the GNU General Public License       #
# along with STP-Core. If not, see <http://www.gnu.org/licenses/>.        #
#                                                                         #
#########################################################################

#
# Author: Francesco Brun
# Last modified: July, 8th 2016
```

```
#

from sys import argv, exit
from glob import glob
from os import linesep
from os.path import sep, basename, exists
from time import time
from numpy import float32, nanmin, nanmax
from multiprocessing import Process, Lock

# pystp-specific:
from stp_core.postprocess.postprocess import postprocess

from tifffile import imread, imsave



def main(argv):
    """To do...

    Usage
    -----


    Parameters
    ---------

    Example
    --------------------------
    The following line processes the first ten TIFF files of input path
    "/home/in" and saves the processed files to "/home/out" with the
    application of the Boin and Haibel filter with smoothing via a Butterworth
    filter of order 4 and cutoff frequency 0.01:

    reconstruct 0 4 C:\Temp\Dullin_Aug_2012\sino_noflat C:\Temp\Dullin_Aug_2012\sino_
↪noflat\output
    9.0 10.0 0.0 0.0 0.0 true sino slice C:\Temp\Dullin_Aug_2012\sino_noflat\tomo_
↪conv flat dark

    """
    lock = Lock()
    # Get the from and to number of files to process:
    idx = int(argv[0])

    # Get input and output paths:
    inpath = argv[1]
    outfile = argv[2]

    if not inpath.endswith(sep): inpath += sep

    # Get parameters:
    convert_opt = argv[3]
    crop_opt = argv[4]
    crop_opt = '0:0:0:0'

    outprefix = argv[5]
    logfilename = argv[6]

```

```
    # Get the files in infile:
    files = sorted(glob(inpath + '*.tif*'))
    num_files = len(files)

    if ((idx >= num_files) or (idx == -1)):
        idx = num_files - 1

    # Read the image:
    im = imread(files[idx])

    # Process the image:
    im = postprocess(im, convert_opt, crop_opt)

    # Write down reconstructed preview file (file name modified with metadata):
    im = im.astype(float32)
    outfile = outfile + '_' + str(im.shape[1]) + 'x' + str(im.shape[0]) + '_' + str(␣
↪nanmin(im)) + '$' + str( nanmax(im) )
    im.tofile(outfile)


if __name__ == "__main__":
    main(argv[1:])
```

## preview_reconstruct

This section contains the preview_reconstruct script.

Download file: `preview_reconstruct.py`

```
################################################################################
# (C) 2016 Elettra - Sincrotrone Trieste S.C.p.A.. All rights reserved.        #
#                                                                              #
#                                                                              #
# This file is part of STP-Core, the Python core of SYRMEP Tomo Project,       #
# a software tool for the reconstruction of experimental CT datasets.          #
#                                                                              #
# STP-Core is free software: you can redistribute it and/or modify it          #
# under the terms of the GNU General Public License as published by the        #
# Free Software Foundation, either version 3 of the License, or (at your       #
# option) any later version.                                                   #
#                                                                              #
# STP-Core is distributed in the hope that it will be useful, but WITHOUT      #
# ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or        #
# FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License         #
# for more details.                                                            #
#                                                                              #
# You should have received a copy of the GNU General Public License            #
# along with STP-Core. If not, see <http://www.gnu.org/licenses/>.             #
#                                                                              #
################################################################################

#
# Author: Francesco Brun
# Last modified: Sept, 28th 2016
#

# python:
```

```
from sys import argv, exit
from os import remove, sep, linesep, listdir
from os.path import exists, dirname, basename, splitext
from numpy import array, finfo, copy, float32, double, amin, amax, tile, concatenate,␣
↪asarray, isscalar, pi
from numpy import empty, reshape, log as nplog, arange, squeeze, fromfile, ndarray,␣
↪where, meshgrid, roll
from time import time
from multiprocessing import Process, Array

# pystp-specific:
from stp_core.preprocess.extfov_correction import extfov_correction
from stp_core.preprocess.flat_fielding import flat_fielding
from stp_core.preprocess.dynamic_flatfielding import dff_prepare_plan, dynamic_flat_
↪fielding
from stp_core.preprocess.ring_correction import ring_correction
from stp_core.preprocess.extract_flatdark import extract_flatdark, _medianize

from stp_core.phaseretrieval.tiehom import tiehom, tiehom_plan
from stp_core.phaseretrieval.phrt   import phrt, phrt_plan

from stp_core.reconstruct.rec_astra import recon_astra_fbp, recon_astra_iterative
from stp_core.reconstruct.rec_fista_tv import recon_fista_tv
from stp_core.reconstruct.rec_mr_fbp import recon_mr_fbp
from stp_core.reconstruct.rec_gridrec import recon_gridrec

from stp_core.postprocess.postprocess import postprocess

from stp_core.utils.padding import upperPowerOfTwo, padImage, padSmoothWidth
from stp_core.utils.caching import cache2plan, plan2cache

from tifffile import imread, imsave
from h5py import File as getHDF5
import stp_core.io.tdf as tdf


def reconstruct(im, angles, offset, logtransform, recpar, circle, scale, pad, method,
                zerone_mode, dset_min, dset_max, corr_offset, rolling, roll_shift):
    """Reconstruct a sinogram with FBP algorithm (from ASTRA toolbox).

    Parameters
    ----------
    im1 : array_like
        Sinogram image data as numpy array.
    center : float
        Offset of the center of rotation to use for the tomographic
        reconstruction with respect to the half of sinogram width
        (default=0, i.e. half width).
    logtransform : boolean
        Apply logarithmic transformation before reconstruction (default=True).
    filter : string
        Filter to apply before the application of the reconstruction algorithm.␣
↪Filter
        types are: ram-lak, shepp-logan, cosine, hamming, hann, tukey, lanczos,␣
↪triangular,
        gaussian, barlett-hann, blackman, nuttall, blackman-harris, blackman-nuttall,
        flat-top, kaiser, parzen.
    circle : boolean
```

```
        Create a circle in the reconstructed image and set to zero pixels outside the
        circle (default=False).

    """
    offset = int(round(offset))

    # Upscale projections (if required):
    if (abs(scale - 1.0) > finfo(float32).eps):
        siz_orig1 = im.shape[1]
        im = imresize(im, (im.shape[0], int(round(scale * im.shape[1]))), interp=
↪'bicubic', mode='F')
        offset = int(offset * scale)

    # Apply transformation for changes in the center of rotation:
    if (offset != 0):
        if (offset >= 0):
            im = im[:,:-offset]

            tmp = im[:,0] # Get first column
            tmp = tile(tmp, (offset,1)) # Replicate the first column the right number␣
↪of times
            im = concatenate((tmp.T,im), axis=1) # Concatenate tmp before the image

        else:
            im = im[:,abs(offset):]

            tmp = im[:,im.shape[1] - 1] # Get last column
            tmp = tile(tmp, (abs(offset),1)) # Replicate the last column the right␣
↪number of times
            im = concatenate((im,tmp.T), axis=1) # Concatenate tmp after the image

    # Sinogram rolling (if required).  It doesn't make sense in limited angle␣
↪tomography, so check if 180 or 360:
    if ((rolling == True) and (roll_shift > 0)):
        if ( (angles - pi) < finfo(float32).eps ):
            # Flip the last rows:
            im[-roll_shift:,:] = im[-roll_shift:,::-1]
            # Now roll the sinogram:
            im = roll(im, roll_shift, axis=0)
        elif ((angles - pi*2.0) < finfo(float32).eps):
            # Only roll the sinogram:
            im = roll(im, roll_shift, axis=0)

    # Scale image to [0,1] range (if required):
    if (zerone_mode):

        #print dset_min
        #print dset_max
        #print numpy.amin(im_f[:])
        #print numpy.amax(im_f[:])
        #im_f = (im_f - dset_min) / (dset_max - dset_min)

        # Cheating the whole process:
        im = (im - numpy.amin(im[:])) / (numpy.amax(im[:]) - numpy.amin(im[:]))

    # Apply log transform:
    if (logtransform == True):
        im[im <= finfo(float32).eps] = finfo(float32).eps
```

```
        im = -nplog(im + corr_offset)

    # Replicate pad image to double the width:
    if (pad):

        dim_o = im.shape[1]
        n_pad = im.shape[1] + im.shape[1] / 2
        marg = (n_pad - dim_o) / 2

        # Pad image:
        im = padSmoothWidth(im, n_pad)

    # Perform the actual reconstruction:
    if (method.startswith('FBP')):
        im = recon_astra_fbp(im, angles, method, recpar)
    elif (method == 'MR-FBP_CUDA'):
        im = recon_mr_fbp(im, angles)
    elif (method == 'FISTA-TV_CUDA'):
        im = recon_fista_tv(im, angles, recpar, recpar)
    elif (method == 'GRIDREC'):
        [im, im] = recon_gridrec(im, im, angles, recpar)
    else:
        im = recon_astra_iterative(im, angles, method, recpar, zerone_mode)


    # Crop:
    if (pad):
        im = im[marg:dim_o + marg, marg:dim_o + marg]

    # Resize (if necessary):
    if (abs(scale - 1.0) > finfo(float32).eps):
        im = imresize(im, (siz_orig1, siz_orig1), interp='nearest', mode='F')

    # Return output:
    return im.astype(float32)


#def _testwritedownsino(tmp_im):

#    for ct in range(0, tmp_im.shape[0]):
#        a = tmp_im[ct,:,:].squeeze()
#        fname = 'C:\\Temp\\StupidFolder\\sino_' + str(ct).zfill(4) + '.tif'
#        imsave(fname, a.astype(float32))

#def _testwritedownproj(tmp_im):

#    for ct in range(0, tmp_im.shape[1]):
#        a = tmp_im[:,ct,:].squeeze()
#        fname = 'C:\\Temp\\StupidFolder\\proj_' + str(ct).zfill(4) + '.tif'
#        imsave(fname, a.astype(float32))

def process(sino_idx, num_sinos, infile, outfile, preprocessing_required, corr_plan,␣
↪skipflat, norm_sx, norm_dx, flat_end, half_half,
            half_half_line, ext_fov, ext_fov_rot_right, ext_fov_overlap, ringrem,␣
↪phaseretrieval_required, phrtmethod, phrt_param1,
            phrt_param2, energy, distance, pixsize, phrtpad, approx_win, angles,␣
↪angles_projfrom, angles_projto,
            offset, logtransform, recpar, circle, scale, pad, method, rolling, roll_
↪shift,
```

```
            zerone_mode, dset_min, dset_max, decim_factor, downsc_factor, corr_offset,
↪ postprocess_required, convert_opt,
            crop_opt, dynamic_ff, EFF, filtEFF, im_dark, nr_threads, logfilename):
    """To do...

    """
    # Perform reconstruction (on-the-fly preprocessing and phase retrieval, if
    # required):
    if (phaseretrieval_required):

        # In this case a bunch of sinograms is loaded into memory:

        #
        # Load the temporary data structure reading the input TDF file.
        # To know the right dimension the first sinogram is pre-processed.
        #

        # Open the TDF file and get the dataset:
        f_in = getHDF5(infile, 'r')
        if "/tomo" in f_in:
            dset = f_in['tomo']
        else:
            dset = f_in['exchange/data']

        # Downscaling and decimation factors considered when determining the
        # approximation window:
        zrange = arange(sino_idx - approx_win * downsc_factor / 2, sino_idx + approx_
↪win * downsc_factor / 2, downsc_factor)
        zrange = zrange[(zrange >= 0)]
        zrange = zrange[(zrange < num_sinos)]
        approx_win = zrange.shape[0]

        # Approximation window cannot be odd:
        if (approx_win % 2 == 1):
            approx_win = approx_win - 1
            zrange = zrange[0:approx_win]

        # Read one sinogram to get the proper dimensions:
        test_im = tdf.read_sino(dset, zrange[0]).astype(float32)

        # Apply projection removal (if required):
        test_im = test_im[angles_projfrom:angles_projto, :]

        # Apply decimation and downscaling (if required):
        test_im = test_im[::decim_factor, ::downsc_factor]

        # Perform the pre-processing of the first sinogram to get the right
        # dimension:
        if (preprocessing_required):
            if not skipflat:
                if dynamic_ff:
                    # Dynamic flat fielding with downsampling = 2:
                    test_im = dynamic_flat_fielding(test_im, zrange[0] / downsc_
↪factor, EFF, filtEFF, 2, im_dark, norm_sx, norm_dx)
                else:
                    test_im = flat_fielding(test_im, zrange[0] / downsc_factor, corr_
↪plan, flat_end, half_half,
                                            half_half_line / decim_factor, norm_sx,␣
↪norm_dx).astype(float32)
```

```
            test_im = extfov_correction(test_im, ext_fov, ext_fov_rot_right, ext_fov_
↪overlap / downsc_factor).astype(float32)
            if not skipflat and not dynamic_ff:
                test_im = ring_correction(test_im, ringrem, flat_end, corr_plan['skip_
↪flat_after'], half_half,
                                          half_half_line / decim_factor, ext_fov).
↪astype(float32)
            else:
                test_im = ring_correction(test_im, ringrem, False, False, half_half,
                                          half_half_line / decim_factor, ext_fov).
↪astype(float32)

        # Now we can allocate memory for the bunch of slices:
        tmp_im = empty((approx_win, test_im.shape[0], test_im.shape[1]),␣
↪dtype=float32)
        tmp_im[0,:,:] = test_im

        # Reading all the the sinos from TDF file and close:
        for ct in range(1, approx_win):

            # Read the sinogram:
            test_im = tdf.read_sino(dset, zrange[ct]).astype(float32)

            # Apply projection removal (if required):
            test_im = test_im[angles_projfrom:angles_projto, :]

            # Apply decimation and downscaling (if required):
            test_im = test_im[::decim_factor, ::downsc_factor]

            # Perform the pre-processing for each sinogram of the bunch:
            if (preprocessing_required):
                if not skipflat:
                    if dynamic_ff:
                        # Dynamic flat fielding with downsampling = 2:
                        test_im = dynamic_flat_fielding(test_im, zrange[ct] / downsc_
↪factor, EFF, filtEFF, 2, im_dark, norm_sx, norm_dx)
                    else:
                        test_im = flat_fielding(test_im, zrange[ct] / downsc_factor,␣
↪corr_plan, flat_end, half_half,
                                                half_half_line / decim_factor, norm_sx,␣
↪norm_dx).astype(float32)
                test_im = extfov_correction(test_im, ext_fov, ext_fov_rot_right, ext_
↪fov_overlap / downsc_factor).astype(float32)
                if not skipflat and not dynamic_ff:
                    test_im = ring_correction(test_im, ringrem, flat_end, corr_plan[
↪'skip_flat_after'], half_half,
                                              half_half_line / decim_factor, ext_fov).
↪astype(float32)
                else:
                    test_im = ring_correction(test_im, ringrem, False, False, half_
↪half,
                                              half_half_line / decim_factor, ext_fov).
↪astype(float32)

            tmp_im[ct,:,:] = test_im

        f_in.close()

```

```
290         # Now everything has to refer to a downscaled dataset:
291         sino_idx = ((zrange == sino_idx).nonzero())
292 
293         #
294         # Perform phase retrieval:
295         #
296 
297         # Prepare the plan:
298         if (phrtmethod == 0):
299             # Paganin's:
300             phrtplan = tiehom_plan(tmp_im[:,0,:], phrt_param1, phrt_param2, energy,␣
    ↪distance, pixsize * downsc_factor, phrtpad)
301         else:
302             phrtplan = phrt_plan(tmp_im[:,0,:], energy, distance, pixsize * downsc_
    ↪factor, phrt_param2, phrt_param1, phrtmethod, phrtpad)
303             #phrtplan = prepare_plan (tmp_im[:,0,:], beta, delta, energy, distance,
304             #pixsize*downsc_factor, padding=phrtpad)
305 
306         # Process each projection (whose height depends on the size of the bunch):
307         for ct in range(0, tmp_im.shape[1]):
308             #tmp_im[:,ct,:] = phase_retrieval(tmp_im[:,ct,:], phrtplan).
    ↪astype(float32)
309             if (phrtmethod == 0):
310                 tmp_im[:,ct,:] = tiehom(tmp_im[:,ct,:], phrtplan).astype(float32)
311             else:
312                 tmp_im[:,ct,:] = phrt(tmp_im[:,ct,:], phrtplan, phrtmethod).
    ↪astype(float32)
313 
314         # Extract the requested sinogram:
315         im = tmp_im[sino_idx[0],:,:].squeeze()
316 
317     else:
318 
319         # Read only one sinogram:
320         f_in = getHDF5(infile, 'r')
321         if "/tomo" in f_in:
322             dset = f_in['tomo']
323         else:
324             dset = f_in['exchange/data']
325         im = tdf.read_sino(dset,sino_idx).astype(float32)
326         f_in.close()
327 
328         # Apply projection removal (if required):
329         im = im[angles_projfrom:angles_projto, :]
330 
331         # Apply decimation and downscaling (if required):
332         im = im[::decim_factor,::downsc_factor]
333         sino_idx = sino_idx / downsc_factor
334 
335         # Perform the preprocessing of the sinogram (if required):
336         if (preprocessing_required):
337             if not skipflat:
338                 if dynamic_ff:
339                     # Dynamic flat fielding with downsampling = 2:
340                     im = dynamic_flat_fielding(im, sino_idx, EFF, filtEFF, 2, im_dark,
    ↪ norm_sx, norm_dx)
341                 else:
342                     im = flat_fielding(im, sino_idx, corr_plan, flat_end, half_half,␣
    ↪half_half_line / decim_factor,
```

```
                                norm_sx, norm_dx).astype(float32)
            im = extfov_correction(im, ext_fov, ext_fov_rot_right, ext_fov_overlap)
            if not skipflat and not dynamic_ff:
                im = ring_correction(im, ringrem, flat_end, corr_plan['skip_flat_after
↪'], half_half,
                                half_half_line / decim_factor, ext_fov)
            else:
                im = ring_correction(im, ringrem, False, False, half_half,
                                half_half_line / decim_factor, ext_fov)


    # Additional ring removal before reconstruction:
    #im = boinhaibel(im, '11;')
    #im = munchetal(im, '5;1.8')
    #im = rivers(im, '13;')
    #im = raven(im, '11;0.8')
    #im = oimoen(im, '51;51')

    # Actual reconstruction:
    im = reconstruct(im, angles, offset / downsc_factor, logtransform, recpar, circle,
↪ scale, pad, method,
                     zerone_mode, dset_min, dset_max, corr_offset, rolling, roll_
↪shift).astype(float32)

    # Apply post-processing (if required):
    if postprocess_required:
        im = postprocess(im, convert_opt, crop_opt)
    else:
        # Create the circle mask for fancy output:
        if (circle == True):
            siz = im.shape[1]
            if siz % 2:
                rang = arange(-siz / 2 + 1, siz / 2 + 1)
            else:
                rang = arange(-siz / 2,siz / 2)
            x,y = meshgrid(rang,rang)
            z = x ** 2 + y ** 2
            a = (z < (siz / 2 - int(round(abs(offset) / downsc_factor))) ** 2)
            im = im * a

    # Write down reconstructed preview file (file name modified with metadata):
    im = im.astype(float32)
    outfile = outfile + '_' + str(im.shape[1]) + 'x' + str(im.shape[0]) + '_' +␣
↪str(amin(im)) + '$' + str(amax(im))
    im.tofile(outfile)

    #print "With %d thread(s): [%0.3f sec, %0.3f sec, %0.3f sec]." % (nr_threads,
    #t1-t0, t2-t1, t3-t2)

def main(argv):
    """To do...

    Usage
    -----


    Parameters
    ----------
```

```
    Example
    -------------------------


    """
    # Get the from and to number of files to process:
    sino_idx = int(argv[0])

    # Get paths:
    infile = argv[1]
    outfile = argv[2]

    # Essential reconstruction parameters:
    angles = float(argv[3])
    offset = float(argv[4])
    recpar = argv[5]
    scale = int(float(argv[6]))

    overpad = True if argv[7] == "True" else False
    logtrsf = True if argv[8] == "True" else False
    circle = True if argv[9] == "True" else False

    # Parameters for on-the-fly pre-processing:
    preprocessing_required = True if argv[10] == "True" else False
    flat_end = True if argv[11] == "True" else False
    half_half = True if argv[12] == "True" else False

    half_half_line = int(argv[13])

    ext_fov = True if argv[14] == "True" else False

    norm_sx = int(argv[17])
    norm_dx = int(argv[18])

    ext_fov_rot_right = argv[15]
    if ext_fov_rot_right == "True":
        ext_fov_rot_right = True
        if (ext_fov):
            norm_sx = 0
    else:
        ext_fov_rot_right = False
        if (ext_fov):
            norm_dx = 0

    ext_fov_overlap = int(argv[16])

    skip_ringrem = True if argv[19] == "True" else False
    ringrem = argv[20]

    # Extra reconstruction parameters:
    zerone_mode = True if argv[21] == "True" else False
    corr_offset = float(argv[22])

    reconmethod = argv[23]

    decim_factor = int(argv[24])
    downsc_factor = int(argv[25])
```

```
    # Parameters for postprocessing:
    postprocess_required = True if argv[26] == "True" else False
    convert_opt = argv[27]
    crop_opt = argv[28]

    # Parameters for on-the-fly phase retrieval:
    phaseretrieval_required = True if argv[29] == "True" else False
    phrtmethod = int(argv[30])
    phrt_param1 = double(argv[31])    # param1( e.g.  regParam, or beta)
    phrt_param2 = double(argv[32])    # param2( e.g.  thresh or delta)
    energy = double(argv[33])
    distance = double(argv[34])
    pixsize = double(argv[35]) / 1000.0 # pixsixe from micron to mm:
    phrtpad = True if argv[36] == "True" else False
    approx_win = int(argv[37])

    angles_projfrom = int(argv[38])
    angles_projto = int(argv[39])

    rolling = True if argv[40] == "True" else False
    roll_shift = int(argv[41])

    preprocessingplan_fromcache = True if argv[42] == "True" else False
    dynamic_ff = True if argv[43] == "True" else False

    nr_threads = int(argv[44])
    tmppath = argv[45]
    if not tmppath.endswith(sep): tmppath += sep

    logfilename = argv[46]

    # Open the HDF5 file:
    f_in = getHDF5(infile, 'r')
    if "/tomo" in f_in:
        dset = f_in['tomo']
    else:
        dset = f_in['exchange/data']
        if "/provenance/detector_output" in f_in:
            prov_dset = f_in['provenance/detector_output']

    dset_min = -1
    dset_max = -1
    if (zerone_mode):
        if ('min' in dset.attrs):
            dset_min = float(dset.attrs['min'])
        else:
            zerone_mode = False

        if ('max' in dset.attrs):
            dset_max = float(dset.attrs['max'])
        else:
            zerone_mode = False

    num_sinos = tdf.get_nr_sinos(dset) # Pay attention to the downscale factor

    if (num_sinos == 0):
        exit()
```

```
513
514     # Check extrema:
515     if (sino_idx >= num_sinos):
516         sino_idx = num_sinos - 1
517
518     # Get correction plan and phase retrieval plan (if required):
519     skipflat = False
520
521     corrplan = 0
522     im_dark = 0
523     EFF = 0
524     filtEFF = 0
525     if (preprocessing_required):
526         if not dynamic_ff:
527             # Load flat fielding plan either from cache (if required) or from TDF file
528             # and cache it for faster re-use:
529             if (preprocessingplan_fromcache):
530                 try:
531                     corrplan = cache2plan(infile, tmppath)
532                 except Exception as e:
533                     #print "Error(s) when reading from cache"
534                     corrplan = extract_flatdark(f_in, flat_end, logfilename)
535                     if (isscalar(corrplan['im_flat']) and isscalar(corrplan['im_flat_
↪after'])):
536                         skipflat = True
537                     else:
538                         plan2cache(corrplan, infile, tmppath)
539             else:
540                 corrplan = extract_flatdark(f_in, flat_end, logfilename)
541                 if (isscalar(corrplan['im_flat']) and isscalar(corrplan['im_flat_after
↪'])):
542                     skipflat = True
543                 else:
544                     plan2cache(corrplan, infile, tmppath)
545
546             # Dowscale flat and dark images if necessary:
547             if isinstance(corrplan['im_flat'], ndarray):
548                 corrplan['im_flat'] = corrplan['im_flat'][::downsc_factor,::downsc_
↪factor]
549             if isinstance(corrplan['im_dark'], ndarray):
550                 corrplan['im_dark'] = corrplan['im_dark'][::downsc_factor,::downsc_
↪factor]
551             if isinstance(corrplan['im_flat_after'], ndarray):
552                 corrplan['im_flat_after'] = corrplan['im_flat_after'][::downsc_factor,
↪::downsc_factor]
553             if isinstance(corrplan['im_dark_after'], ndarray):
554                 corrplan['im_dark_after'] = corrplan['im_dark_after'][::downsc_factor,
↪::downsc_factor]
555
556         else:
557             # Dynamic flat fielding:
558             if "/tomo" in f_in:
559                 if "/flat" in f_in:
560                     flat_dset = f_in['flat']
561                     if "/dark" in f_in:
562                         im_dark = _medianize(f_in['dark'])
563                     else:
564                         skipdark = True
```

```
                else:
                    skipflat = True # Nothing to do in this case
            else:
                if "/exchange/data_white" in f_in:
                    flat_dset = f_in['/exchange/data_white']
                    if "/exchange/data_dark" in f_in:
                        im_dark = _medianize(f_in['/exchange/data_dark'])
                    else:
                        skipdark = True
                else:
                    skipflat = True # Nothing to do in this case

            # Prepare plan for dynamic flat fielding with 16 repetitions:
            if not skipflat:
                EFF, filtEFF = dff_prepare_plan(flat_dset, 16, im_dark)

                # Downscale images if necessary:
                im_dark = im_dark[::downsc_factor,::downsc_factor]
                EFF = EFF[::downsc_factor,::downsc_factor,:]
                filtEFF = filtEFF[::downsc_factor,::downsc_factor,:]

    f_in.close()

    # Run computation:
    process(sino_idx, num_sinos, infile, outfile, preprocessing_required, corrplan, skipflat, norm_sx,
                norm_dx, flat_end, half_half, half_half_line, ext_fov, ext_fov_rot_right, ext_fov_overlap, ringrem,
                phaseretrieval_required, phrtmethod, phrt_param1, phrt_param2, energy, distance, pixsize, phrtpad, approx_win, angles,
                angles_projfrom, angles_projto, offset,
                logtrsf, recpar, circle, scale, overpad, reconmethod,
                rolling, roll_shift,
                zerone_mode, dset_min, dset_max, decim_factor,
                downsc_factor, corr_offset, postprocess_required, convert_opt, crop_opt, dynamic_ff, EFF, filtEFF, im_dark, nr_threads, logfilename)

    # Sample:
    # 311 C:\Temp\BrunGeorgos.tdf C:\Temp\BrunGeorgos.raw 3.1416 -31.0 shepp-logan
    # 1.0 False False True True True True 5 False False 100 0 0 False rivers:11;0
    # False 0.0 FBP_CUDA 1 1 False - - True 5 1.0 1000.0 22 150 2.2 True 16 0 1799
    # True True 2 C:\Temp\StupidFolder C:\Temp\log_00.txt


if __name__ == "__main__":
    main(argv[1:])
```

## preview_phaseretrieval

This section contains the preview_phaseretrieval script.

Download file: `preview_phaseretrieval.py`

```
##########################################################################
# (C) 2016 Elettra - Sincrotrone Trieste S.C.p.A.. All rights reserved.   #
#                                                                        #
#                                                                        #
```

```
# This file is part of STP-Core, the Python core of SYRMEP Tomo Project,  #
# a software tool for the reconstruction of experimental CT datasets.     #
#                                                                          #
# STP-Core is free software: you can redistribute it and/or modify it      #
# under the terms of the GNU General Public License as published by the    #
# Free Software Foundation, either version 3 of the License, or (at your   #
# option) any later version.                                               #
#                                                                          #
# STP-Core is distributed in the hope that it will be useful, but WITHOUT  #
# ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or    #
# FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License     #
# for more details.                                                        #
#                                                                          #
# You should have received a copy of the GNU General Public License        #
# along with STP-Core. If not, see <http://www.gnu.org/licenses/>.         #
#                                                                          #
############################################################################

#
# Author: Francesco Brun
# Last modified: July, 8th 2016
#

from sys import argv, exit
from os import remove, sep, linesep
from os.path import exists
from numpy import float32, double, nanmin, nanmax, finfo, ndarray
from time import time
from multiprocessing import Process, Lock
from pyfftw.interfaces.cache import enable as pyfftw_cache_enable, disable as pyfftw_
↪cache_disable
from pyfftw.interfaces.cache import set_keepalive_time as pyfftw_set_keepalive_time

# pystp-specific:
from stp_core.phaseretrieval.tiehom import tiehom, tiehom_plan
from stp_core.phaseretrieval.phrt   import phrt, phrt_plan

from h5py import File as getHDF5
from stp_core.utils.caching import cache2plan, plan2cache
from stp_core.preprocess.extract_flatdark import extract_flatdark
import stp_core.io.tdf as tdf


def main(argv):
    """To do...

    """
    lock = Lock()

    skip_flat = True
    first_done = False
    pyfftw_cache_disable()
    pyfftw_cache_enable()
    pyfftw_set_keepalive_time(1800)

    # Get the from and to number of files to process:
    idx = int(argv[0])

```

```
    # Get full paths of input TDF and output TDF:
    infile = argv[1]
    outfile = argv[2]

    # Get the phase retrieval parameters:
    method = int(argv[3])
    param1 = double(argv[4])    # param1( e.g. regParam, or beta)
    param2 = double(argv[5])    # param2( e.g. thresh or delta)
    energy = double(argv[6])
    distance = double(argv[7])
    pixsize = double(argv[8]) / 1000.0 # pixsixe from micron to mm:
    pad = True if argv[9] == "True" else False

    # Tmp path and log file:
    tmppath = argv[10]
    if not tmppath.endswith(sep): tmppath += sep
    logfilename = argv[11]


    # Open the HDF5 file:
    f_in = getHDF5(infile, 'r')
    if "/tomo" in f_in:
        dset = f_in['tomo']
    else:
        dset = f_in['exchange/data']
    num_proj = tdf.get_nr_projs(dset)
    num_sinos = tdf.get_nr_sinos(dset)

    # Check if the HDF5 makes sense:
    if (num_proj == 0):
        log = open(logfilename,"a")
        log.write(linesep + "\tNo projections found. Process will end.")
        log.close()
        exit()


    # Get flats and darks from cache or from file:
    try:
        corrplan = cache2plan(infile, tmppath)
    except Exception as e:
        #print "Error(s) when reading from cache"
        corrplan = extract_flatdark(f_in, True, logfilename)
        remove(logfilename)
        plan2cache(corrplan, infile, tmppath)

    # Read projection:
    im = tdf.read_tomo(dset,idx).astype(float32)
    f_in.close()

    # Apply simple flat fielding (if applicable):
    if (isinstance(corrplan['im_flat_after'], ndarray) and isinstance(corrplan['im_
↪flat'], ndarray) and
        isinstance(corrplan['im_dark'], ndarray) and isinstance(corrplan['im_dark_
↪after'], ndarray)) :
        if (idx < num_proj/2):
            im = (im - corrplan['im_dark']) / (abs(corrplan['im_flat'] -␣
↪corrplan['im_dark']) + finfo(float32).eps)
        else:
```

```
            im = (im - corrplan['im_dark_after']) /␣
↪(abs(corrplan['im_flat_after'] - corrplan['im_dark_after'])
                + finfo(float32).eps)

    # Prepare plan:
    im = im.astype(float32)
    if (method == 0):
        # Paganin's:
        plan = tiehom_plan (im, param1, param2, energy, distance, pixsize, pad)
        im = tiehom(im, plan).astype(float32)
    else:
        plan = phrt_plan (im, energy, distance, pixsize, param2, param1,␣
↪method, pad)
        im = phrt(im, plan, method).astype(float32)

    # Write down reconstructed preview file (file name modified with␣
↪metadata):
    im = im.astype(float32)
    outfile = outfile + '_' + str(im.shape[1]) + 'x' + str(im.shape[0]) + '_' +␣
↪str( nanmin(im)) + '$' + str( nanmax(im) )
    im.tofile(outfile)

if __name__ == "__main__":
    main(argv[1:])
```

# Credits

## Citations

We kindly request that you cite the following article *[A1]* if you use project.

## References

# Bibliography

[A1] Francesco Brun, Serena Pacile, Agostino Accardo, George Kourousias, Diego Dreossi, Lucia Mancini, Giuliana Tromba, and Roberto Pugliese. Enhanced and flexible software tools for x-ray computed tomography at the italian synchrotron radiation facility elettra. *Fundamenta Informaticae*, 141(2-3):233–243, Oct 2015. URL: http://doi.org/10.3233/FI-2015-1273, doi:10.3233/FI-2015-1273.

[B1] Francesco Brun, Serena Pacile, Agostino Accardo, George Kourousias, Diego Dreossi, Lucia Mancini, Giuliana Tromba, and Roberto Pugliese. Enhanced and flexible software tools for x-ray computed tomography at the italian synchrotron radiation facility elettra. *Fundamenta Informaticae*, 141(2-3):233–243, Oct 2015. URL: http://doi.org/10.3233/FI-2015-1273, doi:10.3233/FI-2015-1273.

# Python Module Index

## S

# Index

## Z